Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informatiky a kvantitativních metod

Vývoj webových aplikací na platformě ASP.NET MVC a Single Page Application

Diplomová práce

Autor: Přemysl Šulc

Studijní obor: I1450 – P – IM2

Vedoucí práce: Ing. Jiří Štěpánek

Hradec Králové Listopad 2017

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 12. listopadu 2017 Přemysl Šulc

Poděkování

Touto cestou bych rád poděkoval panu Ing. Jiřímu Štěpánkovi za konzultace, velmi cenné rady a odborné vedení této diplomové práce. Dále bych rád poděkoval své drahé rodině, za podporu, jež mi byla během studia velkou oporou. V neposlední řadě také děkuji Andree Šulcové za kvalitní korektorskou práci a další užitečné rady při psaní diplomové práce.

Anotace

Téma: Vývoj webových aplikací na platformě ASP.NET MVC a Single Page Application

Diplomová práce se zabývá porovnáním tradičního přístupu k vývoji webových aplikací na platformě ASP.NET MVC s moderní a stále populárnější technologií Single Page Application. Veškeré informace jsou čerpané z mezinárodních knižních zdrojů, oficiálních dokumentací probíraných technologií a jako ukázky jsou použity zdrojové kódy vlastní tvorby. Práce se zabývá popisem platformy ASP.NET MVC a všech ostatních .NET technologií, na kterých ASP.NET MVC stojí. Dále je detailně popsána architektura Single Page Appliaction a analyzovány výhody, přínosy a nedostatky jednotlivých technologií. Práce je podložena praktickými ukázkami z vlastní aplikace, popřípadě typovými příklady vytvořenými pro specifické situace.

Annotation

Title: Development of web applications on ASP.NET MVC platform and Single Page Application

The diploma thesis focuses on comparison of two ways of web applications development. Traditional approach based on ASP.NET MVC platform is compared with modern and currently more and more popular Single Page Application technology. All the information is gathered from foreign book sources, official documentations of above mentioned technologies, and the source codes that are used as demonstrative illustrations are made by the author of the thesis himself. The diploma thesis is dedicated to the ASP.NET MVC platform description and other .NET technologies, which ASP.NET MVC is based on. Furthermore, Single Page Application architecture is scrutinized in detail, and the advantages, benefits and shortcomings of individual technologies are described. The work is supported by practical examples of the application designed by the author, or by examples made for specific situations.

Obsah

Seznam obrázků a tabulek

Seznam ukázkových zdrojových kódů

Úvod

V dnešní době, kdy je lidstvo stále více závislé na IT technologiích, se prakticky vše přesouvá na internet. Jsou to právě webové a mobilní aplikace, které jsou v dnešním moderním světě součástí každodenního profesního i osobního života. Stále větší procento populace takové aplikace používá a poptávka po nich neustále roste. V důsledku toho se obecně zvyšuje zájem o IT sektor, což má za následek podstatně rychlejší rozvoj technologií, než tomu bylo dříve. Postupně vzrůstající úroveň hardware a software technologií nabízí také prostor pro inovaci na poli vývoje webových aplikací.

Stávající úroveň hardware produktů umožňuje vytvářet webové aplikace, které jsou náročnější pro klientskou stranu. Pro vývoj klientských částí webových aplikací je nejrozšířenějším jazykem JavaScript, který zaznamenal v posledních deseti letech velký progres. Díky tomu se začal rozvíjet velmi moderní koncept jednostránkových aplikací, jimž se výstižně říká Single Page Application. Jedná se o přístup, kdy je webová aplikace tvořena pouze jedinou HTML stránkou, jež je plněna JavaScriptem na straně klienta za použití dat přicházejících ze serveru.

JavaScript byl dříve považován za mrtvý programovací jazyk bez budoucnosti, avšak dnes je velmi populární. Architektura Single Page Application byla diskutována i ve vzdálenější minulosti, nicméně její implementace nebyla z důvodu nevyzrálosti hardware a JavaScriptových technologií možná, nebo při nejmenším vhodná. Největší vzestup byl zaznamenán až v tomto desetiletí, kdy začala vznikat spousta JavaScriptových knihoven, frameworků a dalších nástrojů, kolem kterých se tvoří rozsáhlé komunity. Single Page Application se staly velmi moderním a podporovaným přístupem zejména díky své schopnosti rychlého vykreslení HTML a oddělení prezenční vrstvy od datové. Největší důraz je tak kladen na výkon a rychlost webových stránek, aby přinesly uživateli příjemný a plynulý zážitek.

To vše je důvodem k zamyšlení, zda jsou Single Page Application vhodnou alternativou nebo dokonce náhradou za klasické webové aplikace, kdy je výsledné HTML při každém requestu generováno na serveru a odesláno na klientskou stranu k zobrazení.

Metodika práce a cíl práce

Hlavním cílem této diplomové práce je analýza a porovnání dvou různých přístupů a technologií pro tvorbu webových aplikací. Jedná se o srovnání vývoje klasických webových aplikací na platformě ASP.NET MVC s tvorbou Single page aplikací za použití JavaScriptové knihovny React. Cílem práce je popsat zkoumané technologie, nastínit odlišnosti, přínosy a nedostatky z hlediska vlastností i samotného procesu vývoje a vše podložit příkladovými ukázkami kódu, popřípadě kódu z vlastní aplikace. Zároveň jsou představeny moderní a oblíbené nástroje, jenž se při vývoji v těchto technologiích používají. Autor si klade za cíl poskytnout čtenáři dostatečně zpracované materiály, aby popisovaným technologiím porozuměl a byly mu užitečným průvodcem při jejich použití.

První dvě kapitoly se věnují popisu samotné ASP.NET MVC platformy. Jelikož je framework ASP.NET MVC součástí většího celku, jsou zde pro komplexní porozumění problematice analyzovány i frameworky .NET a ASP.NET, kterých je ASP.NET MVC součástí a je tedy nutné je znát. V této části diplomové práce je kladen důraz na specifikaci frameworků, nastínění historie a evoluce, charakterizována je také jejich struktura, jsou vysvětleny mechanismy, a rovněž se autor zabývá stěžejní terminologií a životním cyklem zpracování zdrojového kódu.

Následující segment se zabývá charakteristikou Single Page Application. Mimo samotného vymezení jsou zde rozebrány její přínosy a nedostatky, popsány populární technologie a nástroje používané v rámci vývoje SPA. Dále jsou vysvětleny principiální rozdíly mezi klasickými webovými aplikacemi a Single page aplikacemi a v neposlední řadě je podrobně představena knihovna React, jež je pro vývoj příkladové Single page aplikace použita.

Poslední kapitola diplomové práce předvádí výše popsanou problematiku prakticky, na příkladech uceleného řešení vlastní aplikace. Příkladová aplikace je velmi odlehčenou formou e-shopu, respektive plní úlohu online katalogu zboží hudebnin, avšak v budoucnu se na e-shop rozšíří. Jelikož je jedním z cílů diplomové práce ukázat použití probíraných technologií v praxi, nikoli vytvořit a předat konkrétní aplikaci, projekt tak slouží pouze jako zdroj demonstrativních ukázek. Čtenáři se tak dostává náhled struktury projektu a ukázky použití jednotlivých technologií a nástrojů při vývoji dílčích segmentů aplikace.

1 Úvod do ASP.NET MVC

Technologie ASP.NET MVC je součástí větších technologických celků. Pro správné porozumění je hned v samotném úvodu nutné pojem ASP.NET MVC rozvést a rozdělit jej na jednotlivé technologie. Jedná se o .NET framework, ASP.NET framework, architekturu MVC a samotný ASP.NET MVC framework. Zmíněným termínům jsou věnovány samostatné kapitoly či podkapitoly, kde jsou definovány a analyzovány. První kapitola se tedy zabývá technologiemi .NET a následně ASP.NET, jichž je framework ASP.NET MVC součástí.

1.1 .NET Framework

Tato kapitola je dedikována charakteristice a vývoji .NET frameworku, který je stavebním kamenem pro všechny ostatní technologie postavené na .NET platformě.

1.1.1 Vývoj a charakteristika

.NET Framework byl vytvořen společností Microsoft jakožto komplexní soubor nástrojů pro vývoj webových i desktopových aplikací. Aplikace postavené na platformě .NET mají stejný model návrhu a stejné rysy, ať už se jedná o aplikace desktopové, webové či velmi výkonné serverové produkty. Platforma .NET byla oficiálně uvedena na trh v roce 2002 společně s programovacím jazykem C#[1]. [1]

Vývojáři, kteří dříve vytvářeli aplikace pro operační systém Windows, do té doby převážně používali *COM model* (Component Object Model), jenž byl vydán společností Microsoft v roce 1993. Za předchůdce .NET frameworku se označuje právě COM model, a to i přes to, že se principiálně relativně liší. Nebyl multiplatformní, nicméně umožňoval vývojářům vytvářet knihovny, které bylo možné používat napříč různými programovacími jazyky. To v praxi znamenalo, že knihovna navržená v programovacím jazyce Visual Basic mohla být použita v projektu, který byl psán v jazyce C++. Vlastnost jazykové nezávislosti byla bezpochyby velmi silnou stránkou Component Object Modelu, avšak nesl s sebou i podstatné nedostatky. Disponoval velmi komplikovanou infrastrukturou, křehkým vývojovým modelem a byl určen pouze pro operační systém Windows. Přestože byly na

[1] C# - vysokoúrovňový objektově orientovaný jazyk od společnosti Microsoft.

jeho základech vytvořené úspěšné aplikace, narůstající nutnost evoluce vyústila v návrh a následně v roce 2002 zveřejnění první verze .NET frameworku. [2]

.NET byl navržen tak, aby přinesl efektivnější nástroje, větší flexibilitu a aby byl samotný vývoj oproti práci s COM snazší.

V rámci této diplomové práce byl k vývoji zvolen jazyk C#, nicméně framework .NET podporuje více programovacích jazyků. Mezi ty nejrozšířenější patří Visual Basic, F#, Managed C++, ale umožňuje použití i dalších. Všechny zmíněné jazyky nejsou pevně svázány s .NET platformou. Jedinou výjimkou je jazyk C#, jenž byl původně předurčen pro práci na .NET frameworku. Mezi jeho nesporné výhody patří *cross-language interoperability*, neboli možnost využití mezi-jazykové kooperace. To znamená, že lze v rámci projektu použít knihovny napsané v různých programovacích jazycích, které .NET podporuje.

To je možné zásluhou frameworku *CLS* (Common Language Specification), jenž je součástí .NET vrstvy a díky mechanismu, jakým se v prostředí .NET zpracovává zdrojový kód. V rámci životního cyklu je kód jazyka vyšší úrovně (C#, VB, F# apod.) kompilován do jazyka nižší úrovně (mezikódu), platformě neutrálního jazyka CIL (Common Intermediate Language). Tento mezikód často nese označení MSIL (Microsoft Intermediate Language), které je už poměrně zastaralé a dnes se příliš často nepoužívá. V jeho tvaru mají instrukce kompilovaných kódů z různých programovacích jazyků totožnou podobu. Procesor, jenž pak následně instrukční sady vykonává, neví, že původní zdrojový kód je psán v různých programovacích jazycích a bez problému proces vykoná. Aplikace napsané v .NET frameworku běží na CLR (Common Language Runtime) běhovém prostředí, což je virtuální stroj, který součástí jádra .NET frameworku. Kompletní životní cyklus zpracování zdrojového kódu je podrobněji popsán níže v kapitole 1.1.3.[3]

1.1.2 Struktura .NET frameworku

Framework .NET je charakteristický poměrně robustní, komplexní a složitější strukturou. Základními a stěžejními stavebními prvky jsou sady knihoven *CL* (Class Library), aplikační virtuální stroj *CLR* (Common Language Runtime), framework *Common Language Specification* (CLS) a *Common Type System* (CTS). [12]

Pro práci na .NET platformě bylo společně s první verzí frameworku vytvořeno společností Microsoft i vývojové prostředí Microsoft Visual Studio. První verze, publikována v roce 2002, nesla název *Visual Studio .NET* a byla vydána v několika edicích: *Academic, Professional, Enterprise Developers* nebo *Enterprise Architect*. Zároveň Microsoft představil i nový programovací jazyk *C#*. Vývojové prostředí společně s jazykem C# a ostatními výše zmíněnými jazyky je průběžně updatováno v rámci častých aktualizací .NET frameworků nebo jeho podmnožin. Pro správnou funkčnost aplikace je nezbytné, aby byla na koncové stanici nasazena stejná verze frameworku, ve které byla vytvořena, popřípadě verze s ní kompatibilní. Microsoft tento fakt bere v potaz a do svých operačních systémů Windows vždy integruje aktuální verzi frameworku, která je se staršími verzemi kompatibilní. Případně lze potřebnou konkrétní verzi stáhnout a doinstalovat. Starší verze operačního systému Microsoft Windows, jako například Windows XP nebo Windows Vista, .NET podporovaly, avšak framework nebyl součástí instalace operačního systému. [2]

1.1.2.1 Common Language Runtime

Common Language Runtime je základní stavební jednotka, která řídí a provádí programy běžící na platformě .NET. CLR je označení pro virtuální stroj na platformě .NET a kód běžící pod jeho kontrolou se často označuje jako *managed code*. Dříve se v tomto spojení nepoužívala obecná terminologie virtuálního stroje *Virtual Machine*, ale *Execution Engine*. Důvodem je terminologická podobnost s Java platformou, kde se prostředí pro běh programu nazývá *Java Virtual Machine*. Dnes je již zažité označení *virtual machine* jako obecná terminologie pro běhové prostředí na všech platformách, ne pouze pro Javu. CLR zároveň poskytuje služby a výhody, které činí proces vývoje jednodušší a bezpečnější. Primárním úloha CLR spočívá v hledání, načítání a řízení objektů. Jeho součástí je *Code Manager* a *Garbage Collector*. Mimo řízení běhu aplikací a provádění kódů se také stará o řízení paměti, dostupných zdrojů a *Code Access Security* (CAS). Zároveň se stará o další nízkoúrovňové operace včetně řízení aplikačního hostingu, koordinace vláken nebo provádění základních bezpečnostních kontrol. Na stejné úrovni struktury se nachází *Just In Time Compiler* (JIT compiler), který mezikód CIL převádí do nativního strojového kódu, jenž dále procesor provádí. [9]

Virtuální stroj přináší řadu výhod:

- Zlepšení výkonnosti,
- Jednoduché použití komponent vytvořených v jiném jazyce,
- Dědičnost, rozhraní a přetížení pro objektově orientované programování,
- Podpora více vláken a škálovatelnosti aplikace,
- Podpora strukturovaného řešení výjimek,
- Použití Garbage Collectoru.

Garbage Collector

Garbage Collector je podstatná a silná komponenta. V rámci svých aktivit se GC automaticky stará o management paměti a odstiňuje tak programátora od manuální dealokace přidělené, ale nepoužívané paměti. Principem činnosti je nalezení paměti přidělené objektům, které už program nepoužívá a následné uvolnění neboli dealokace. Takové ošetření znatelně zlepšuje práci s například nekvalitně napsaným zdrojovým kódem. Nemělo by dojít k naprostému vyčerpání systémových prostředků a následnému zhroucení serveru. [4]

1.1.2.2 Class Library

Důležitou součástí .NET frameworku je sada standardních knihoven. Třídy knihoven jsou organizovány do hierarchie jmenných prostorů (*namespace*). Většina vestavěných API jsou součástí „ System.* “ nebo „ Microsoft.* “ jmenných prostorů. Zde je implementováno velké množství běžných funkcionalit, jako je grafické renderování, interakce s databází, čtení a zápis souborů (I/O), práce s dokumenty typu XML, nástroje pro práci s formuláři, konzolí, autentizaci, popřípadě zajištění bezpečnosti a další. Všechny předpřipravené komponenty jsou dostupné pro veškeré programovací jazyky kompatibilní s platformou .NET. Class Library se dále dělí na Framework Class Library (FCL) a Base Class Library (BCL).

Framework Class Library odkazuje na veškeré knihovny poskytované .NET frameworkem. Zahrnuje v sobě rozsáhlé knihovny a frameworky, které jsou mnohem větší než standartní knihovny v porovnání s knihovnami pro jazyky C++ nebo JAVA. Mezi

vestavěné komponenty patří *ASP.NET, Windows Forms, Windows Presentation Foundation* (WPF), dále jsou to rozšíření k základním knihovnám *LINQ, ADO.NET, Workflow Foundation* (WF) a *Windows Communication Foundation* (WCF). [5] [2]

Jak vyplývá z textu výše, **Base Class Library** je pouze malá podmnožina Class Library. Do BCL spadají pouze stěžejní knihovny, což jsou v tomto případě základní API aplikačního virtuálního stroje CLR. Většina tříd spadajících pod hlavičku BCL je umístěna v knihovnách „System.dll“, „System.core.dll“, popřípadě „mscorlib.dll“. [6] [9]

1.1.2.3 Common Type System

V rámci CTS jsou definovány instrukce a způsoby, jak jsou datové typy deklarovány, používány a řešeny virtuálním strojem. Zároveň se jedná o důležitou část, která má na starosti mezi-jazykovou integraci. *Common Type System* provádí následující funkce: [8]

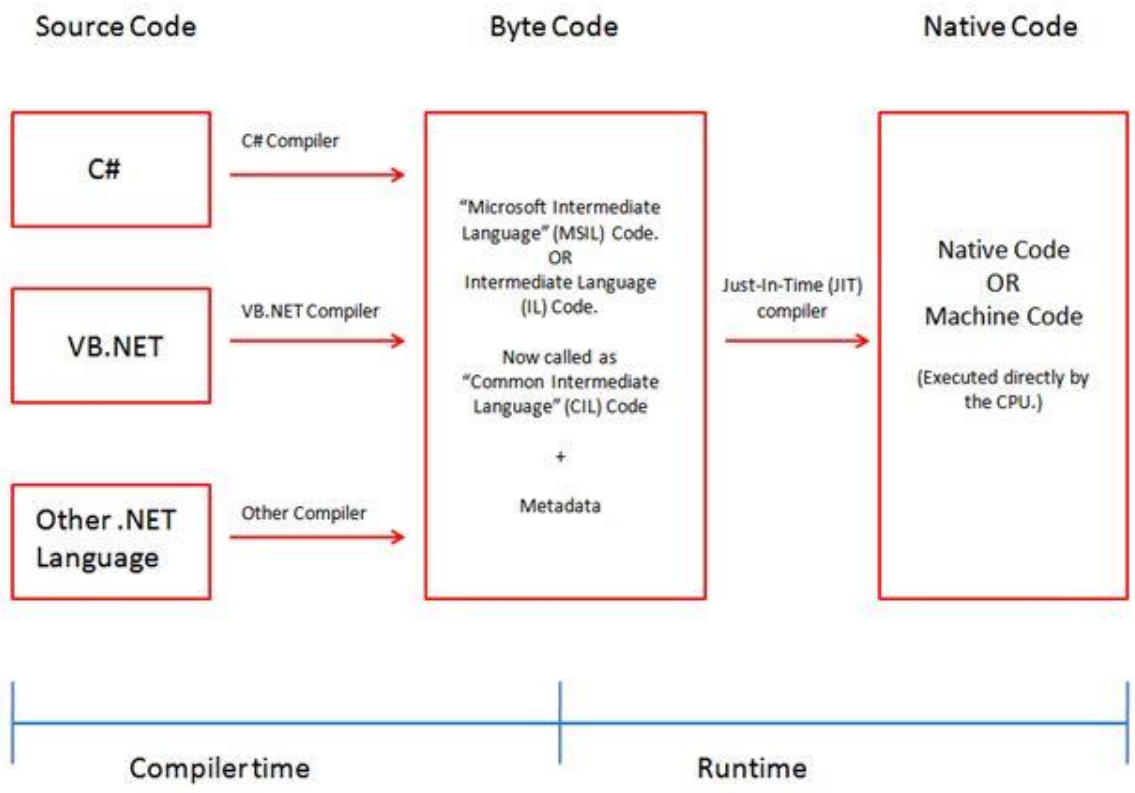
- Zajišťuje mezi-jazykovou integraci, typovou bezpečnost a vysokorychlostní provádění kódu.
- Poskytuje objektově orientovaný model, podporující kompletní implementaci mnoha programovacích jazyků.
- Definuje pravidla, která zajišťují bezproblémovou spolupráci mezi různými jazyky.
- Poskytuje knihovnu obsahující primitivní datové typy používané při vývoji (Boolean, Char, Int32 a UInt64).

1.1.2.4 Common Language Specification

Common Language Specification je podmnožina *Common Type System.* Jedná se o soubor pravidel a instrukcí, jež musí jednotlivé jazyky dodržovat, aby byly s platformou .NET kompatibilní. Zachování těchto omezení zaručuje interoperabilitu mezi všemi jazyky, které .NET framework podporuje. V rámci CLS jsou specifikovány způsoby, podle něž jsou zdrojové kódy jednotlivých programovacích jazyků převáděny do strojového mezikódu CIL. [7] [2]

1.1.3 Princip procesu (životní cyklus)

Jak už bylo zmíněno výše, framework .NET podporuje interoperabilitu mezi několika programovacími jazyky, jenž na .NET platformě pracují. To je možné díky samotnému návrhu frameworku, respektive mechanismu, jakým je zpracováván zdrojový kód. Na obrázku níže je graficky znázorněn životní cyklus kódu, neboli proces jeho zpracování a následné vykonání.



Obrázek 1 Proces kompilace zdrojového kódu
zdroj: http://www.c-sharpcorner.com/UploadFile/8911c4/code-execution-process/

Součástí .NET frameworku jsou kompilátory jednotlivých .NET jazyků. Může to být Visual Basic compiler, C# compiler, Visual C++ compiler, JScript compiler apod, nebo jeden z kompilátorů třetí strany jako jsou Eiffel, Pearl či COBOL compiler. Programovací jazyky .NET, stejně jako například jazyk Java, se řadí mezi jazyky s virtuálním strojem, což je aktuálně nejmodernější, nejrozšířenější a nejsofistikovanější druh programovacích jazyků. [11] Dříve, pomineme-li původní strojový kód, Assembler, popřípadě jazyky třetí generace, se používaly jazyky s kompilátorem a následně s interpretem, přičemž kompilátor překládal celý zdrojový kód do strojového najednou, zatímco interpret se staral o přenositelnost mezi

platformami a kód překládal po částech, které byly zrovna potřeba. Jazyky s virtuálním strojem jsou kombinací těchto dvou přístupů a přinášejí sebou řadu výhod:

- **Přenositelnost** – Kód je možné vyvíjet i provádět na různých platformách. Zdrojové kódy se převádí do mezikódu, proto je kód i jazykově nezávislý.
- **Málo zranitelný kód –** Aplikace jsou šířeny v mezikódu CIL – binární kód je hůře čitelný pro člověka.
- **Stabilita a odhalení chyb zdrojového kódu –** Díky kompilaci kódu do CIL mezikódu interpret detekuje chyby a proces zastaví před vykonáním potenciálně nebezpečné operace.
- **Poměrně vysoká rychlost**
- **Jednoduchý vývoj**

Samotný proces zpracování zdrojového kódu lze z širšího úhlu pohledu rozdělit do dvou fází – *Compilertime* a *Runtime*. Zdrojový kód se nejprve převádí z jakéhokoli .NET jazyka do mezikódu CIL pomocí adekvátního kompilátoru (C# kompilátor pro jazyk C#, *C++* kompilátor pro jazyk C++ apod.). Přeložený kód z různých jazyků má v CIL formě stejnou podobu, proto jsou všechny .NET jazyky platformě nezávislé a jsou schopny spolupracovat. Jazyk CIL, známý také pod označením *Common Intermediate Language*, je binární (strojový) kód podporující OOP (objektově orientované programování) a má výrazně jednodušší instrukční sadu, což ho dělá relativně rychleji interpretovatelným. Společně s CIL jsou vygenerována metadata, se nimiž je CIL uložen v *assembly*. Metadata slouží k ověření a zajištění integrity při vykonávání CIL kódu. Součástí virtuálního stroje CLR je také *Just In Time compiler*, označovaný také jako *JIT compiler* nebo *Jitter*. Během běhu aplikace (Runtime) JIT compiler používá generovaná metadata a stará se o následný převod binárního mezikódu CIL do nativní podoby, do strojového kódu, podle kterého následně procesor provede instrukce.

1.2 ASP.NET

V této kapitole je rozebrán ASP.NET framework, rodičovský framework ASP.NET MVC. Mimo charakteristiky zde čtenář najde informace o jeho struktuře a chronologickém vývoji.

1.2.1 Charakteristika

ASP.NET je open-source[2] webový Framework pracující na straně serveru, který je určený pro tvorbu dynamických[3] webových stránek. Byl vytvořen společností Microsoft a je nadstavba, respektive součástí mohutnějšího .NET frameworku. ASP.NET je v podstatě sadou knihoven a funkcionalit navržených tak, aby programátorům poskytly nástroje pro tvorbu dynamických webových stránek, webových aplikací nebo *web services*[4]. Součástí je sada hotových komponent a řešení pro základní problematiky, jako je práce s databází, správa formulářů, bezpečnost, autentizace uživatele, *key management, session management,* cashování apod. Na platformě ASP.NET lze vytvořit webové stránky a aplikace různého zaměření i rozsahu. Je určen pro design od malých osobních webů až po velké korporátní projekty.

1.2.2 Struktura ASP.NET

ASP.NET se rozšířil do více frameworků a technologií, mezi které patří například *Web Forms, MVC, Web Page, Web API* a *SignalR*. Ačkoli všechny zmíněné technologie byly vyvíjené separátně, s příchodem ASP.NET byly sjednoceny do společného celku a nyní jsou podmnožinou ASP.NET frameworku. Framework tedy umožňuje vývoj webových stránek či aplikací ve výše zmíněných technologiích.

---

[2] Open source – software, jehož zdrojový kód je dostupný veřejnosti zcela zdarma s možností volné použitelnosti, redistribuce a modifikace za dodržení určitých podmínek.

[3] Statický vs Dynamický web – **Statické webové stránky** jsou ukládány na server v konkrétní konečné podobě, ve které ji následně server při requestu vrací klientovi (uživatel + prohlížeč). Jsou nejčastěji ukládány v podobě HTML dokumentu a nenabízí uživateli žádnou další možnost interakce se serverem (např. ukládání nebo načítání záznamu z a do databáze). **Dynamické webové stránky** jsou generovány na straně serveru podle instrukcí server-side kódu jakožto odezva na akce uživatele a výsledné HTML je vráceno klientovi. Kód není pouze statický *markup*, je tak schopen reagovat na události ze strany uživatele a dynamicky modifikovat obsah. Klientovi je dodáno pouze výsledné HTML a JavaScriptový zdrojový kód. Lze tak vytvořit interaktivní prvky, například přihlašování uživatelů, guest booky, editace obsahu a jiné.

[4] Web services patří mezi pojmy, které se běžně do české terminologie nepřekládají. Na portálu mezinárodní organizace W3C jsou *web services* definovány jako softwarové systémy navržené pro podporu interoperability mezi zařízeními v rámci sítě.

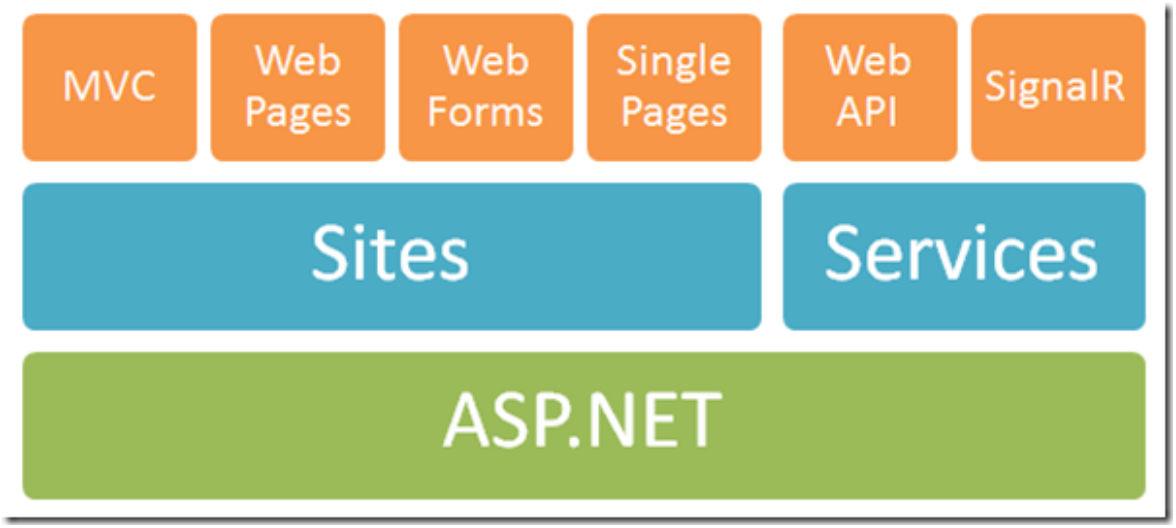
Základní členění dělí framework na dvě části – *ASP.NET Sites* a *ASP.NET Services*. V každé z těchto kategorií se nacházejí odlišné technologie, rozčleněné podle kritérií, zda technologie slouží pro vývoj *web services* nebo webových stránek.

ASP.NET Site

- **Web Forms, MVC, Web Pages, Single Page Application**

ASP.NET Services

- **Web Api –** Framework sloužící ke stavbě HTTP služeb (services), které jsou podporované širokou škálou klientů.
- **SignalR** – Knihovna, která zjednodušuje proces přidávání real-time funkcionalit do webových aplikací. Data jsou serveru předávána okamžitě, jakmile jsou dostupná, bez čekání na žádost o data ze strany klienta.



Obrázek 2 Struktura .NET Frameworku

Na Diagramu výše jsou znázorněné veškeré dílčí frameworky, ze kterých se ASP.NET skládá. [16]

1.2.3 Chronologický vývoj frameworku ASP.NET

První oficiální verze ASP.NET, **Framework ASP.NET 1.0**, vyšla v lednu 2002 jako součást frameworku .NET společně s vývojovým prostředím Visual Studio .NET. Framework byl vydán jako nástupce **ASP** (Active Server Pages), což byl první Microsoft

skriptovací engine na straně serveru pro dynamicky generované webové stránky. Přestože je označení ASP.NET frameworku odvozeno od původního ASP, obě technologie jsou znatelně odlišné. Obsažené komponenty, nástroje a celkově té doby nový přístup přinesl průlom ve vývoji webových aplikací. Velká oblíbenost byla zapříčiněna především díky kompatibilitě mezi jazyky .NET platformy, jež umožňovala flexibilně vyvíjet dynamické webové stránky v kterémkoli .NETem podporovaném jazyce. Verze 1.0 podporovala objektově orientovaný přístup, umožňovala využití dědičnosti, polymorfismu a dalších standardních vlastností OOP. Vývojářům bylo umožněno používat DLL knihovny a jiné serverové funkcionality k vytvoření robustnější dynamické aplikace, která zvládala pokročilé operace oproti pouhému renderování HTML. Příkladem může být řešení výjimek, validace apod. *ASP.NET Web Forms* zjednodušil přechod z vývoje desktopových Windows aplikací na vývoj aplikací webových díky možnosti vytvářet stránky složené z podobných ovládacích prvků, které jsou specifické pro uživatelské rozhraní desktopových aplikací. Zde byla velká podobnost technologií Windows Forms a Web Forms. ASP.NET 1.1 přidala ovládací prvky pro mobilní zařízení a automatickou validaci vstupu.

V listopadu 2005 Microsoft publikoval **ASP.NET 2.0**, jenž nesl pracovní název *Whidbey*. Aktualizovaný Framework přinesl dlouhou řadu rozšíření a novinek, mezi které patří především nové techniky deklarace přístupu k datům *SqlDataSource*, *ObjectDataSource*, *XmlFataSource* a nové datové ovládací prvky *GridView*, *FormView*, *DetailsView*. Společně s touto verzí frameworku přichází i nová verze IDE, *Microsoft Visual Studio 2005*. Další novinky:

- ovládací prvky pro navigaci,
- login ovládací prvky,
- master pages – umožňují vývoj webu pomocí šablon, kterých může mít aplikace více,
- skiny, témata a předpřipravené hotové části webu,
- personalizace, Provider class model, úplná pre-kompilace kódu.

ASP.NET 3.0 bylo vydáno 21.11.2006. a výrazným způsobem ulehčilo dosavadní způsob programování připojením *Windows Presentation Foundation, Windows Workflow Foundataion* a *Windows Communication Foundation*, které je možné použít k hostování aplikací na ASP.NET platformě.

Společně s vydáním verze **ASP.NET 3.5** publikoval Microsoft novou verzi vývojového prostředí *Visual Studio 2008*. Byla zlepšena podpora práce s daty a přidány datové ovládací prvky *ListView* a *DataPager*. Framework zahrnuje rozšíření ASP.NET AJAX[5] a umožňuje programátorům vytvářet AJAXové aplikace s podporou veškerých ostatních nástrojů ASP.NET.

V dubnu 2010 vyšla nová verze frameworku **ASP.NET 4** společně s .NET 4.0, ASP.NET MVC 4 a novou verzí vývojového prostředí *Visual Studio 2010*. Verze 4.0 byla obohacena o nové možnosti a pro vývojáře byla poměrně zlomová. Za největší novinky se označuje rozšiřitelnost cache, komprese stavu *session* nebo individuální správa View modelu. S touto verzí mohl programátor sám kontrolovat a konfigurovat routování. Pro mnohé vývojáře spočíval největší přínos v rozšíření ASP.NET Web Pages o nový View engine ***Razor***. Syntaxe *Razor* poskytuje rychlý, přístupný a odlehčený způsob tvorby obsahu dynamického webu kombinací serverového kódu a HTML.

Během následujících dvou let v Microsoftu pracovali na verzi frameworku **ASP.NET 4.5**, který oficiálně publikovali 15.8.2012. Stejně jako u starších verzí i tato přinesla další balík novinek včetně nové verze *Visual Studio 2012, ASP.NET MVC 4.5* a *ASP.NET API*. Kromě nového IDE a rozšíření frameworku přinesl silnější podporu AJAX technologie a podporu cloudových řešení. Další důležité novinky:

- unobtrusive JavaScript validace na straně klienta,
- bundling a minifikace klientských scriptů,
- podpora Web Socket protokolu.

Po verzi 4.5 následovaly **ASP.NET 4.5.1 a 4.5.2**, které byly vydány v intervalu necelého roku, v rozmezí října 2013 a května 2014. Přinesl nové metodiky řízení asynchronních dotazů na pozadí a opět aktualizované IDE. V červenci roku 2015 byla vydána nová verze frameworku – **ASP.NET 4.6**, s níž přibyla na trh i nová verze vývojového prostředí *Visual Studio 2015*. Další přínosy frameworku:

- Web API 2,

[5] AJAX – Na portálu mezinárodní organizace W3C je pojem AJAX definován jako technologie pro tvorbu rychlých a dynamických webových stránek. AJAX umožňuje update obsahu webových stránek asynchronně na pozadí výměnou menšího množství dat se serverem. Jinými slovy jsou části webové stránky aktualizovány bez nutnosti realizace opětovného načtení stránky. [17]

- MVC 5,
- SignalR,
- OWIN[6].

[6] OWIN – celým názvem *Open Web Interface for .NET* je standard pro rozhraní mezi .NET webovým serverem a webovou aplikací. Jedná se o open-source projekt ve vlastnictví komunity. Cílem tohoto standardu je oddělit server od aplikace a podpořit vývoj jednotlivých modulů pro .NET webový vývoj.

2 ASP.NET MVC

ASP.NET MVC je open-source framework vytvořený společností Microsoft. První oficiální verze byla vydána v roce 2009 pod licencí *Microsoft Public License* – MS-PL. Framework ASP.NET MVC nabízí alternativu k ASP.NET Web Forms, a slouží tak pro tvorbu webových aplikací. Díky svým nástrojům nabízí ideální prostředí pro tvorbu aplikací postavených na architektuře MVC, které jsou rozdělené do tří separátních komponent – Model, View, Controller. ASP.NET MVC je velmi lehký framework a jelikož se jedná o nadstavu k frameworku ASP.NET, umožňuje využití všech .NET a ASP.NET nástrojů. Stavba aplikace podle architektury MVC za použití ASP.NET MVC přináší řadu výhod: [38]

- jednoduší řízení komplexnosti aplikace rozdělením na tři různé vrstvy
- lepší podpora pro programování řízené testy (Test Driven Development) – nižší závislost mezi vrstvami přináší lepší testovatelnost
- vhodné pro velké týmy a vývoj vyžadující vysokou kontrolu nad chováním aplikace
- Využívá *Front Controller* vzor, při jehož implementaci se zpracovávají requesty skrz jediný kontroler. To umožňuje stavět aplikace se složitou a bohatou infrastrukturou routování.

2.1 MVC

MVC je architektonický vzor, podle kterého se aplikace separuje na tři základní komponenty – Model, View, Controller. Jeho hlavním cílem je oddělit prezentační vrstvy (View a Controller) od vrstvy aplikační (Model). MVC vzor se poprvé objevil v roce 1979, kdy byl integrován do knihoven programovacího jazyka Smalltalk. MVC je jedním z prvních přístupů popisujících implementaci separátních částí aplikace a jejich zodpovědnosti. Dříve se používal především pro vývoj desktopových aplikací, avšak v dnešní době je velmi populární v odvětví webového vývoje. MVC vzor se v čase vyvinul a je možné jej použit v odlišných variacích v závislosti na různých kontextech:

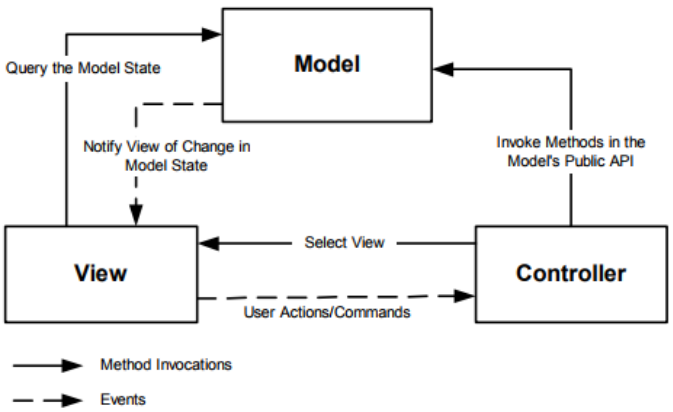
- **MVP – Model-View-Presenter,**
- **MVVM – Model-View-ViewModel,**
- **MVA – Model-View-Adapter,**
- **HMVC – Hierarchical Model-View-MVC.**

MVC architektura přináší výrazné benefity pro vývoj a v určitých směrech ulehčuje vývojářům práci. V projektu s takovou strukturou je možné vyvíjet jednotlivé vrstvy paralelně, aplikace je lépe testovatelná, dobře strukturovaná a jednotlivé komponenty jsou znovupoužitelné.

2.1.1 Životní cyklus

Koloběh všech operací v rámci životního cyklu aplikace iniciuje koncový uživatel svými akcemi – interakcí s uživatelským rozraním. [40]

- V rámci akce uživatele se odešle HTTP request ze strany klienta na server. Request nejprve zachytí router, který podle přijatých parametrů zavolá patřičný kontroler.
- Kontroler podle přijatých parametrů provede specifickou akci a předá instrukce modelu.
- Model podle instrukcí získá data z databáze, provede transformaci dat a aktualizuje model (změní stav). Notifikuje *View* vrstvu o změně stavu modelu a aktualizovaný model pošle zpět.
- *View* použije aktualizovaný model k naplnění šablony novými daty a následně vygeneruje patřičné HTML uživatelského rozhraní, které se zobrazí koncovému uživateli. V případě Single page aplikace se model nepředává do pohledů, ale ve specifickém datovém formátu (většinou JSON nebo XML) se posílá na front-end aplikace.
- UI čeká na další akci uživatele, aby se mohl cyklus opakovat.



Obrázek 3 Životní cyklus aplikace postavené na architektuře MVC [40]

2.2 Model

Model je nejnižší vrstva celé MVC architektury. Je tvořen doménovými objekty, obsahuje business logiku aplikace a reprezentuje data aplikace. Model uchovává data získaná z databáze podle instrukcí kontroleru a zodpovídá za jejich následnou manipulaci. Reaguje na requesty z View vrstvy a instrukce kontroleru, ačkoli na těchto vrstvách není přímo závislý. Model je reprezentovaný sadou doménových tříd, které mají specifické atributy a vlastní metody, jež pracují s daty.

```
public class Product
{
    public int ProductID { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public int DiscountPerc { get; set; }
    ...

    public decimal CalculateFinalPrice()
    {
        ...
    }
}
```

Kód 1 Ukázka třídy modelu v jazyce C#

Model přijímá parametry z venčí a podle jejich instrukcí data zpracuje a vydá ven – model neví, odkud byla data poslána a ani jak budou výsledná data ve výstupu vypadat. Zjednodušeně řečeno, model zapouzdřuje data a stěžejní funkcionalitu a stará se pouze o persistenci dat v databázi a vydávání dat ven. K takovým operacím se využívají ORM frameworky.[7] [35]

2.2.1 Entity Framework

Entity Framework, zkráceně také EF, je ORM framework z dílny společnosti Microsoft. Původně byl součástí .NET frameworku, ale od verze 6 je od .NET oddělen a opensourcován. Entity Framework je sada technologií, jenž rozšiřuje stávající ADO.NET[8] a slouží jako API pro komunikaci s databází – získávání, ukládání a manipulace dat v databázi. Framework mapuje databázové tabulky na instance (objekty) tříd a jejich atributy na jednotlivé sloupce tabulky. Vývojář při práci s EF v kódu používá

[7] ORM framework – Object Relational Mapping framework slouží k mapování, respektive k automatické konverzi dat mezi relační databází a objekty v rámci objektově orientovaného programování.
[8] ADO.NET – technologie nabízející nástroje pro přístup k datům. Je součástí .NET frameworku, konkrétně součástí base class library. Jedná se o set softwarových komponent, které je možné použít k přístupu a modifikaci dat uložených v relační databázi.

klasické objekty, zatím co framework na pozadí generuje SQL dotazy, které následně provede, a tak vývoj plně odstíní od ručního psaní SQL dotazů. Entity Framework patří mezi nejpoužívanější ORM frameworky na platformě .NET. Velmi populární je framework *NHibernate*, jež je také z dílny Microsoftu, nicméně v rámci diplomové práce je použit právě Entity Framework.

2.2.1.1 Přístupy

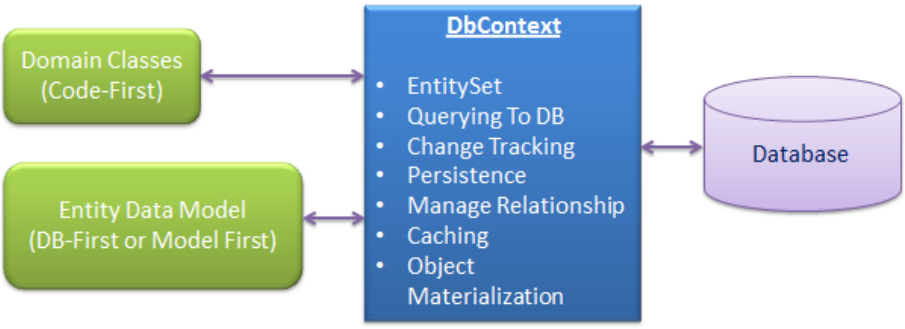
Entity Framework nabízí tři různé přístupy vytvoření propojení aplikace s databází. Jedná se o přístupy:

- **Code first –** EF vytvoří databázi podle doménových tříd. Tabulky v databázi odpovídají třídám modelu a jejich atributům, které se v konfiguraci na jednotlivé sloupce tabulky namapují. Vývojář má plnou kontrolu nad kódem aplikace a nemusí se starat o návrh a údržbu databáze – je vytvářena a aktualizována podle namapovaných modelových tříd automaticky. Každá změna v modelu se následně projeví i v databázi. Manuální změny v databázi budou naopak ztraceny, jelikož se databáze s aplikací drží v synchronním stavu a databáze je definována doménovými třídami.
- **Database first –** EF vygeneruje doménové třídy modelu podle navrženého databázového schématu. Při každé další změně ve schématu se změny projeví i v kódu tříd modelu aplikace. Vhodné je použití při velkých komplexních databázích navržených databázovými architekty. Všechny manuální změny v doménových třídách budou ztraceny.
- **Model first –** V rámci tohoto přístupu je nutné vytvořit diagram entit, jejich vztahů a dědičných vazeb pomocí editoru modelu – *Entity Data Model Wizard*. Výsledek je podobný class diagramu. Podle tohoto schématu se následně vygeneruje databáze a doménové třídy modelu. Výhodou tohoto přístupu je, že uživatel vše „nakliká“ a o zbytek se postará EF. Manuální změny v databázi budou opět ztraceny.

Při použití kteréhokoli přístupu udělá EF velkou část práce za vývojáře, což je jeden z důvodů, proč je tak oblíbený. Nicméně to s sebou nese i stinnou stránku. EF je poměrně snadné použít, ale vývojář často bez hlubší analýzy pořádně neví, jaké operace se uvnitř vlastně provádějí.

2.2.1.2 DbContext

Třída *DbContext* je důležitá část Entity Frameworku. Pomocí instance této třídy se přistupuje do databáze a slouží tak jako most mezi doménovými třídami a databází. Dále umožňuje práci s daty jako s objekty a je zodpovědný například za: persistenci dat, stopování změn, dotazování, caching, řízení vztahů mezi entitami apod. Instance *DbContext* třídy reprezentuje kombinaci vzorů *Repository* a *Unit Of Work* a umožňuje tak dotazování do databáze a seskupování všech změn, které jsou následně do databáze zapsány najednou. [36]



Obrázek 4 DbContext role [36]

2.2.1.3 Způsoby načítání dat

Entity framework nabízí tři různé způsoby načítání dat entit, které jsou svázány ve vztahu – např. produkt a kategorie:

- **eager loading,**
- **lazy loading,**
- **explicit loading.**

Eager loading

Eager loading je proces, kdy dotaz na specifický typ entity načte z databáze i data entit s tímto typem entit spojených. Je možné takovým způsobem načíst i více úrovní propojených entit, viz následující příklad. Eager loadingu je dosaženo pomocí použití klíčové metody *Include()*.

```
Product product = ctx.Products
    .Where(p => p.ProductID == 1)
    .Include(p => p.Category.Select(c => c.Subcategories))
```

Kód 2 Ukázka použití Eager loading

Lazy loading

Lazy loading je proces, při kterém jsou entity nebo kolekce související s dotazovanou entitou automaticky načteny z databáze až v době, kdy se přistoupí na navigační (vazební) atribut, přes kterou jsou propojeny. Ve zjednodušeném podání to znamená, že související data jsou načtena až v době, kdy jsou potřeba. Lazy loading se zajistí označením atributu třídy klíčovým slovem *virtual*.

```
public virtual Category ProductCategory { get; set; }
// nastavení navigační property v modelu dané třídy
...
// získání konkrétního produktu z databáze
Product product = from p in ctx.Products
                  where p.ProductID == 1
                  select p;

Category productCategory = product.ProductCategory;
// v tuto chvíli došlo k načtení dat kategorie produktu
```

Kód 3 Ukázka použití techniky Lazy loading

Při serializaci může lazy loading zapříčinit načtení většího objemu propojených dat, někdy dokonce všech dat databáze, proto je nutné jej pro tento případ zakázat. Toho je možné docílit odstraněním klíčového slova *virtual* z navigačních referenčních vlastností třídy nebo konfigurací samotné *DbContext* třídy, kde se lazy loading zakáže plošně pro všechny entity.

```
this.Configuration.LazyLoadingEnabled = false;
```

Explicit loading

Explicit loading je speciální metoda načítání dat z databáze, pomocí které je možné načítat data souvisejících entit lazy technikou i v případě, kdy je lazy loading zakázán. To je možné přímým voláním metody L*oad()* na již dotažených datech z databáze. Na rozdíl od obou předešlých technik, kdy se všechna data načetla jedním dotazem do databáze, se v tomto případě použijí dva separátní dotazy.

```
Product product = from p in ctx.Products
                  where p.ID == 1
                  select p;
context.Entry(product).Reference(p => p.Category).Load();
```

Kód 4 Ukázka použití techniky Explicit loading

2.3 View

View vrstva zodpovídá za poskytování uživatelského rozhraní uživateli. Poté, co kontroler přijme a zpracuje request prohlížeče, vrací při stavbě klasických webových aplikací ve většině případech pohled. View představuje HTML šablonu pro internetovou stránku, ve které je možné kombinovat HTML tagy se server-side zdrojovým kódem. To je možné díky specializovaným ***view enginům***, jež takový kód zpracovávají. View přijímá hotová data z kontroleru – nestará se o způsob zpracování, nezná vnitřní logiku aplikace, neví, odkud data pocházejí, jeho úkolem je pouze vložit model dat do pohledu a výsledné HTML odeslat na stranu klienta. K pohledu není možné z prohlížeče přistoupit přímo, ale pouze přes kontroler, který pohled renderuje.

2.3.1 View engine

View engine je zodpovědný za přeložení námi vytvořeného pohledu do HTML a jeho následné vykreslení v prohlížeči. Nabízí set nástrojů, pomocí kterého vývojář vytváří pohledy, jež jsou následně přeloženy a zobrazeny v prohlížeči. Platforma ASP.NET má zabudované dva výchozí view enginy. **ASPX** View Engine, známý také pod označením **Web Forms** View Engine a **Razor** View Engine, který byl vydaný později společně s ASP.NET MVC 3 frameworkem. Oba view enginy kombinují HTML se serverovým zdrojovým kódem, díky čemuž umožňují do jinak statického obsahu dynamicky vkládat různá data a logiku. Obecně se ale doporučuje takovou logiku uvnitř pohledů omezit na nezbytně nutné operace. Pro návrh pohledů na ASP.NET platformě je aktuálně nejvíce používán **Razor**, nicméně vývojář není striktně vázán na jeden z těchto enginů, existuje velké množství view enginů třetí strany – *Spark, Nhaml, NDjango, Hasic, Brail, Bellvue, Sharp Tiles, String Template, Wing Beats, SharpDOM* a další.

2.3.1.1 ASPX

ASPX View Engine je, stejně jako Razor, součástí ASP.NET frameworku. Syntaxe pro psaní pohledů je stejná jako pro ASP.NET Web Forms a stejně je tomu i s příponami souborů, které používá (*.aspx, .ascx, .master*). ASPX používá bloky "<% = %>" nebo "<% : %>" pro vykreslení server-side obsahu a v základu neposkytuje žádnou ochranu proti Cross-Site Scripting útokům. ASPX soubory mají vazbu na ASP.NET runtime, aby mohly být provedeny. Razor takové závislosti nemá, je avšak v porovnání s ASPX znatelně pomalejší.

2.3.1.2 Razor

Razor je pokročilý view engine, jenž byl představen společně s ASP.NET MVC 3 frameworkem. Nejedná se o nový jazyk, ale o modifikovanou markup syntaxe. Razor syntaxe je založena na programovacím jazyce C#, podporuje však i jazyk Visual Basic, se kterým lze vytvořit ten samý výstup. Razor používá *.cshtml* příponu a uvnitř pohledu používá „@“ místo „<% %>” k vytvoření skriptovacího bloku. Značnou výhodou je, že Razor šifruje HTML tagy a skripty před samotným renderem, což zamezuje Cross-Site Scripting útokům. V porovnání s ASPX přináší důmyslnější *intellisence*, nicméně je o poznání pomalejší.

2.3.2 Typovost pohledu

Na platformě ASP.NET MVC existují dva typy pohledů. Jedná se o pohled dynamický a silně typový. Typ pohledu je odvozen ze způsobu, jakým kontroler předává data do pohledu. Existují tři varianty předávání dat pohledu:

- použití ViewBag nebo ViewData objektu,
- použití dynamického typu,
- použití silně typového model objektu.

Při posílání dat do pohledu prvními dvěma způsoby se vytvoří dynamický pohled, zatímco poslední zmíněný zapříčiní vytvoření silně typového pohledu.

Pro **dynamické pohledy** je typické, že přijímají data v objektech, které nemají pevně definovaný typ. V rámci pohledu tak není dostupná funkční *intellisense*, jelikož vlastnosti objektů jsou čteny až za běhu aplikace. K vlastnostem je ale stále možné přistoupit pomocí *dot* operátoru, nicméně vývojář musí znát a zadat přesný tvar vlastností.

ViewBag je dynamický objekt, jenž nemá žádný typ a lze do něho vložit naprosto vše. Data se ukládají pod určitou vlastností a pod stejnou vlastností se k datům v pohledu přistupuje.

```
public ActionResult Index() {
    ViewBag.PageTitle = "Main Page";
    return View();
}
```

```
<h1>@ViewBag.PageTitle </h1>
```

Při použití ***dynamického typu*** se z kontroleru do pohledu opět předává dynamický objekt bez typu. Pro definování pohledu se používá direktiva *@model* s klíčovým slovem *dynamic*. K datům se v pohledu následně přistupuje přes klíčové slovo *Model*.

```
public ActionResult ProductList()
{
    var products = GetProducts();
    return View(products);
}
```

```
@model dynamic

@foreach(var product in Model)
{
<h2>@product.Name</h2>
<p>@product.Description</p>
}
```

Silně typový pohled ví, s jakým typem dat bude v rámci pohledu pracovat. Díky tomu je zpřístupněna plně funkční *intellisense*. Aby tomu tak bylo, je nutné na začátku pohledu specifikovat jaký typ objektu bude uvnitř používán.

Pro předání dat do silně typového pohledu byl použit stejný kód, a tedy stejná akce jako pro dynamický typ pohledu. Rozdíl je pouze ve specifikaci typu pohledu, viz první řádek kódu. Obecně se doporučuje používat silně typový pohled vždy, kdy je to možné, aby se zamezilo chybám při běhu aplikace – při změně vlastností na objektech se při dynamickém typu pohledu zjistí chyba až po kompilaci, za běhu aplikace, zatím co v případě silně typového pohledu se chybové hlášky objeví už v samotném vývojovém prostředí při psaní zdrojového kódu.

```
@model IEnumerable<Eshop.Model.Product>

@foreach(Eshop.Model.Product product in Model)
{
    <h2>product.Name</h2>
    <p>product.Description</p>
}
```

2.3.3 View Model & Data Transfer Object

V některých situacích pohled nepřebírá žádná data a skládá se pouze z HTML tagů. Výstup je v podstatě statická webová stránka. V opačném případě může nastat situace, kdy pohled pracuje s vlastnostmi více typů objektů nebo využívá pouze omezenou sadu atributů objektu. V tomto případě není žádoucí zasílat pohledu celý objekt. K takovým účelům slouží *View Model* nebo *Data Transfer Object*. V rámci použití obou typů objektu lze namapovat různé atributy z jednoho nebo více objektů do objektu jednoho, jenž se následně předá pohledu. Jejich způsob užití se ovšem liší.

Data Transfer Object umožňuje namapovat specifické atributy doménového objektu. Slouží ke zjednodušení provázanosti aplikace, je snadno serializovatelný a téměř nikdy neobsahuje žádné chování.

View Model se typicky skládá z dat jednoho či více doménových nebo DTO objektů. Může zahrnovat i další atributy, specifické pro chování pohledu nebo metody prováděné uvnitř pohledu.

2.3.4 Partial View

Partial View je znovupoužitelný pohled renderovaný uvnitř jiného pohledu. Slouží k omezení duplicitního kódu použitím jednoho nebo více *Partial View* v několika různých pohledech. *Partial View* je možné použít v hlavním souboru šablon, ale i v jednotlivých dílčích šablonách.

2.4 Controller

Controller, česky kontroler, je poslední komponenta MVC architektury. Úlohou kontroleru je reagovat na akce uživatele, respektive zpracovat klientské requesty. Každý request je pomocí routeru mapován na specifickou akci konkrétního kontroleru. Kontroler slouží jako komunikační most a mezi modelem a pohledem. Předává instrukce modelu, jimiž specifikuje, jaká data má načíst z databáze a jakým způsobem je zpracovat. Data aktualizovaného modelu vkládá do pohledu, který vrací prohlížeči jako odpověď na request. Ve zjednodušeném podání kontroler řídí celkový flow aplikace. Přebírá uživatelský vstup, zodpovídá za jeho správné zpracování a následné vydání adekvátního výstupu zpět na stranu klienta. [44]

2.4.1 Routování

Routování není záležitostí čistě MVC frameworku. Nástroje pro routování jsou integrovány přímo do ASP.NET, což znamená, že je mohou využít všechny frameworky, jež jsou frameworkem ASP.NET zastřešeny. Routování slouží k propojení specifické URI adresy s konkrétní akcí. HTTP request odeslaný klientem na server zachytí nejprve router a ten, v případě MVC, následně na příslušném kontroleru zavolá akci specifikovanou v přijatých parametrech - *http://localhost/home/contact* zavolá akci *Contact* na kontroleru *Home*.

Aktuálně ASP.NET nabízí dva různé způsoby routování. Starší *convention-based routing* a novější *attribute routing,* jenž přineslo vydání *Web API 2*. Od verze MVC 5 je routování pomocí atributů podporováno i v ASP.NET MVC. *Convention-based routing* je stále plně podporován a je možné oba přístupy kombinovat v rámci jednoho projektu.

Při použití ***Convention-based routing*** vývojář definuje jednu nebo více šablon, které jsou v podstatě parametrizované textové řetězce. Při obdržení requestu router nasměruje konkrétní kontroler na specifickou akci podle definovaných šablon. Šablony jsou definovány na jednom místě, což je výhoda. Na druhou stranu je těžké podporovat určité hierarchické URI, jež jsou typické pro RESTful API. Jedná se například o zanořené cesty, které odráží vztahy mezi objekty – např.: kniha a autor, nebo film a herec apod.

```
config.Routes.MapHttpRoute(
     name: "DefaultApi",
     routeTemplate: "api/{controller}/{id}",
     defaults: new { id = RouteParameter.Optional }
);
```

Kód 5 Ukázka convention-based routování

Attribute-based routing naopak přináší větší kontrolu nad URI a umožňuje tak lehce sestavit hierarchické routovací cesty. K aplikaci tohoto způsobu routování je nutné jej nejprve povolit zavoláním *MapHttpAttributeRoutes()* při konfiguraci Web API (pro API kontrolery), nebo zavoláním *MapMVCAttributeRoutes()* při konfiguraci routování pro klasické kontrolery. Specifická URI se pak jednoduše nastaví pomocí atributu *Route* nad konkrétní metodou. *Attribute-based routing* umožňuje na úrovni kontroleru nastavit prefix adresu pomocí atributu *RoutePrefix* pro všechny vnořené akce uvnitř kontroleru.

```
[RoutePrefix("api")]
    public class HomeController : ApiController
     {
       [Route("testRoute")]
       public TestDataObject GetTestJson(){
       …
       }

       Route("testRoute2")]
       public TestDataObject GetAnotherTestJson(){
       …
       }

      }
```

Kód 6 Ukázka routování pomocí atributů

Ukázky výsledných URI adres: *api/testRoute* a *api/testRoute2*

2.4.2 Web API

Web API slouží jako komunikační rozhraní pro předávání dat a poskytování HTTP služeb, respektive posílání HTTP requestů a odpovědí mezi back-endem a front-endem aplikace. V terminologii Web API se tato data často označují jako *resources*. Ke stavbě takového API slouží na platformě .NET framework ASP.NET Web API.

ASP.NET Web API je framework určený k sestavování HTTP services dostupných pro širokou škálu klientů různých platforem a zařízení, jako jsou desktopové prohlížeče, mobilní telefony i tablety. Framework je uzpůsobený tak, aby nabízel ideální prostředí pro tvorbu HTTP služeb postavených na architektuře **REST** a umožňoval tedy provádět CRUD operace nad databází. Je součástí ASP.NET platformy a je možné jej použít společně s jinými typy .NET webových technologií, zejména *MVC* nebo *WebForms*. *Web API* je *MVC* velmi podobný nejenom svou funkcionalitou, ale obsahuje také řadu stejných nástrojů: [31]

- *controller, action results, model binders, IOC container, dependency injection*

Jako Web API rozhraní jsou využívány kontrolery, jenž dědí z třídy *ApiController*. V tomto případě jsou kontrolery objekty vyřizující HTTP requesty a ve svých akcích místo HTML pohledu vrací serializovaná data v XML/JSON formátu. Ta dále posílají na specifickou URI adresu, odkud si je klient přebírá a naopak. Toho je možné dosáhnout několika způsoby:

- Při použití klasického MVC kontroleru, jenž dědí ze třídy C*ontroller*, jako je to pro tento typ kontroleru běžné, se nevrací *view*, ale data obalená JSON objektem, viz. následující příklad.

```
[HttpGet]
public ActionResult GetTestJSON()

{
    TestDataObject testObject = new TestDataObject
    {
        Name = "testObject",
        Created = DateTime.Now
    };

    return Json(testObject, JsonRequestBehavior.AllowGet);
 }
```

Kód 7 Ukázka klasického kontroleru vracející data v JSON formátu

- Za použití WEB API nástrojů. kontroler poděděný ze třídy *ApiController* vrací data ve XML formátu. Pro JavaScriptové aplikace se dodávají data ve formátu JSON, a proto je nutné upravit konfigurační nastavení pro WEB API.

```
config.Formatters.JsonFormatter.SupportedMediaTypes.Add(new
MediaTypeHeaderValue("text/html"));
```

Aktuálně by metoda vracející data v JSON formátu vypadala následovně:

```
public TestDataObject GetTestJson()
      {
          TestDataObject testObject = new TestDataObject
          {
              Value = "testValue",
              Created = DateTime.Now
          };
          return testObject;
        }
```

Kód 8 Ukázka API kontroleru vracejícího data v JSON formátu

2.4.2.1 Web API Operace

ASP.NET Web API umožňuje stavbu HTTP služeb postavených na architektuře REST. V rámci RESTful API je možné provádět nad daty v databázi (*resouces*) CRUD operace za pomocí volání předdefinovaných HTTP metod *GET, PUT, POST, DELETE,* kterými jsou v API reprezentovány. [32]

POST	GET	PUT	DELETE
Create	**Read**	**Update**	**Delete**
Vytvoří nový zdroj v kolekci	Vrátí jediný zdroj nebo všechny zdroje v kolekci	Update zdroje	Odstranění zdroje

Tabulka 1 Přehled základních HTTP metod pro CRUD operace

Konkrétní akci je možné namapovat na specifickou HTTP operaci dvěma způsoby.

- použití názvu HTTP operace jako prefix názvu akce,

```
public TestDataObject GetTestData(){}
```

- při definování akce použitím atributu – *HttpGet, HttpPost, HttpPut, HttpDelete*

```
[HttpGet]
public TestDataObject FindTestData(){}
```

Klasické RESTful API podporuje použití několika dalších metod, které nejsou v tabulce uvedeny. Jedná se o HTTP operace HEAD, PATCH a OPTIONS. Atribut *AcceptVerbs* umožňuje použití i těchto operací a zároveň slouží k volání více HTTP operací nad jednou akcí, viz následující příklad:

```
[AcceptVerbs(“GET”, “HEAD”)]
public TestDataObject FindTestData(){}
```

2.4.2.2 Konfigurace WEB API

Všechna nastavení WEB API se provádí pomocí metody *GlobalConfiguration.Confiure()* volané v metodě *Application_start*, jež při startu aplikace provádí veškerou konfiguraci aplikace. *Configure()* metoda přebírá v parametru funkci s jediným parametrem typu *HttpConfiguration*, která obsahuje konfigurační nastavení Web API. Příkladem může být následující ukázka konfigurace formátu, ve kterém jsou data předávána klientovi a povolení routování pomocí atributů.

```
protected void Application_Start()
{
   GlobalConfiguration.Configure(config =>
      config.Formatters.JsonFormatter.SupportedMediaTypes.Add(new
      MediaTypeHeaderValue("text/html"));
      config.MapHttpAttribudeRoutes();
   );
}
```

Ideálním způsobem konfigurace je vytvoření separátní statické metody uvnitř třídy *WebApiConfig.cs*. Uvnitř této metody specifikovat veškerá Web API konfigurační nastavení.

Konfigurační metodu dále předat jako parametr metodě *GlobalConfiguration.Confiure()*, která by se měla zavolat při startu aplikace – možné umístit do třídy Global.asax.

2.5 LINQ

LINQ, celým názvem Language Integrated Query, je dotazovací jazyk integrovaný do .NET frameworku. Byl publikován společně s jazyky Visual Basic 9 a C# 3.0 v rámci vydání .NET Frameworku 3.5. LINQ je deklarativní jazyk a svou syntaxí velmi podobný jazyku SQL. Na rozdíl od SQL LINQ nabízí unifikovaný způsob dotazování nad zdroji dat různého typu a umožňuje tak dotazovat na data například v poli, respektive v kolekcích objektů, XLM nebo v databázi stejným, jednotným způsobem. LINQ pracuje na bázi velké abstrakce, díky které se lehce snižuje výkon, avšak s příchodem novějších verzí běží takové operace ve více vláknech a snížení výkonu je poměrně zanedbatelné.

```
ICollection<Product> allProducts = new List<Product>
{
    new Product {Name = "Product1", Price = 10000 },
    new Product {Name = "Product2", Price = 20000 },
    new Product {Name = "Product3", Price = 5000 },
    new Product {Name = "Product4", Price = 9000 }
};
// dotaz na data pomocí SQL like syntaxe
var sortedProducts = from p in allProducts
                     where p.Price > 8000
                     orderby p.Price
                     select p;
// dotaz pomocí lambda výrazů a rozšiřujících funkcí
var sortedProductsShort = allProducts.Where(p => p.Price > 8000)
    .OrderBy(p => p.Price);
```

Kód 9 Ukázka dotazování dat pomocí LINQ technologie

Zdroj dat je možné dotazovat dvěma způsoby. Syntaxí imitující SQL nebo stručnější formou pomocí lambda výrazů a rozšiřujících funkcí. SQL syntaxe je pro mnohé intuitivnější a uchopitelnější, nicméně C# si kód na pozadí vždy přetvoří právě do podoby lambda výrazů a volá rozšiřující metody jako *Where(), Select(), OrderBy()* a další.

Oba dotazy v příkladovém obrázku výše vykonají ty samé operace a vrátí totožná data. Výpis obou polí vrátí následující produkty ve zmíněném pořadí: Proruct4, Proruct1 Proruct2.[37] [2]

2.6 NuGet Package Manager

NuGet Package Manager sice není výsadou ASP.NET či ASP.NET MVC frameworku, je ale důležitým nástrojem pro vývoj na .NET platformě. Jedná se o volně dostupný a nejvíce rozšířený open-source správce balíčků vytvořený společností Microsoft pro vývoj na .NET technologiích. Byl publikován v roce 2010, kdy byl distribuován jako rozšíření pro IDE Microsoft Visual Studio. S vydáním Visual Studia 2012 je NuGet zahrnut už v rámci instalace samotného vývojového prostředí. NuGet je možné obsluhovat pomocí wizardu v IDE (pravé kliknutí na projekt či knihovnu => výběr možnosti *Manage NuGet packages*) nebo pomocí příkazové řádky *Package Manager Console*. Package manager umožňuje vývojáři snadno vyhledat a nainstalovat potřebné balíčky knihoven či frameworků do projetu.

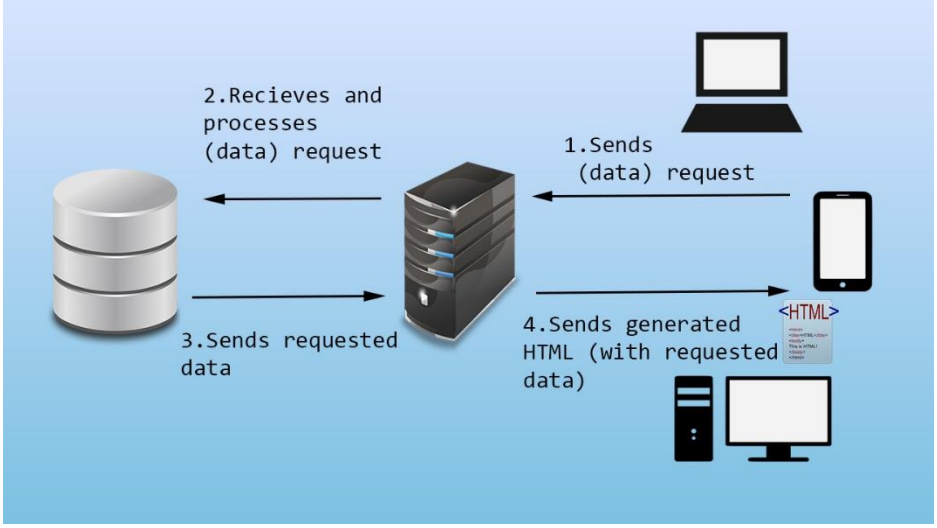
3 Single Page Application

Cílem této kapitoly je představit přístup vývoje webových aplikací formou Single Page Application a její architekturu. Je definován samotný pojem Single Page Application, nastíněn vývoj technologie a popsány přínosy a omezení, která SPA architektura přináší. Dále jsou uvedeny technologie, jež jsou pro vývoj SPA nezbytně nutné, nebo výrazným způsobem dokáží vývojáři ulehčit práci, a jsou proto nejvíce používány.

3.1 Představení a vývoj

Před příchodem Single Page Application architektury byly webové aplikace sestaveny pomocí statického HTML (generováno na straně serveru), stylovány pomocí CSS a JavaScript sloužil jako doplněk k interakci, stylování a k drobným animacím. Veškeré další interakce, především ty, které manipulovaly s daty, vyžadovaly kompletní cyklus oběhu dat (server round-trip). To znamená, že při každém requestu byla data od klienta odeslána na server, kde byla patřičně zpracována (popřípadě uložena či získána z/do databáze), zde se následně vygenerovalo HTML s požadovanými daty, nakonec se hotové HTML odeslalo klientovi a zobrazilo se v internetovém prohlížeči (viz Obrázek 5). I v případě pouhého přenačtení stránky, nebo requestu na stránku se statickým obsahem (např. pevně daným prezentačním textem a strukturou stránky – záložka *About*) bylo nutné tento cyklus zopakovat. Z toho vyplývá, že i sebemenší operace uživatele vyústila v request na server a vygenerování kompletně nového HTML, byť stejného jako při posledním requestu. Zatímco se request na serveru zpracovává, je klient zaneprázdněn a nemůže plně interagovat – uživatel musí jednoduše čekat na dokončení požadavku. Každý takový požadavek zatěžuje server, stránka se znovu celá načítá a čekání na nové vykreslení uživateli znepříjemňuje *user experience*[9] na webové aplikaci. [20]

[9] User experience, respektive uživatelská zkušenost, reprezentuje zážitek uživatele z použití aplikace.



Obrázek 5 Cyklus odeslání requestu od klienta na server typické pro klasické webové aplikace
zdroj: autor

Toto omezení je možné částečně eliminovat za použití technologie *AJAX*, která byla v době svého příchodu revoluční a byla základním kamenem pro evoluci a vznik nových technologií webového vývoje, mezi něž patří také *Single Page Application*.

Nasazení projektů se Single page architekturou by ještě před několika lety nebylo pro většinu projektů možné – nebo by bylo velmi nevýhodné, a to hned z několika důvodů. Při vývoji musí mít vývojáři na paměti, že výsledná aplikace musí korektně fungovat všem koncovým uživatelům, pro které je aplikace určena – ne pouze některým.

Aby aplikace, popřípadě ta část napsaná v JavaScriptu, fungovala, je zapotřebí naplnit několik předpokladů ze strany koncového uživatele, tedy na straně klienta. K vykonání instrukcí JavaScriptového kódu je zapotřebí JavaScript engine, dostatečně moderní webový prohlížeč, jenž JavaScript engine používá a povolený JavaScript v prohlížeči. Dnes to není problém, lze to pokládat za samozřejmost, neboť všechny moderní prohlížeč podporují JS a není potřeba ho vypínat. Nicméně před několika lety se na to vývojáři spolehnout nemohli, a proto nebylo možné vyvíjet aplikace postavené z velké části na JavaScriptu.

Jak bylo zmíněno výše, v architektuře *Single Page Application* je velká část zodpovědnosti a procesů přesunuta ze strany serveru na klientskou část, a tím pádem jsou i vyšší nároky na parametry koncových zařízení. Dříve nebyla mezi lidmi rozšířená

dostatečně výkonná zařízení, a to jak PC, tak především mobilní zařízení. Dnes již toto neplatí a většina uživatelů disponuje dostatečně výkonově silnými přístroji.

Jedním z dalších, podstatných důvodů byla nevyzrálost JavaScriptu a jeho slabá podpora v minulých letech. Před rokem 2008 nebyl JavaScript ani zdaleka tak silně podporovanou technologií. Jeho konkurenceschopnost vůči ostatním programovacím jazykům a technologiím byla velmi nízká a z důvodu nevyspělého prostředí pro JavaScript a nedostatku podpůrných knihoven a nástrojů nebylo možné aplikace na JS postavit. Používal se tedy zejména k méně rozsáhlým, většinou „dekorativním“, účelům. Během posledních devíti let nabral JS výrazně na popularitě. Vznikla spousta JavaScriptových frameworků, knihoven a různých nástrojů, kolem nichž intenzivně pracují rozsáhlé komunity vývojářů. Tato tendence vývoje a růstu popularity se v posledních letech stále více stupňuje a JS se obecně stal velmi podporovaným a protěžovaným jazykem, ve kterém je dnes možné napsat téměř vše, včetně skriptování na straně serveru. [18]

3.1.1 AJAX

Pojmem AJAX (celým názvem Asynchronous JavaScript and XML) se označuje set technik pro webový vývoj, které používají mnoho webových technologií na klientské straně, k vytvoření asynchronních webových aplikací.

Smyslem *AJAX*u je, zjednodušeně řečeno, zajistit aktualizaci webové stránky nebo její části – za použití dat zachycených ze serveru bez znovunačtení celé stránky. *AJAX* umožňuje webovým stránkám a webovým aplikacím odesílat a získávat data ze serveru asynchronně na pozadí, aby mohl dynamicky měnit určité segmenty stránky bez nutnosti znovunačtení celého obsahu. [19]

První zmínky o AJAXu jsou evidovány již v devadesátých letech. V roce 1996 jako první zabudovala společnost Microsoft do svého prohlížeče *Internet Explorer* možnost načítání a zachytávání obsahu asynchronně. V následujících letech tyto technologie začaly používat i vývojáři ostatních webových prohlížečů. Termín AJAX vytvořil a následně v roce 2005 publikoval James Garrett ve svém článku “*Ajax: A New Approach to Web Applications*”, ve kterém vymezil set technologií, jenž označení AJAX zastřešuje.

AJAX zahrnuje:

- standardizované prezentační nástroje – HTML/XHTML a CSS,
- dynamické zobrazení a interakce za použití *DOMu* (Document Object Model),
- výměnu a manipulaci dat pomocí XML a XSLT,
- asynchronní vyhledávání dat pomocí *XMLHttpRequest*,
- JavaScript, který vše svazuje dohromady.

Webové stránky či aplikace využívající AJAX technologii působí jako vysoce interaktivní, bohužel však stále není průběh srovnatelně plynulý s desktopovou aplikací, jelikož stále odesílá každý request na server, kde se následně zpracovává a výsledek se pošle zpět klientovi.

3.2 Charakteristika

Single Page Application, často také *One Page Application*, popřípadě *Single Page Interface* nebo zkráceně SPA, je označení pro webové aplikace, které běží v internetovém prohlížeči, skládají se pouze z jedné stránky a jsou z většiny napsány v JavaScriptu. To je zásadní rozdíl oproti klasickým webovým aplikacím, jež se mohou skládat z desítek nebo až stovek webových stránek. V dnešní době se postupně ustupuje od vývoje desktopových aplikací – jsou postupně nahrazovány aplikacemi webovými. Pro nativní, desktopové aplikace je typická vysoká interaktivita s téměř okamžitou odezvou a možnost práce i bez internetového připojení. Takové vlastnosti klasické webové aplikace nabídnout nemohou z hlediska jejich architektury a mechanismů, na nichž jsou postaveny. Proto přicházejí Single page aplikace, které se, mimo jiné, snaží tyto požadavky naplnit.

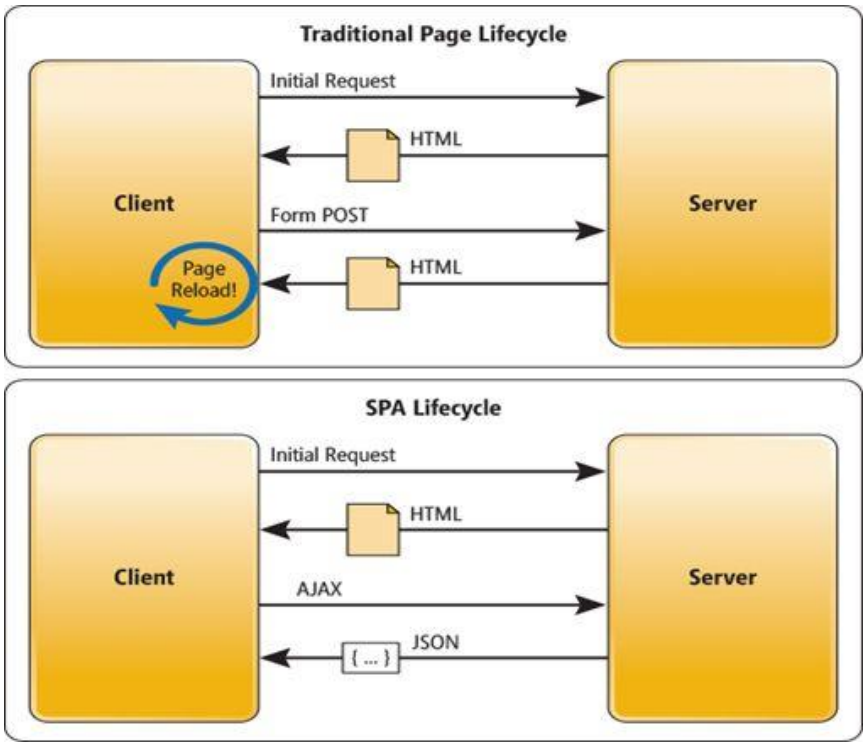
Gil Fink a Ido Flatow charakterizují *Single Page Application* jako*: „an SPA is a full web application that has only one page, which is used as a shell for all the other application web pages and uses JavaScript, HTML5, and CSS for all front-end interaction. In SPAs there are no full post back to the server, no full refreshes of a single web page, and no embedded objects. On the other hand, you use an HTML element in which its content is replaced from one view to the other by a front-end routing and templating mechanism.“* [20]

Aplikace postavené na architektuře SPA nabízejí uživatelům kvalitnější a mnohem plynulejší zážitek v porovnání s klasickými webovými aplikacemi. Jelikož se stránky znovu

nenačítají při každém kliknutí uživatele na nějaký referenční prvek, jsou vysoce interaktivní a působí velmi podobně nativním, desktopovým aplikacím, kde je minimalizována doba odezvy.

3.2.1 Princip a životní cyklus SPA

U architektury Single page je kladen důraz na klientskou část aplikace, kdy velká část operací a zodpovědností přechází ze serveru na klientskou stranu. Server lze napsat tak, že se většinou stará pouze o záležitosti jako je validace, autentizace a především persistence dat s databází. Poměrně často se ale server používá pouze jako zdroj a úložiště dat, jelikož se ostatní zmíněné operace mnohdy provádí už na klientovi. Klient se serverem komunikuje pomocí Web API postaveném ve většině případů na REST architektuře, přes které zasílá asynchronně requesty na server a ten následně zpracovává a vrací požadovaná data.



Obrázek 6 Porovnání životního cyklu tradiční a Single page aplikace [39]

Na obrázku výše je zobrazen životní cyklus stránky tradiční a Single page aplikace. Pro tradiční webové aplikace vrací server vždy nově vygenerované HTML a vyvolá znovunačtení celé stránky, ať už se jedná o inicializační render nebo další HTTP requesty

v rámci životního cyklu. Při prvním vykreslení Single page aplikace se odešle inicializační request na server a ten vrátí vygenerované HTML klientovi stejně jako u klasických aplikací. Následně si browser stáhne všechna potřebná data, zdrojový JavaScript a CSS, podle kterého je hlavní HTML dokument naplněn, stylován a následně zobrazen v prohlížeči uživatele. První render může trvat o něco déle než u klasických aplikací, protože si prohlížeč stahuje všechny potřebné zdrojové kódy a data. Nicméně poté už má vše potřebné uložené v paměti, což mu umožňuje reagovat pohotově. Při každé další akci uživatele už je postup zcela odlišný. Requesty jsou serveru posílány pomocí AJAX volání asynchronně, na pozadí, mezitím klient může vykonávat jiné operace. Po zpracování nevrací server znovu vygenerované nové HTML dané stránky, ale pouze data, většinou v JSON formátu. V prohlížeči, na straně klienta, se pak překreslí jen specifická část stránky, která vlivem akce uživatele změnila stav. Data a stav jsou ukládány v paměti prohlížeče, proto SPA umožňují uživateli fungovat částečně i v offline režimu, například při výpadku internetového připojení. Stav aplikace se synchronizuje se serverem a databází v určitých intervalech, popřípadě je proces synchronizace některými akcemi vynucen – záleží na konkrétní implementaci řešení. V případě offline režimu proběhne synchronizace až při připojení k internetu.

3.2.2 Základní stavební kameny SPA

- JavaScriptové frameworky, knihovny a další nástroje používané pro vývoj front-endu aplikace
- Routování – front-end routování je způsob navigace mezi pohledy v rámci SPA. Všechny pohledy jsou vytvořeny v rámci hlavního a jediného HTML souboru, nebylo by možné použít klasické navigační metodiky (pevně dané URL, backwards a forward tlačítko)
- HTML5 – HTML5 je použito k vytvoření hlavní stránky SPA
- Web API – s jehož pomocí klient komunikuje se serverem
- AJAX – veškerá interakce se serverem probíhá díky AJAXovým asynchronním voláním

3.3 Výhody a nevýhody

Single Page Application architektura nabízí zcela odlišný přístup k vývoji webových aplikací. Přináší řadu výhod, avšak nelze tento přístup označit obecně za nejlepší možné řešení všech webových aplikací, neboť s sebou nese i některá omezení, která mohou být pro určité druhy projektů překážkou.

3.3.1 Klady SPA

- **Rychlé načítání stránek –** po prvním renderu, kdy si prohlížeč stáhne většinu potřebných zdrojů (HTML, CSS, JavaScript – zdrojový kód klientské části aplikace, frameworky a knihovny), probíhá komunikace se serverem asynchronně, na pozadí. SPA nabízí vysoce rychlé a interaktivní prostředí, čímž zkvalitňuje uživateli jeho *user experience* při manipulaci s aplikací. Tento přístup mimo jiné šetří data uživatele, jelikož se na straně klienta stáhnou pouze jednou, ne při každém requestu znovu.
- **SPA kombinuje nejsilnější stránky webových a desktopových aplikací.**
    - rychlost, je multiplatformní a působí nativním dojmem, není potřeba instalace, offline režim, client-state management
- **Menší zátěž serveru** – díky tlustému klientovi je omezena zátěž serveru, spousta operací je prováděna přímo na klientovi, tudíž je omezen traffic.
- **Snadná distribuce a udržitelnost.**
- **Oddělení front-endu od back-endu** – back-end a front-end jsou prakticky dvě separátní části projektu, které je možné snadno vyvíjet a testovat zvlášť.
- **Zjednodušený vývoj pro mobilní aplikace –** je možné použít stejný back-end pro webovou aplikaci i nativní mobilní aplikaci.
- **Silná podpora** – SPA je poměrně nová technologie, jež jde rychle do popředí. V rámci SPA vzniklo několik silných JS frameworků s rozsáhlou komunitní základnou, která intenzivně vyvíjí novinky. Nové technologie a samotný JS se rychle vyvíjejí, proto lze očekávat, že zájem stále poroste a budoucnost webového vývoje se bude ubírat tímto směrem.
- **Offline –** SPA umožňuje efektivní využití *local storage*. Aplikace při inicializaci stáhne potřebná data, drží je lokálně, a tak může fungovat částečně i v offline režimu.
- **Snadné Debugování –** protože aplikace běží na klientovi, je možné zdrojový kód debugovat a analyzovat přímo v prohlížeči. Internetové prohlížeče nabízejí řadu

vestavěných nástrojů nebo možnost rozšíření pro vývojáře, pomocí kterých lze jednoduše sledovat traffic, operace na síti, debugovat, prozkoumat elementy stránky a sní spojená data.

3.3.2 Zápory SPA

- **Nefunkčnost bez JavaScriptu** – vzhledem k tomu, že SPA je v podstatě aplikace napsaná v JS, je nutné, aby na straně klienta byl vždy dostatečně moderní prohlížeč s povoleným JavaScriptem. Dnes je to téměř samozřejmostí, ale i tak se tento fakt nesmí opomenout.
- **Delší doba prvního načtení** – během inicializačního requestu prohlížeč stahuje všechny potřebné zdroje do paměti, podle kterých následně plní hlavní HTML stránku. Klient musí stahovat velké JS frameworky, proto obvykle trvá první vykreslení stránky déle.
- **Nefunkční historie v prohlížeči** – obsah SPA se skládá z jedné stránky, pro kterou JS pouze skrývá nebo odkrývá obsah, a tedy v základu při navigaci nemění URL adresu. Pevné vkládání URL adresy včetně navigace je pomocí *backwards* a *forward* tlačítka nefunkční. Je nutné použít další nástroje pro vytváření „falešných" URL adres jednotlivých pohledů a pro ukládání historie – pro *React* je to *react-router* nebo nově *react-router-dom.*
- **Menší bezpečnost** – v porovnání s tradičními webovými aplikacemi jsou SPA méně bezpečné. Jedním z příkladů může být *Cross-Site Scripting*, díky kterému může útočník vkládat škodlivé scripty na straně klienta.
- **SEO –** objevuje se problém s indexací stránek Single Page Aplikace a obtížné zpracování pro roboty vyhledávačů. Server posílá vygenerované HTML pouze při inicializačním renderu, dále jsou stránky sestavovány až na straně klienta pomocí JavaScriptu.

3.3.3 Shrnutí

Výše je uveden přehled přínosů a omezení při použití *Single Page Application* architektury. Určitě je možné oba seznamy rozšířit, ale toto jsou nejvýraznější a nejčastěji diskutované body. Přínosy SPA jsou evidentní, naopak u velké části nedostatků se dá diskutovat. Nutnost JavaScriptu je dnes už zanedbatelný bod, jelikož je přítomný na všech moderních zařízeních a není potřeba z žádných důvodů JS v prohlížeči zakazovat. Problém

v navigaci historií tu fyzicky už z principu fungování SPA je, ale lze lehce odstranit a řešení doimplementovat. Nedostatek špatného SEO už dnes není tak černobílý a dá se řešit. Z počátku se problém řešil pomocí užití „#!“ v URL a dnes je možné použít *Push State* v rámci HTML*5 history API,* a umožnit tak botům vyhledávačů prohledávat i JavaScript a CSS soubory. [33]

Z výše uvedeného textu je patrné, že technologie *Single Page Application* přináší řadu podstatných benefitů, rychle se vyvíjí a její klady převažují, avšak nutně nemusí vítězit nad zápory. Single page aplikace sice nabízí uživateli moderní a interaktivní prostředí a současně plynulý a příjemný zážitek z použití takové aplikace, to však nutně nemusí být vždy prioritní faktor. SPA je velmi sympatické, moderní řešení, přesto stále existují aplikace, kde může být vhodnější použít klasický přístup. Příkladem mohou být aplikace s vysokými nároky na náročné logické operace, na které je potřeba silný výpočetní výkon a strana klienta by nemusela takovou zátěž patřičně zvládat. Obecně se zatím považuje klasické řešení webových aplikací vhodnější pro rozsáhlé, komplexnější projekty s prioritou v bezpečnosti. Vždy je to ale o pečlivém zvážení, jaké parametry jsou pro konkrétní projekt stěžejní a o následné implementaci.

Nabízí se úvaha, proč nevyvíjet aplikace hybridně, kdy se použijí silné stránky tradičních *multi page* aplikací a SPA. Kombinace obou technologií se zřídka používá, je komplikovanější, náročnější na implementaci a většinou nepřináší požadované benefity. Hrozí naopak vysoká pravděpodobnost kombinace nedostatků z obou přístupů – pomalé načítání stránek, komplikovanější struktura a vývoj, front-end svázán s back-endem a následná horší testovatelnost a nemožnost znovupoužití stejného back-endu pro mobilní aplikaci.

3.4 Používané technologie

Pro vývoj SPA existuje řada nástrojů a technologií, bez kterých se developer neobejde, nebo naopak, které umožňují vytvářet aplikace efektivněji a pohodlněji. V této kapitole jsou popsány stěžejní a nejčastěji používané z nich.

3.4.1 JavaScript

Jedná se o multiplatformní, interpretovaný a objektově orientovaný skriptovací jazyk, který vyvinul Brendan Eichman ze společnosti Netscape. Byl publikován na přelomu

roku 1995 a 1996 a o rok později normován asociací ECMA podle standardu pro skriptovací jazyky **ECMAScript.** Společně s HTML a CSS tvoří JavaSciprt stěžejní technologie pro obsahovou produkci webových stránek. Majorita dnešních webových stránek používá JS a všechny moderní prohlížeče mají zabudovaný JavaScript engine, na kterém JS běží. V dnešní době ale už není pouze klientským skriptovacím jazykem, běží i v jiných prostředích, ne pouze v internetovém prohlížeči. Mezi taková běhová prostředí patří například *Apache CouchDB* nebo ***Node.js*** a právě ***Node.js*** nás ve spojitosti se Single page aplikacemi bude zajímat.

3.4.1.1 ECMAScript

ECMAScript, zkráceně ES, je skriptovací jazyk, jenž byl oficiálně představen v roce 1997, kdy byl standardizován organizací ECMA International podle specifikace ISO/IEC 16262. Dnes jsou používány především jeho konkrétní implementace, zejména JScript, ActionScript a jeho nejvíce známá forma JavaScript. ES dnes slouží jako standard nebo předpis a jeho konkrétní implementace jsou často označovány za dialekty. ECMAScript je zmíněn účelně, jelikož se s každou další publikací nové verze přidávají a mění nástroje v rámci JavaScriptu včetně syntaxe, což se projeví i v konkrétních JavaScriptových frameworcích. [21]

Nejznámější vydané verze:

Verze	ECMAScript3	ECMAScript5	ECMAScript6	ECMAScript7
Rok vydání	1999	2009	2015	2016
Jiné označení	ES3	ES5	ES6 ECMAScript2015	ES7 ECMAScript2016
Podpora	Všechny prohlížeče	Všechny moderní prohlížeče	Moderní prohlížeče - částečně	Moderní prohlížeče - slabě
Vybrané Novinky	Regulární výrazy, Try/catch	Strict mode, JSON podpora	Třídy a moduly	Exponenciální operátory (**), Array.proptype.includes

Tabulka 2 Přehled nejznámějších verzí ECMAScriptu [34]

Tabulka výše demonstruje, že ES6 a nejnovější ES7 jsou částečně nebo dokonce velmi omezeně podporovány moderními prohlížeči. I přesto je možné psát aplikace podle

nejnovějších JS specifikací za pomocí *transpileru*[10]. Jeden z nejvíce používaných je bezesporu ***Babel***.

3.4.2 Node.js

Node.js je asynchronní, multiplatformní a událostmi řízené JavaScriptové běhové prostředí postavené na V8 JavaScript enginu z prohlížeče Google Chrome. Publikován byl v roce 2009 a jeho autorem je Ryan Dahl. Jedná se o program napsaný v jazyce C++, jenž umožňuje běh JavaScriptu mimo internetový prohlížeč a umožňuje tak pomocí JS vyvíjet aplikace všeho druhu – od nástrojů příkazové řádky až po dynamické HTTP servery. Zaměřuje se především na efektivnost a maximální výkon, používá *event-driven* a *non-blocking I/O* model, který dělá Node.js velmi lehkým a svižným. [22] Node.js je volně dostupný open-source software, který si může každý stáhnout a nainstalovat z oficiálních webových stránek, případně se podílet na jeho vývoji. Součástí instalace je také **NPM** balíčkovací systém, který je nejpopulárnějším a největším registrem balíčků. [23]

3.4.3 Node Package Manager

Node Package Manager, známý také pod zkratkou NPM, je manažer balíčků JavaScriptového kódu pro běhové prostředí Node.js. Je napsaný čistě v JavaScriptu, jeho autorem je Isaac Z. Schlueter a první oficiální publikace se datuje k lednu 2010. Nabízí možnost distribuce vlastních modulů do veřejného nebo privátního repositáře poskytuje snadný přístup ke sdíleným balíčkům od jiných autorů.

Součástí NPM je online databáze všech publikovaných Node.js modulů, nazvaná *npm registry* a *npm* klient pro příkazovou řádku, který s *npm registry* komunikuje. Node Package Manager je nejpoužívanější nástroj pro správu balíčků a ve své online databázi jich aktuálně spravuje přibližně půl milionu. Ty jsou uloženy v CommonJS formátu a zahrnují soubor s metadaty ve formátu JSON. [24]

Node Package Manager stahuje balíčky, jež jsou specifikovány v *package.json*[11] konfiguračním souboru, v seznamu závislostí balíčků - *dependencies*. Stažené balíčky se organizují do složky *node_modules* v kořenovém adresáři projektu. Ještě do nedávna se

[10] Transpiler – je překladač, který transformuje (překládá) zdrojový kód z jednoho programovacího jazyka do jiného. V tomto případě převádí ES6/7 kód do nejvyšší verze ECMAScriptu, která je podporovaná všemi prohlížeči – aktuálně ES5.

[11] Package.json – soubor obsahující metadata aplikace nebo modulu. Důležitou částí jsou sekce *dependencies* a *devDependencies*, kde jsou specifikovány používané balíčky v projektu, které se při spuštění *npm install* doinstalují.

organizovaly do stromové struktury, což zvyšovalo konečnou velikost *node_modules* a zapřičiňovalo zpomalování procesu dotahování všech závislých balíčků, jelikož se mohly některé moduly stahovat vícekrát. V důsledku toho vznikl konkurenční ***YARN*** od Facebooku, jenž je mnohem rychlejší a balíčky organizuje do ploché, jednoúrovňové *flat* struktury a zamezuje tak duplicitám při stahování. *Yarn* všechny stažené balíčky taktéž ukládá do složky *node_modules* v kořenovém adresáři projektu a nově do globální cache v systémové složce uživatele. V případě požadavku na stažení již dříve staženého balíčku jej nestahuje znovu z online databáze, nýbrž z lokálního repozitáře. Tento přístup umožňuje dotahování balíčku v offline režimu a zároveň podstatně zvyšuje rychlost procesu. To přimělo vývojáře NPM k optimalizacím a v roce 2017 publikovali novou verzi manažeru, která taktéž využívá plochou strukturu a je rychlostně srovnatelná s *YARN*. [25]

Mimo *Yarn*, který je bezesporu největším konkurentem NPM, existují i další alternativní nástroje pro správu balíčků, jako například *ied, pnpm, bower, npmd.* Všechny zmíněné jsou kompatibilní s *npm registry*, zaměřují se především na rozšíření možností manipulace s balíčky a zvýšení výkonu a rychlosti.

3.4.4 Webpack

Při vývoji aplikací pomocí JavaScriptu je zvykem rozdělit zdrojový kód do dílčích modulů se vzájemnými závislostmi. Tomuto principu se říká *code splitting* a je to známá praxe už z jazyků C, C++, PHP, Ruby a dalších. Prohlížeče nativně *code splitting* nepodporují, proto se klientské straně dodává jeden JavaScriptový soubor, tzv. *bundle*. Takový soubor je možné vytvořit manuálně, což je velmi nepraktické a velikost souboru může poměrně narůstat. Ke snadné optimalizaci a vytvoření korektního *bundle* souboru existuje několik různých nástrojů. Za nejrozšířenější se považuje **Webpack**.

Webpack je open-source JavaScript bundler, který skládá všechny JS moduly do jednoho *bundle* souboru v závislosti na interních vazbách. Z toho vyplývá, že do výsledného *bundle* souboru jsou zahrnuty pouze ty soubory a části kódu, které jsou vzájemně provázány pomocí klíčových slov *import* a *export*. Podle dokumentace je webpack definován jako: „*webpack is a module bundler for modern JavaScript applications. When webpack processes your application, it recursively builds a dependency graph that includes every module your application needs, then packages all of those modules into a small number of bundles - often only one - to be loaded by the browser*.“ [26]

Webpack nabízí široké konfigurační možnosti a je možné ho rozšířit o řadu pluginů, které jej doplňují o další funkcionality. Konfigurační soubor nese název *webpack.config.json* a základem jsou čtyři nejdůležitější objekty:

- **Entry –** specifikace vstupního souboru aplikace – odtud webpack začíná při sestavování *bundle* souboru.
- **Output –** nastavení výstupu *bundling* procesu (umístění a jméno *bundle* souboru)
- **Loader –** transformuje soubory různých typů (xml, json, css, svg, jpeg apod.) do modulů a umožňuje, aby byly webpackem zpracovány a následně importovány a zahrnuty do grafu závislostí.
    - Provádí se na úrovni modulů.
- **Plugins –** nástroje, ze kterých se samotný webpack skládá. Používají se pro funkcionality, kterých není schopen loader – např. extrakce stylů, minifikace a uglifikace JavaScriptu.
    - Provádí se na úrovni celého *bundle* souboru.

Alternativním řešením může být použití *Browserify*, který v kombinaci s nástrojem *Gulp* poskytuje dostačující set nástrojů. Tato kombinace se často doporučuje pro menší projekty, je však na uvážení každého vývojáře, jaké nástroje vybere. Pro vývoj v knihovně *React* se doporučuje Webpack, jelikož je pro tuto knihovnu optimalizován.

3.4.5 Babel

Moderní prohlížeče prozatím nepodporují JavaScript ve formě ES6 a ES7, je tedy nutné takový kód přeložit do podoby, jíž standardně podporují všechny prohlížeče. Babel je JavaScriptový transpiler převádějící JS kód do čistého standardizovaného ES5 JavaScriptu, který běží ve všech moderních i starších prohlížečích. Umožňuje tak použití všech syntaktických novinek v rámci novějších ES specifikací. Babel také dokáže přeložit kód napsaný v JSX syntaxi, která se hojně využívá při vývoji v knihovně *React*.

4 React

React je jedním z JavaScriptových frameworků,[12] jenž slouží k tvorbě Single page aplikací. Formálně se nejedná o framework, ale o rozsáhlou open-source, komponentově orientovanou JavaScript knihovnu, která byla vytvořena pod hlavičkou Facebooku a slouží k vývoji uživatelských rozhraní. I přesto se ale velmi často v rámci terminologie používá označení framework. Jako open-source byl publikován v roce 2013, avšak Instagram, který byl napsán právě v Reactu, byl nasazen už v roce 2012. Autorem je Jordan Walke a inspiroval se XHP frameworkem, což je HTML komponentový framework pro PHP.

V konceptu MVC (Model-View-Controller) zastává aplikace napsaná v Reactu úlohu „V“, tedy *View*. Na rozdíl od frameworků, jako jsou například *Angular* nebo *EmberJS*, nepřináší reprezentaci modelu ani kontroleru, nebo jinou modifikaci MVC architektury. React přistupuje ke stavbě UI odlišným způsobem, nepoužívá šablony ani HTML směrnice, ale skládá ho dohromady z jednotlivých stavových a znovupoužitelných komponent. Při změně dat se díky práci s virtuálním DOMem nepřekresluje celá stránka, nýbrž pouze změněná část, která se následně zpropaguje do skutečného DOMu. Tento přístup přináší zvýšení rychlosti, jednoduchost a škálovatelnost. Základní stavební pilíře frameworku:

- **Komponenta,**
- **virtuální DOM,**
- **one-way data flow.**

Aktuálně je *Angular* v porovnání s Reactem stále používanějším frameworkem. To je celkem pochopitelné, jelikož je dostupný od roku 2010, zatím co React přišel až o tři roky později. Nicméně křivka růstu popularity Reactu je strmější – komunita roste velmi rychle a trendy naznačují, že zájem o React bude stále silnější. Důkazem může být následující seznam vybraných společností a projektů přecházejících na React: Whatsapp, Instagram, Facebook, Netflix, Airbnb, Twitter mobile, Flipkart, Housing, Imgur, Walmart[13]

[12] Mezi další JS framewroky pro tvorbu SPA patří AngularJS, EmberJS, Vue.js, Meteor.js, Knockout, Backbone.js

[13] Kompletní seznam je dostupný na adrese https://github.com/facebook/react/wiki/sites-using-react

4.1.1 JSX

React komponenty jsou ve většině případů psány v JSX syntaxi, což je JavaScriptová nadstavba, připomínající svou podobou XML. Kód napsaný v JSX využívá stromovou strukturu zanořených elementů po vzoru XML, umožňuje použití atributů a HTML tagů. Zápis v JSX podobě je oproti klasickému JS stručnější, srozumitelnější a umožňuje uvnitř HTML tagů použití JavaScriptových objektů. Psaní komponent v JSX je doporučenou praxí, není ale nutností a je na uvážení každého, jakou variantu zvolí. JSX není validní JS syntaxí a pro produkci je nutné kód transpilovat (přeložit) do prohlížeči podporovaného JS standardu (v tuto chvíli ES5). Pro tyto účely se využívá specializovaných nástrojů, mezi které patří například *Babel*.

4.1.2 Komponenta

Jak už bylo zmíněno výše, React framework slouží k vytvoření User Interface webové aplikace, které je složeno z nezávislých znovupoužitelných komponent, které dokáží držet stav. Díky svým vlastnostem lze nad komponentami přemýšlet jako nad izolovanými prvky, jež je možné znovupoužít nejen napříč celou aplikací, ale i v dalších projektech. Každá komponenta dědí z abstraktní třídy *React.Component*, má své vlastní vlastnosti, stav a sadu metod specifických pro konkrétní fáze životního cyklu. Pro každou komponentu je při nejmenším nutné definovat metodu *render()*. Ta přebírá vstupní data, která vloží do HTML a postará se o následné zobrazení v prohlížeči.

```
class ExampleComponent extends React.Component {
  render() {
    return <div>My name is {this.props.name}</div>;
  }
}
ReactDOM.render(<ExampleComponent name="Gydeon" />,
mountNode);
```

Kód 10 Ukázka React komponenty

4.1.2.1 Props & State

Hlavní úlohou React komponenty je převedení čistých dat do bohatého HTML. V React ekosystému existují dva druhy reprezentace dat modelu, které se přenáší mezi komponentami. Jedná se o *props* a *state*. Obě skupiny dat mají odlišnou úlohu a jejich znalost je pro vývoj stěžejní, nicméně nesou některé společné charakteristické znaky: [27]

- *Props* i *state* jsou prosté JavaScriptové objekty.
- Změna *props* nebo *state* spustí render update.
- *Props* i *state* jsou deterministické, ze stejných dat je sestaven vždy identický výsledek.

Props

Props je zkrácené označení pro properties. Jedná se o neměnná, *read only* data, která jsou fixní po dobu životního cyklu komponenty a jsou předávána z předka potomkovi jako parametr při vytváření instance komponenty. Na příkladovém obrázku výše je do *props* komponenty *ExampleComponent* předáván parametr *name* s hodnotou „Gydeon“. Každá komponenta může přijímat data skrz neomezené množství parametrů (*name, surname, age* apod.), ke kterým se v potomkovi přistoupí pomocí *this.props.parametrName*. Do *props* je možné předávat data různých datových typů – přes *string, number* až po objekty, nebo celé funkce.

State

State je druhý objekt nesoucí data aplikace, respektive data komponenty. Na rozdíl od *props* není *state* neměnný a během životního cyklu obvykle dochází k jeho modifikaci. Jeho počáteční hodnota je nastavena v metodě *constructor()* (viz kapitola 4.1.2.2 - *Životní cyklus komponenty*) a následně se v čase mění většinou v důsledku eventů vytvořených uživateli. *State* se mění pomocí funkce *setState* a k datům uloženým ve stavu komponenty se přistupuje přes klíčové spojení *this.state.parametrName*. Každý stav může mít neomezené množství parametrů, je přístupný pouze na samotné komponentě – lze jej tedy označit jako privátní. Potomek ke stavu rodiče nemůže přistoupit přímo, ale pouze pomocí *props,* pokud mu rodič hodnotu stavu do *props* předá. Stav rodiče je možné měnit pouze pomocí *callback* funkce, jíž rodič musí předat potomkovi v *props*. [28]

4.1.2.2 Životní cyklus komponenty

Na každou komponentu je vázáno několik metod životního cyklu, jež jsou standardně prázdné a je možné je uvnitř komponenty předefinovat. Porozumění životnímu cyklu je pro vývoj v Reactu stěžejní. Metody s prefixem *will* jsou volány před výskytem nějaké konkrétní situace, naopak metody s prefixem *did* jsou volány v případě, kdy zmiňovaná situace už nastala. Pro upřesnění je možné události životního cyklu rozdělit do tří, po sobě jdoucích kategorií – *Mounting, Updating, Unmounting*. [29]

Mounting

Metody této kategorie jsou volány ve chvíli vytváření komponenty a její integrace do DOMu.

- **Constructor()** – metoda volána při vytváření komponenty a před jejím vložením do DOMu – je tedy volána pouze jednou. *Constructor* slouží jako inicializační metoda, díky níž se nastavují výchozí hodnoty pro *props, state* a context pro vlastní metody. Před příchodem ES6 obstarávaly inicializaci metody *getDefaultProps()* a *getInitialState()*, které *constructor()* nahradila.
- **ComponentWillMount() –** metoda, která je volána také pouze jednou a to před provedením počáteční render funkce.
- **Render() –** *pure*[14] funkce, jež je jako jediná vyžadována. Vrací jeden React element, který může být nativní DOM element nebo vlastní React komponenta. Pokud takových komponent chceme vrátit více, je nutné je obalit <*div*></*div*> blokem. Při dosazení *null* nebo *false* nevyrenderuje komponenta nic.
- **ComponentDidMount() –** metoda volaná hned po inicializačním renderu. Inicializace, jež pracuje s DOMem, by měla být volána právě v tento moment životního cyklu. Pokud komponenta potřebuje data ze serveru, je toto vhodné místo pro zavolání requestu. Přenastavení stavu komponenty vyvolá znovu render.

Updating

Metody v této kategorii jsou vyvolány změnou *state* nebo *props* komponenty.

- **ComponentWillRecieveProps() –** metoda volaná před tím, než načtená komponenta obdrží nové *props*. Je vhodná pro zavolání *setState()* metody v případě, že se změnily *props*.
- **ShouldComponentUpdate() –** metoda volána před novým renderem, jenž je vyvolán změnou *props* nebo *state* komponenty. V této metodě se definuje, zda má změna stavu vyvolat nový render. Výchozí hodnota je nastavena na *true*, při změně na *false* nebudou zavolány zbylé metody z *Updating* kategorie.
- **ComponentWillUpdate() –** metoda zavolána okamžitě před renderem zapříčiněným změnou *props* nebo *state* za předpokladu, že metoda *ShouldComponentUpdate()* vrací hodnotu *true***.**

[14] *Pure function* – označení pro funkci, jež při stejném vstupu vrací vždy stejný výstup

- **Render().**
- **ComponentDidUpdate() –** metoda zavolaná hned po updatu komponenty pokud metoda *shouldComponentUpdate()* vrací hodnotu *true* a render proběhl. Je vhodná pro síťové requesty nebo pro operaci s DOMem.

Unmounting

Metody volané při odstraňování komponenty z DOMu.

- **ComponentWillUnmount() –** metoda se zavolá těsně před odebráním komponenty z DOMu a jejím následným zničením. Zde se volají nezbytné metody pro uklizení po komponentě, jako je zrušení síťových requestů, odstranění timerů, event listenerů nebo vyčištění dalších DOM elementů, které byly vytvořeny v metodě *componentDidMount()*.

4.1.3 Virtuální DOM

Moderní webové aplikace jsou vysoce interaktivní a jako odezvu na události vyvolané uživatelem dynamicky mění svůj obsah. Tyto akce vyžadují časté manipulace s Document Object Modelem aplikace – patří ale mezi nejpomalejší JavaScriptové operace. React se snaží počet a rozsah manipulací s DOMem omezit a vývojáře odstínit od přímého kontaktu se skutečným DOMem za použití DOMu virtuálního.

Vituální DOM je odlehčená abstrakce skutečného DOMu a lze ho chápat jako lokální zjednodušenou *in-memory* kopii. Má stejné vlastnosti a strukturu jako skutečný DOM, ale nedokáže zapisovat do prohlížeče, díky čemuž je práce s ním mnohem rychlejší. Princip jeho využití je velmi jednoduchý. React při updatu porovná původní stav VDOMu s aktuálním a následně ve skutečném DOMu provede update pouze na elementech, jež byly změnou skutečně ovlivněny. Například při editaci jedné položky seznamu se nepřekreslí celý seznam, či celá stránka, ale pouze jedna konkrétní položka, na níž je změna vázána. [27]

4.1.4 One-way data flow

Data jsou předávána jako atributy uvnitř tagu komponenty a putují směrem dolů, z rodičovské komponenty potomkovi. Taková dceřiná komponenta nemůže modifikovat předaná data přímo, ale pouze pomocí callback funkce, kterou může rodič potomkovi předat v *props*. React nepodporuje mechanismy, které by umožnily HTML přímo změnit komponentu. HTML může pouze vyvolat událost, na kterou komponenta reaguje. V případě nutnosti změní stav a zavolá render funkci, jež komponentu aktualizuje.

4.1.5 Redux

Redux je knihovna navržená pro správu stavu JavaScriptové webové aplikace. Původně byla vytvořená pro práci s React knihovnou, ale dnes se dá použít i ve spojení s jinými frameworky a knihovnami. Redux vychází z architektury *Flux*[15] a často se označuje jako její implementace, nicméně v určitých principech se zcela liší. Redux může být popsán skrze tři základní principy:

- **Jediný zdroj pravdy (single source of truth)** – stav celé aplikace je uložen v jediném objektu, který se nazývá *store*.
- **Stav je read-only** – jediná možnost, jak změnit stav aplikace, je provedení objektu akce, který popisuje, jakým způsobem se stav změní a aktuální stav uložit do nového stavového objektu pomocí.
- **Změny provádí *pure* funkce**.

Architektura Redux knihovny stojí na třech stavebních kamenech:

Store

Store slouží jako jediné úložiště dat celé aplikace, která reprezentují její současný stav. Při použití *Reduxu* jsou všechny komponenty vázány na *store*, nesledují pouze stav jiných komponent a v případě změny v globálním *state* objektu se patřičná komponenta přerenderuje. Stav je neměnný, při změně se vytváří vždy nový stavový objekt. To umožňuje ukládat a načítat aplikace v různých stavech, jako je tomu podobně

[15] Flux – architektonický návrhový vzor pro stavbu client-side webové aplikace. Aplikuje se při návrhu jednosměrného toku dat a slouží jako šablona pro strukturování projektu a pro správu stavu aplikace. [30]

u počítačových her. Proces funguje na principu *predictable state* – stejná data vrací aplikaci vždy ve stejném stavu.

Instance *store* se vytváří pomocí *Redux.createStore(reducer)* a poskytuje tři základní metody:

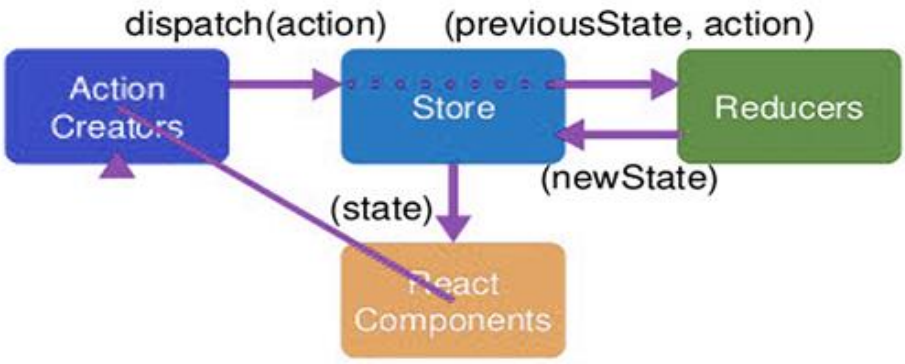
- **Store.getState** – vrací aktuální data.
- **Store.subscribe(callback)** – zavolá vloženou callback funkci při změně stavu.
- **Store.dispatch(action)** – zavolá akci, která provádí změnu ve *store*.

Action

Action je prostý JavaScriptový objekt, který specifikuje, jakým způsobem se mají data změnit a operaci pro změnu dat ve *store*. Jediný povinný parametr je *type*, jenž udává, jaká akce *reduceru* se má provést. Další, nepovinné parametry jsou naplněny daty, nad nimiž se má provést změna v rámci odkazované akce. Akce se vyvolá použitím *store.dispatch(ActionObject)*.

Reducer

Reducer je funkce, která přebírá v parametru objekt akce a vytváří nový stavový objekt, jenž naplní nově příchozími daty. Objekt akce specifikuje, jakou akci má reducer při modifikaci provést a předává finální data, kterými reducer naplní nový stavový objekt. Reducer (nebo sada reducerů) se vkládá jako parametr do *store* objektu při jeho vytváření a čeká na zavolání *store.dispatch(ActionObject)*, aby mohl zavolat akci, která provede změnu stavu.



Obrázek 7 Redux flow

5 Praktické použití popisovaných technologií

Tato kapitola a její dílčí podkapitoly se zabývají ukázkami použití výše popisovaných technologií v praxi. Jedná se tedy o představení vývoje klasických webových aplikací na platformě ASP.NET MVC a ukázky vývoje Single Page Application s back-endem taktéž na ASP.NET MVC, a front-endem postaveným na JavaScriptové knihovně React. Cílem této kapitoly je předvést použití zmiňovaných technologií při vývoji komplexnějšího celku, ne pouze na dílčích ukázkách. Jako příkladová úloha, na níž jsou předváděny a vysvětlovány zdrojové kódy, byl vybrán online katalog zboží na téma hudebnin. Nutno brát ale v potaz, že cílem diplomové práce není předat ucelené řešení aplikace, nýbrž demonstrovat samotný vývoj probíraných technologií – aplikace tak slouží pouze jako zdroj ukázkových kódů.

Katalog je základem pro budoucí e-shop a pokrývá tak veškeré scénáře kromě přidávání produktů do košíku a dokončení objednávky. Zboží je roztříděné do různých kategorií, podkategorií a značek a je možné je procházet, filtrovat a stránkovat. Aplikace nabízí také administrativní sekci, jež pokrývá scénáře pro CRUD operace se všemi základními entitami (produkt, kategorie, značka, uživatel a uživatelská role).

Tato kapitola poskytuje přehled, jak vyvíjet na platformě ASP.NET s MVC architekturou a jak projekt správně strukturovat a na co si dát při vývoji pozor. Ukazuje také, jak integrovat do projektu Entity Framework za použití přístupu *code first* a jak s tímto ORM frameworkem pracovat. Dále demonstruje propojení back-endu s front-end částí pomocí *REST API*, které se na platformě ASP.NET implementuje pomocí *ASP.NET WEB API* frameworku. Jsou pokryty základní CRUD operace pro stěžejní entity projektu a současně je probrána i autentizace za pomoci *ASP.NET Identity* frameworku. V rámci vývoje front-end části SPA jsou předvedeny tyto úkony: vývoj na frameworku *React*, správa stavu pomocí knihovny *Redux* a propojení back-endu s front-endem.

Předpoklady a přehled použitých technologií

- Obě verze aplikace (klasická i SPA) jsou postavené na **ASP.NET** 4.5.2.
- Nejsou použity žádné šablony, aplikace jsou holé *ASP.NET web application* projekty.

- Pro vývoj front-end části (View z architektury **MVC**) SPA verze je použita JavaScriptová knihovna *REACT* bez jakýchkoli předpřipravených šablon nebo startovacích projektů.
- **Entity Framework** – ORM framework z dílny Microsoftu. Jedná se o nadstavbu nad ADO.NET, poskytuje vývojáři automatizované mechanismy pro přístup a správu dat v databázi.
- **Automapper** – knihovna využívaná na straně back-endu k mapování objektů a jejich atributů na objekty jiného typu.
- **Autofac –** knihovna, reprezentující *IoC* kontejner. Spravuje závislosti (dependencies) mezi třídami – dependency injection – a umožňuje tak vkládat entity jedné třídy (respektive referenci na ní) do třídy jiné.
- **ASP.NET Identity –** framework od společnosti Microsoft určený pro správu uživatelských rolí, účtů a jejich autentizaci. Nahrazuje původní ASP.NET Membership systém. Zahrnuje podporu profilů, pracuje s *OWIN*, integruje *OAuth* a od verze Visual Studio 2013 je zahrnut v rámci předpřipravených šablon (například MVC šablona).
- **React** – JavaScriptová knihovna pro vývoj Single page aplikací, jež byla vytvořena společností Facebook.
- **Redux** – knihovna využívaná pro správu stavu aplikace na straně klienta
- **Webpack** – nástroj sloužící primárně ke kompilaci, bundlování a minifikaci modulů JS kódu. V rámci rozšiřujících pluginů, ze kterých je Webpack sestavován, a jichž je velké množství, nabízí nepřeberné možnosti dalších funkcionalit (např. import a načítání souborů různého typu, kompilace z různých jazyků, konfigurace node.js serveru apod.).
- **React-Router** – knihovna sloužící k navigaci v rámci Single page aplikace.
- **Material-UI** – knihovna se znovupoužitelnými React komponentami, které implementují Material Design od společnosti Google
- Pro vývoj front-endu jsou použity další, menší *npm* balíčky knihoven usnadňující vývojáři práci, nicméně výše zmíněné jsou pro projekt stěžejní.

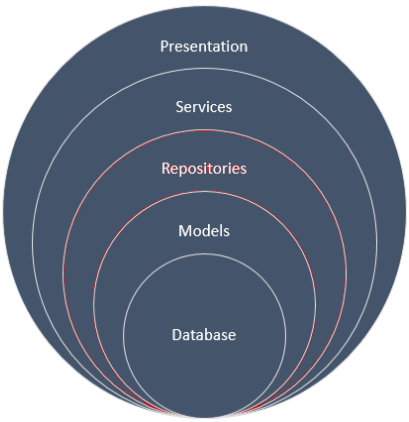
5.1 Struktura serverové části projektu

Před začátkem samotného vývoje je důležité vytvořit jasnou vizi architektury projektu. Běžně se o její specifikaci starají analytici nebo architekti řešení a podle ní se pak aplikace vytváří. Do jisté míry hraje struktura projektu klíčovou roli hned v několika aspektech, a to: znovupoužitelnost, testovatelnost, redundance kódu, přehlednost, rychlost, udržovatelnost, jediná zodpovědnost (princip single responsibility) a další. Každý z těchto aspektů může být v určité fázi rozhodující pro další vývoj projektu nebo jeho zastavení či úplné zrušení. Ovlivňuje časovou či finanční náročnost, má značný dopad na kvalitu řešení a určuje, zda je uskutečnění projektu vůbec reálné.

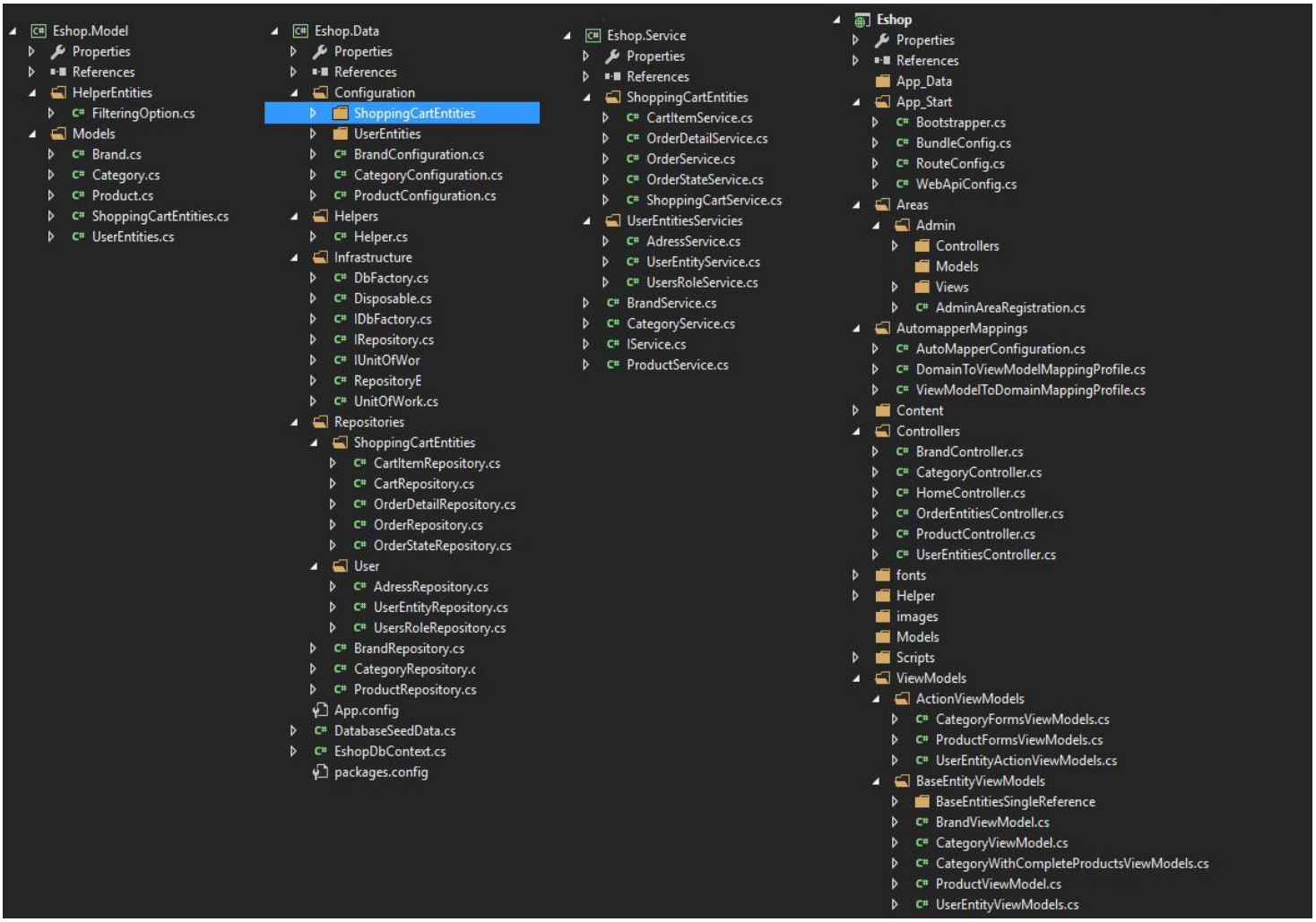
Pro malé aplikace může být rozdělení na *Model/View/Controller* dostačující, zároveň však může použití scaffolding akcí a předpřipravených šablon Visual Studia usnadnit nebo urychlit vývoj. Při stavbě středně velkých a větších projektů toto rozdělení stačit nebude, taktéž použití již připraveného řešení šablon může přinést limitace, jež bude nutné obejít a v konečném důsledku mohou přinést více překážek než užitku. Framework ASP.NET MVC sice umožňuje rozdělit projekt do více segmentů, do takzvaných *Areas*, kde každá reprezentuje určitou část funkcionality aplikace (např. podle rolí uživatele – administrace apod.), ovšem ani to není ve středně velkých a větších projektech dostačující.

Jelikož se projekt prezentovaný v rámci diplomové práce řadí mezi menší až středně velké projekty, rozhodl se autor rozdělit aplikaci na čtyři základní stavební bloky, respektive vrstvy rozdílných zodpovědností:

- **Model** – knihovna, ve které jsou pouze doménové třídy.
- **Data Layer** – knihovna s *repository* objekty, implementací a konfigurací Entity Frameworku. Zodpovídá za komunikaci s databází.
- **Service Layer** – přístupná v prezentační vrstvě, přes kterou kontrolery pracují s daty v databázi a ve které se nachází potřebná business logika.
- **Presentation Layer** – hlavní projekt, jenž obsahuje kontrolery, pohledy, view modely, konfiguraci projektu a dodatečný obsah (obrázky, css soubory, JS soubory apod.).



Obrázek 8 Logická struktura projektu
zdroj: autor



Obrázek 9 Celková struktura projektu

Autor se snažil postavit aplikaci tak, aby byla co největší část znovupoužitelná a mohla tedy být použita jak pro verzi klasické aplikace, tak pro Single page aplikaci a popřípadě i pro mobilní aplikaci. Na obrázku výše je zobrazena struktura, kterou mají obě aplikace společnou. V podstatě se liší pouze jejich prezentační vrstva. Typická webová aplikace postavená na platformě ASP.NET MVC obsahuje navíc adresáře s **pohledy**, které prohlížeči vrátí již hotové HTML generované na serveru. Oproti tomu SPA nezískává své pohledy vygenerované na serveru, ale obsah je naplněn až na straně klienta pomocí JavaScriptu. Jediné, co klientská část ze serveru potřebuje, jsou data, nejčastěji v JSON formátu, která se pak pomocí JavaScriptu vloží do HTML a zobrazí se uživateli v prohlížeči. V jiném případě může požadovat odpověď serveru (status code), zda se požadovaná akce zdařila apod. Zjednodušeně řečeno – klasická aplikace obsahuje navíc pohledy a její kotrolery vrací ve většině případů již hotové pohledy. V případě SPA se ze serveru vrací pouze data v JSON nebo *status code* odpovědi serveru, žádné *Views* projekt neobsahuje a v případě potřeby

obsahuje navíc konfiguraci pro WEB API. To je důvod, proč jsou pro obě verze aplikace separátně vysvětlovány pouze záležitosti, které nemají společné – *View* vrstva a kontrolery.

V aplikaci jsou použity *Repository pattern* a *Unit of Work pattern*, jež jsou napojeny na *Servicies* vrstvu, přes kterou se kontrolery a jiné pomocné metody dotazují na data. *Unit Of Work* zpracovává aktualizace uložené v paměti, které skládá dohromady a ty následně pošle do databáze – zde se vykonají všechny najednou. *Repository* objekty vykonávají CRUD operace nad daty v databázi. Všechny změny udržuje v paměti. Ty jsou provedeny až při zavolání akce *Unit of Work* vrstvy. Účelem je, aby se na databázi žádný subjekt nemohl dotazovat přímo, nevznikaly redundantní kódy a aby každá vrstva plnila jen specifickou úlohu.

V aplikaci se také řeší otázka závislostí mezi objekty. Častou chybou je vytváření nových objektů *servicies* pro každý kontroler nebo nových objektů konkrétních *repository* pro každou *service*. To není dobrá praxe, a proto je v projektu použit *Autofac* Inversion of Control kontejner, který řeší *dependency injection,* na pozadí vytváří instanci daného objektu a injektuje ho (předá referenci) na všechna místa v aplikaci, kde se používá.

5.2 Konfigurace serverové části projektu

Při konfiguraci projektu je nutné se zaměřit především na dva základní soubory, kde se konfigurace nejčastěji provádí:

- **Web.config**
- **Global.asax**

5.2.1 Web.config

Web.config je hlavní konfigurační soubor ve formátu XML, umožňující konfigurovat aplikace běžící na serveru. Jedná se o *Web.config* umístěný v kořenovém adresáři projektu, nikoli o *Web.config* v adresáři s pohledy[16]. Ve skutečnosti dovoluje konfiguraci všech aplikací na serveru, ale také vybraných aplikací, jediné aplikace nebo dokonce jednotlivých stránek či složek webové aplikace. Při vytvoření projektu je již velká část záležitostí nakonfigurovaná, avšak lze konfiguraci upravit a rozšířit. Uvnitř tohoto souboru je možné konfigurovat například debugger, kompiler, connection stringy, autentizaci, zobrazování chybných hlášení a jiné. Další HTTP konfigurační nastavení je možné vkládat v podobě

[16] Soubor Web.config ve složce Views primárně zamezuje přístup k pohledům jiným způsobem než přes kontroler. Není tak možné k jakémukoli pohledu přistoupit přímo přes URL adresu nebo jiným způsobem.

HTTP Modulů, které se ve *Web.config* registrují v elementu *<httpModules></httpModules>*. Pro účely diplomové práce je v této sekci věnována pozornost pouze konfiguraci connection stringu, neboť bez správného nastavení by se nepodařilo propojit aplikaci s databází. [41]

5.2.2 Global.asax

Jedná se o nepovinný ASP.NET aplikační soubor, který zachytá aplikační události pomocí *event handlerů*. Jelikož třída reprezentující *Global.asax* dědí také ze třídy *HttpApplication*, lze v jejím těle taktéž zachytávat HTTP události. V podstatě se nejedná o žádný konfigurační soubor, avšak je možné využít zachycené události mimo jiné i za účelem různých nastavení. Soubor *Global.asax* nemusí být v aplikaci obsažen a často je nahrazován jednotlivými *HttpModuly*. Ty stejně jako *Global.asax* dědí ze třídy *HttpApplication* a díky tomu mají taktéž přístup k *event handlerům* HTTP událostí. *Global.asax* a *HttpModuly* tak do jisté míry slouží ke stejným účelům, nicméně jsou mezi nimi jisté rozdíly. *Global.asax* není znovupoužitelný napříč více projekty, zatímco *HttpModuly* lze poskládat do znovupoužitelných knihoven. Ty na druhou stranu oproti *Global.asax* nemají přístup k aplikačním událostem jako je například *Application_Start* apod. *HttpModuly* jsou registrovány právě v konfiguračním souboru *Web.config*. Níže jsou uvedeny příklady často používaných aplikačních událostí: [42]

- **Application_Start**
- **Application_End**
- **Application_BeginRequest**
- **Application_AuthenticateRequest**
- **Application_Error**

V rámci projektu prezentovaném v diplomové práci se soubor *global.asax* používá ke konfiguraci routování, web API, inicializaci databáze, *Autofac* knihovny, *Automapper* knihovny, autentizaci a dalších. V následujících podkapitolách jsou některé konfigurace demonstrovány. Ve spojitosti s tím je důležitá především událost *Application_Start*, v níž se provádí většina nastavení.

```
public class Global : HttpApplication
{
    // Code that runs on application startup
    void Application_Start(object sender, EventArgs e)
    {
        AreaRegistration.RegisterAllAreas();
        GlobalConfiguration.Configure(WebApiConfig.Register);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);
        System.Data.Entity.Database.SetInitializer(new DatabaseSeedData());

        // run autofac and automapper configuration
        Bootstrapper.Run();
    }

    protected void Application_BeginRequest()
    {
         if(HttpContext.Current.Request.HttpMethod == "OPTIONS")
         {
            HttpContext.Current.Response.AddHeader("Access-Control-Allow-Origin",
                                                   "http://localhost:6060");
            HttpContext.Current.Response.AddHeader("Access-Control-Allow-Methods", "GET, POST, PUT,
                                                                                    DELETE");
            HttpContext.Current.Response.AddHeader("Access-Control-Allow-Headers", "Accepts, Content-
                                                   Type, Origin, X-My-Header");
            HttpContext.Current.Response.AddHeader("Access-Control-Max-Age", "60");
            HttpContext.Current.Response.End();
         }
    }
}
```

Kód 11 Obsah souboru Global.asax

Na ukázce zdrojového kódu výše je znázorněn obsah souboru *Global.asax*, ve kterém se provádí dodatečná konfigurace. Většina konfigurace se koná při zachycení události *Application_Start*, která se provede po startu aplikace a ve které se volají konfigurační metody z jednotlivých konfiguračních tříd.

5.2.3 Routování a Web API konfigurace

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        // enablin cross-origin request (request between different domains)
        var cors = new EnableCorsAttribute("http://localhost:6060", "*", "*");
        config.EnableCors(cors);

        // Web API configuration and services

        // Web API routes
        // this allows atribude routing
        config.MapHttpAttributeRoutes();
        // this is example of convention-based routing
        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );

        // returning data in JSON format from controler - default is XML format
        config.Formatters.JsonFormatter.SupportedMediaTypes.Add(new
                                                MediaTypeHeaderValue("text/html"));
    }
```

Kód 12 Konfigurace routování a Web API

Jelikož front-end Single page aplikace běží na jiném serveru než back-end, requesty tak putují mezi různými doménami. Takovým requestům se říká *cross-origin requests*, zkráceně *cors*, a ty nejsou ve výchozím nastavení na serveru povoleny, proto je nutné je pomocí *config.EnableCors(cors)* umožnit. Instance objektu *EnagleCorsAttribute* specifikuje domény, hlavičky a HTTP requesty, na něž má server povoleno odpovídat. Pro *cors* je důležité především přidat hlavičku *Access-Control-Allow-Origin*. V konfiguraci je dále povoleno routování pomocí atributů. Stále je ale možné používat starší *convention-based* způsob, jehož implementace je taktéž umístěna v tomto souboru – oba přístupy lze kombinovat. *Attribute routing*, neboli routování pomocí atributů, je ale mnohem intuitivnější a pohodlnější, nastavení se aplikuje v podobě anotací přímo nad metodami jednotlivých kontrolerů. Poslední část konfigurace specifikuje, že *ApiController* posílá data v JSON formátu místo výchozího XML.

5.2.4 Automapper konfigurace

Knihovna *Automapper* slouží k mapování objektu jednoho typu včetně jeho atributů na atributy objektu jiného typu. Hodí se například při vytváření DTO objektů nebo View modelů z doménových objektů a opačně. Výsledný objekt je možné skládat z více různých objektů. Závislosti mapování musí být specificky nastavené.

```
public static void Configure()
{
    Mapper.Initialize(x =>
    {
        x.AddProfile<DomainToViewModelMappingProfile>();
        x.AddProfile<ViewModelToDomainMappingProfile>();
    });
}

public class DomainToViewModelMappingProfile : Profile
{
    public DomainToViewModelMappingProfile()
    {
        CreateMap<Product, ProductViewModel>();
        CreateMap<Category, CategoryViewModel>();
        ...
    }
public class ViewModelToDomainMappingProfile : Profile
{
    public ViewModelToDomainMappingProfile()
    {
        CreateMap<ProductAddFormViewModel, Product>()
            .ForMember(p => p.Name, map => map.MapFrom(vm => vm.ProductName))
            .ForMember(p => p.Description, map => map.MapFrom(vm => vm.ProductDescription));
            ...
```

Kód 13 Konfigurace Automapper knihovny

Aby *Automapper* věděl, jakým způsobem má data a objekty mezi sebou mapovat, je nutné zaregistrovat profily mapování (třídy), v nichž je specifikováno, které objekty se mají mezi sebou mapovat. Pokud se názvy jednotlivých atributů obou objektů shodují, je

mapování jednoduché, viz profil *DomainToViewModel*. V opačném případě vyžaduje konfigurace bližší upřesnění, které vlastnosti se mezi sebou mapují, viz ukázka *ViewModelToDomain*. Metoda *Configure()*, jež celou konfiguraci provede je zavolána po startu aplikace ze souboru *Global.asax* v rámci volání metody *Run()*.

5.2.5 Autofac konfigurace

Autofac knihovna se za vývojáře stará o závislosti mezi objekty. V projektu je použita kvůli její funkcionalitě, kdy vytvoří instanci daného objektu na pozadí a reference na objekt distribuuje všude po aplikaci, kde se daný objekt používá. Díky tomu nemusí vývojář sám pracně předávat instanci objektu na různě zanořená místa aplikace nebo používat jiné nečisté způsoby. Často se tento problém obchází vytvářením nových instancí na místě, kde je objekt potřeba (např. v každém kontroleru se vytvoří nová instance *service* objektů) nebo použitím statických tříd či *Singletonu*,[17] to je ale špatná praxe. Knihovna potřebuje pouze vědět o jaké typy objektů se má postarat.

```
private static void setAutofacContainer()
{
    var builder = new ContainerBuilder();
    var config = GlobalConfiguration.Configuration;

    // ====== REGISTERING DEPENDENCIES =======
    // WebApi controllers
    builder.RegisterApiControllers(Assembly.GetExecutingAssembly());
    // Unit of work and DbFactory
    builder.RegisterType<UnitOfWork>().As<IUnitOfWork>().InstancePerRequest();
    builder.RegisterType<DbFactory>().As<IDbFactory>().InstancePerRequest();

    // register Repositories
    builder.RegisterAssemblyTypes(AppDomain.CurrentDomain.GetAssemblies())
        .Where(t => t.Name.EndsWith("Repository"))
        .AsImplementedInterfaces()
        .InstancePerRequest();
    // register Servicies
    builder.RegisterAssemblyTypes(AppDomain.CurrentDomain.GetAssemblies())
        .Where(t => t.Name.EndsWith("Service"))
        .AsImplementedInterfaces()
        .InstancePerRequest();
    ...
    // setting dependency resolver to be autofac
    var container = builder.Build();
    config.DependencyResolver = new AutofacWebApiDependencyResolver(container);
}
```

Kód 14 Autofac konfigurace

Z výše uvedené konfigurace je zřejmé, že *autofac* se stará o vytváření a předávání instancí pro jednotlivé *Controler*, *Repository*, *Servicie*, *UnitOfWork* objekty apod.

[17] Singleton – návrhový vzor v rámci kterého se vytváří pouze jediná instance objektu pro celou aplikaci, jenž je globálně přístupný.

5.3 Model

Jedná se o nejspodnější vrstvu celé aplikace, ve které jsou pouze definované doménové modely. To jsou třídy, reprezentující základní entity aplikace a ty odrážejí tabulky v databázi a vztahy mezi nimi. Na vrstvu *Model* se odkazují všechny tři ostatní vrstvy, jelikož s doménovými objekty pracují. Ke každé doménové třídě se vytváří typicky vlastní *controller*, *repository* i *service*. Všechny doménové třídy obsahují set atributů (vlastností), které odrážejí sloupce jednotlivých tabulek. Na tyto vlastnosti je možné aplikovat řadu anotací, které je různým způsobem nastavují – vlastnosti databázové tabulky a sloupců, chybné hlášky pro pohled, text pro popisek v pohledu a mnoho dalších. Zároveň je možné pomocí tzv. navigačních vlastností definovat vztahy mezi doménovými objekty.

Na obrázku výše je předvedena podoba doménových tříd *Product* a *Category*.

```
[Table("Products")]
public class Product
    {
        [Key]
        public int ProductID { get; set; }
        [Required(ErrorMessage = "Vyplňte název")]
        [StringLength(30, ErrorMessage = "Název je
                                  příliš dlouhý")]
        [Display(Name = "Název produktu")]
        public string Name { get; set; }
        [Required(ErrorMessage = "Vyplňte popis")]
        [StringLength(500, ErrorMessage = "Popis je
                                  příliš dlouhý")]
        [Display(Name = "Popis")]
        public string Description { get; set; }
        [Required(ErrorMessage = "Vyplňte cenu")]
        [Display(Name = "Cena")]
        [Range(0, double.MaxValue, ErrorMessage = "Cena
                                  nesmí být záporná")]
        public decimal Price { get; set; }
        public string Image { get; set; }
        public string StorageState { get; set; }
        public DateTime Created { get; set; }
        public DateTime Modified { get; set; }
        // navigation properties
        public int CategoryID { get; set; }
        public Category ProductCategory { get; set; }
        public int BrandID { get; set; }
        public Brand Brand { get; set; }
        public virtual List<CartItem> CartItemsOfProduct
                                    { get; set; }

        public Product()
        {
            Modified = DateTime.Now;
            Created = DateTime.Now;
        }
    }
```

```
    public class Category
    {
        public int CategoryID { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public string Image { get; set; }
        public DateTime Created { get; set; }
        public DateTime Modified { get; set; }
        public int ParentID { get; set; }
        public string Identifier { get; set; }

        public Category ParentCategory { get; set; }
        public virtual List<Category> chidlCategories
                                    { get; set; }

        public virtual List<Product>
                    ProductsInCategory { get; set; }

        public Category()
        {
            Modified = DateTime.Now;
            Created = DateTime.Now;
        }
    }
```

Kód 15 Ukázka doménových tříd modelu

Z ukázky je zřejmé, že tyto dvě entity jsou mezi sebou propojeny pomocí navigačních vlastností, jež říkají, že mezi entitami produktu a kategorie existuje vazba 1:N. Kdyby i produkt obsahoval *List<Category>* namísto jediné instance, jednalo by se o vazbu M:N.

Klíčové slovo *virtuál* říká, že se vázaná entita bude načítat pomocí *lazy loadingu*. V ukázce je předvedeno použití anotací, nastavujících vlastnosti jednotlivých atributů tříd – pro přehlednost a stručnost byla většina odstraněna a byl ponechán jen vzorek pro demonstrativní účely. Anotace typu *Required*, *Range* nebo *StringLength* určují pravidla pro validaci modelu a zároveň nastavení pravidel samotných sloupců dané tabulky v databázi – konvencí. *ErrorMessage* obsahuje text, který je možné zobrazit v prohlížeči při validaci formuláře v případě nedodržení pravidel. To samé platí i pro *Display*, který určuje, jaký název se má zobrazit v popisku nějakého *input* elementu. Textové zprávy nebo popisky je tedy možné nastavit pomocí anotací, nebo také až v samotném pohledu. Nicméně nastavení týkající se pohledů se primárně provádějí až ve view modelech případně DTO, protože právě ty se posílají do pohledů. Na této úrovni probíhá nastavení pravidel pro databázi a validaci. Dalším způsobem jak nastavit doménové třídy je pomocí *fluent API* v metodě *OnModelCreating()* na třídě databázového kontextu – viz kapitola 5.4.2.

5.4 Data Layer

Data Layer je vrstva, jež přistupuje přímo do databáze a je tak zodpovědná za práci s daty na nejnižší úrovni – vydávání, ukládání, editace a mazání přímo v databázi. Pro tyto účely se v projektu používá ORM framework Entity Framework, který propojuje doménové objekty s databází, respektive databázi podle nastavení doménových tříd vytváří (v případě použití přístupu *code first*) a drží ji v synchronním stavu. Hlavním obsahem této vrstvy je:

- **konfigurace** Entity Frameworku a počátečního stavu databáze,
- **repository objekty.**

5.4.1 Repository

Aby nedocházelo ke zbytečným redundancím kódu a metody pro práci s daty byly přístupné z jednoho místa aplikace, je v projektu použit *repository pattern*. Případné změny tak stačí udělat na jednom místě. Účelem *repository* objektů je manipulace s daty takovým způsobem, aby objekt, který operaci nad daty volá, nevěděl, jakým způsobem jsou data persistována a odkud přišla. Pro každou entitu doménového modelu je tak vytvořen *repository* objekt, obsahující veškeré operace nad daty v databázi nad danou doménovou třídou.

```
public abstract class RepositoryBase<T> : IRepository<T>
                                    where T : class
{
    private EshopDbContext dataContext;
    private readonly IDbSet<T> dbSet;
    protected IDbFactory DbFactory
    {
        get; private set;
    }
    protected EshopDbContext DbContext
    {
        get { return dataContext ??
            (dataContext = DbFactory.Init());}
    }

    protected RepositoryBase(IDbFactory dbFactory)
    {
        DbFactory = dbFactory;
        dbSet = DbContext.Set<T>();
    }
    // Methods
    public virtual void Add(T entity)
    {
        dbSet.Add(entity);
    }
    public virtual void Update(T entity)
    {
        dbSet.Attach(entity);
        dataContext.Entry(entity).State = EntityState.Modified;
    }
    public virtual void Delete(T entity)
    {
        dbSet.Remove(entity);
    }
    public virtual T GetById(int id)
    {
        return dbSet.Find(id);
    }
    public virtual IEnumerable<T> GetAll()
    {
        return dbSet.ToList();
    }
}
```

Kód 17 Ukázka generické abstraktní třídy RepositoryBase

```
public class ProductRepository :
       RepositoryBase<Product>, IProductRepository
{
    public ProductRepository(IDbFactory dbFactory)
        : base(dbFactory) { }

    public ICollection<Product>
           GetProductsByContainsName(string name)
    {
         return DbContext.Products.Where(p =>
         p.Name.Contains(name)).ToList();
    }

    public Product GetProductByExactName(string name)
    {
        return DbContext.Products.Where(p => p.Name ==
                                name).FirstOrDefault();
    }

    public override void Update(Product entity)
    {
        entity.Modified = DateTime.Now;
        base.Update(entity);
    }
```

Kód 16 Ukázka třídy ProductRepository

K dispozici je abstraktní generická třída *RepositoryBase*, která zahrnuje implementaci základních CRUD operací nad daty v databázi. Z této třídy ostatní *repository* třídy dědí, a není tak potřeba tyto operace znovu implementovat. Každá další *repository* implementuje pouze operace specifické pro danou entitu, popřípadě může poděděné metody přepsat pomocí klíčového slova *override*.

5.4.2 Entity Framework

Entity Framework plní funkce komunikačního mostu mezi aplikací a databází. Poskytuje API, přes které vývojář provádí operace nad daty v databázi, udržuje doménový model v synchronním stavu s databází. V projektu je použit přístup *code first*, což znamená, že podle doménového modelu a nastavených konvencí EF vytvoří databázi. Aby vše správě fungovalo, je nutné nakonfigurovat několik věcí:

- mapování jednotlivých doménových tříd na objekty typu *DbSet*, které reprezentují tabulku databáze,
- pravidla a omezení jednotlivých sloupců tabulky – nastavení konvencí,
- inicializaci databáze,
- connection string.

```
public class EshopDbContext : DbContext
{
    // SECTION 1
    public EshopDbContext() : base("EshopDbContext") { }
    // SECTION 2 - mapping domain model to database tables
    public DbSet<Product> Products { get; set; }
    public DbSet<Category> Categories { get; set; }
    ...

    public virtual void Commit()
    {
        base.SaveChanges();
    }
    // SECTION 3 - aditional database configuration
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Configurations.Add(new ProductConfiguration());
        modelBuilder.Configurations.Add(new CategoryConfiguration());
        ...
    }
}
```

Kód 18 Ukázka třídy databázového kontextu

Základem je vytvoření třídy databázového kontextu, přes jehož instanci se k databázi přistupuje. Na obrázku výše je ukázka třídy *EshopDbContext*, která dědí ze třídy *DbContext* a v konstruktoru je registrován pod specifickým identifikátorem. Tento název se používá k identifikaci kontextu v connection stringu, aby bylo jasné, ke kterému kontextu patří (aplikace může mít v horším případě kontextů i connection stringů více), viz následující ukázka.

```
<connectionStrings>
  <add name="EshopDbContext" connectionString="Data Source=DESKTOP-4OGONC3;
   Initial Catalog=EshopDatabase;User Id=sa;Password=***"
   providerName="System.Data.SqlClient" />
</connectionStrings>
```

Kód 19 Ukázka connection stringu aplikace

V connection stringu jsou obvykle specifikovány informace jako databázový kontext, jenž má být použit pro dané databázové spojení, název databáze, přístupové údaje, název počítače a poskytovatele databáze.

V části ukázky *SECTION 2* je předvedeno mapování doménových entit na *DbSet* objekty databázového kontextu, které reprezentují spojení s tabulkami v databázi. Pro přístup na data konkrétní tabulky se použije právě zaregistrovaný *DbSet* objekt na objektu

samotného databázového kontextu. Pro přidání nového produktu do tabulky s produkty by kód vypadal následovně:

```
context.Products.Add(new Product() {...});
```

V kapitole o vrstvě Modelu bylo uvedeno, že pomocí data anotací je možné specifikovat validační vlastnosti a databázové konvence. Tyto konvence je možné taktéž nastavit v samotné třídě kontextu pomocí *Fluent API*, konkrétně v metodě *onModelCreating()*. V rámci nastavení konvencí je možné změnit například primární nebo cizí klíč tabulky, název tabulky, maximální počet znaků textového řetězce, minimální a maximální hodnotu, povolení kaskádového mazání navázaných entit při odstranění apod. Je možné přistoupit přímo na danou entitu a na ní aplikovat nastavení nebo zaregistrovat konfigurační třídu jako je tomu na ukázce v *SECTION 3*.

```
public class ProductConfiguration : EntityTypeConfiguration<Product>
{
    public ProductConfiguration()
    {
        ToTable("Products");
        Property(p => p.Name).IsRequired().HasMaxLength(50);
        Property(p => p.Price).IsRequired().HasPrecision(8, 2);
        Property(p => p.Description).HasMaxLength(500);
        Property(p => p.CategoryID).IsRequired();
        Property(p => p.BrandID).IsRequired();
    }
}
```

Kód 20 Ukázka konfigurační třídy nad tabulkou

V poslední řadě je vhodné nastavit samotnou inicializaci databáze. Toto nastavení slouží prakticky ke dvěma účelům:

- k naplnění databáze daty při jejím vytváření,
- k nastavení pravidel, kdy má být databáze vytvořena a odstraněna.

Pro tyto účely byla v aplikaci vytvořena třída *DatabaseSeedData*, ve které zmíněné nastavení probíhá – viz následující obrázek. Aby nastavení proběhla, je nutné zavolat:

```
Database.SetInitializer(new DatabaseSeedData());
```

Tato metoda přebírá v parametru instanci třídy s inicializačním nastavením a je volána při

startu aplikace v metodě *Application_Start* (v souboru Global.asax) nebo v konstruktoru třídy databázového kontextu.

```
public class DatabaseSeedData : DropCreateDatabaseAlways<EshopDbContext>
{
    protected override void Seed(EshopDbContext context)
    {
        List<Category> categories = new List<Category>
        {
            new Category {Name = "Guitars", ParentID = 0, Identifier="guitars", ... },
            new Category {Name = "Bassguitars", ParentID = 0, Identifier="bassguitars", ... },
            ...
        }
        List<Product> products = new List<Product>
        {
            new Product {Name = "Ibanez RGT 42-DX", Price = 20000, ... },
            new Product {Name = "Jackson 7S", Price = 38000, ...},
            ...
        }
        ...
        categories.ForEach(x => context.Categories.Add(x));
        products.ForEach(x => context.Products.Add(x));
         context.SaveChanges();
    }
}
```

Kód 21 Ukázka inicializační třídy databáze

Z ukázky výše je patrné, jakým způsobem se databáze plní daty při inicializaci. V této třídě se taktéž specifikuje, v jakém případě se má databáze zahodit a vytvořit zcela nová. Entity Framework nabízí několik inicializačních strategií. V tomto případě se při každém opětovném startu aplikace vytvoří nová databáze a stará se smaže. Toho je docíleno poděděním z třídy *DropCreateDatabaseAlways*. Tento přístup je rychlý a je dobré jej používat při vývoji aplikace, a to ve fázi, kdy není potřeba udržovat nově uložená data v databázi. Entity framework dále nabízí možnosti *CreateDatabaseIfNotExists*, *DropCreateDatabaseIfModelChanges*. Názvy tříd vystihují přesně jejich účel a není tak potřeba bližší specifikace, avšak je nutné mít na paměti, že tyto strategie jsou platné pouze pro přístup *code first*.

5.5 Service Layer

Z pohledu architektury leží *service* vrstva mezi vrstvou prezentační a datovou, na níž je přímo napojená. Jejím úkolem je předat prezentační vrstvě již zpracovaná data ve finální podobě nebo naopak data ze vstupu uživatele zpracovat a předat datové vrstvě k uložení do databáze. V této vrstvě se nachází veškerá business logika, která se nad daty provádí. Mimo získávání a posílání dat do databáze skrz datovou vrstvu se jedná například o kalkulace, převody jednotek, stránkování, přepočet rozměrů obrázku, odesílání e-mailů, nastavení a získávání cookies, mapování objektů a další. V rámci servisní vrstvy jsou vytvořeny servisní třídy pro každou doménovou entitu (např. *ProductService*,

CategoryService atd.). Ty obstarávají business operace specifické pro danou doménovou třídu. Tyto servisní třídy jsou někdy označovány jako tzv. manažery – *managers*.

```
public class ProductService : IProductService
{
    private readonly IProductRepository productRepository;
    private readonly ICategoryRepository categoryRepository;
    private readonly IUnitOfWork unitOfWork;

    public ProductService(IProductRepository productRepository, ICategoryRepository
                          categoryRepository, IUnitOfWork unitOfWork)
    {
        this.productRepository = productRepository;
        this.categoryRepository = categoryRepository;
        this.unitOfWork = unitOfWork;
    }
    public IEnumerable<Product> GetAll(string[] includeMembers)
    {
        return productRepository.GetAll(includeMembers);
    }
    public Product GetById(int id)
    {
        return productRepository.GetById(id);
    }
    public void Create(Product entity)
    {
        productRepository.Add(entity);
    }
     ...
```

Kód 22 Ukázka ProductService třídy

V příkladovém kódu výše je vyobrazena část servisní třídy pro produkt. Ukázka je pro demonstrativní účely zjednodušena. Jednotlivé *repository* instance jsou injektovány do konstruktoru knihovnou Autofac a není tak potřeba vytvářet nové instance nebo ručně řešit závislosti mezi objekty jiným způsobem. Veškeré operace na databázi volá servisní třída na instanci objektu *repository*.

```
// generic class takes arguments as type arguments type of data source and type data should map to
public static class FilterDataHelper<T, V> where T : class where V : class
{

    public static FilteredDataResultViewModel<V> GetPagedFilteredDataResult(IEnumerable<T>
                                                    allFilteredItems, FilteringOption filterOption)
    {
        FilteredDataResultViewModel<V> filteredDataResult = new FilteredDataResultViewModel<V>();
        filteredDataResult.FilteredTotalCount = allFilteredItems.Count();
        IEnumerable<T> filteredItems = GetCurrnetPageData(allFilteredItems,
                                                filterOption.RecordsOnPage, filterOption.PageIndex);
        filteredDataResult.FilteredItems = Mapper.Map<IEnumerable<T>, IEnumerable<V>>(filteredItems);
        filteredDataResult.FilteredCount = filteredItems.Count();
        filteredDataResult.PagesCount = (int)Math.Ceiling(((double)allFilteredItems.Count() /
                                                            (double)filterOption.RecordsOnPage));
        filteredDataResult.DataResouceType = filterOption.DataResouceType;
        return filteredDataResult;
    }

    private static IEnumerable<T> GetCurrnetPageData(IEnumerable<T> allFilteredItems, int
                                                    countOnPage, int pageIndex)
    {
        int totalCount = allFilteredItems.Count();
        int skipCount = (pageIndex - 1) * countOnPage;
        int itemsCountLeft = totalCount - (countOnPage * (pageIndex - 1));
        int takeCount = itemsCountLeft >= countOnPage ? countOnPage : itemsCountLeft;
        return allFilteredItems.Skip(skipCount).Take(takeCount).ToList();
    }
}
```

Kód 23 Ukázka generického stránkování

Na ukázce výše je vyobrazena generická statická[18] pomocná *helper* třída, která se stará o stránkování dat jakéhokoli typu. Její metoda bere v parametru seznam dat ke stránkování a nastavení stránkování, podle kterého vybírá správnou stránku a počet záznamů k zobrazení. Následně vybrané položky namapuje z původního typu na požadovaný typ (v případě této aplikace se jedná o mapování z doménové entity na jeho ekvivalentní view model).

5.6 Presentaion Layer

Prezentační vrstva je ta část aplikace, která obsahuje komponenty, jež implementují a zobrazují uživatelské rozhraní. Prezentační vrstva je přímo napojená na servisní vrstvu. Jedná se o hlavní projekt aplikace, kde se nachází konfigurace projektu, CSS soubory a frameworky, JS soubory a frameworky, obrázky, videa a v podstatě celá MVC část – kontrolery, pohledy a model. Model na této vrstvě není reprezentován doménovými entitami, ale view modely nebo DTO objekty, které obsahují pouze data specifická pro konkrétní pohledy. V případě Single Page Aplikace pohledy nejsou potřeba, jelikož *View* vrstva je reprezentována externím JavaScriptovým projektem, jenž vizuální reprezentaci vytváří až na straně klienta, kdy JS plní HTML obsah do jediné stránky aplikace.

Díky tomu se pro oba druhy aplikace mírně liší i kontrolery. Kontrolery klasické webové aplikace dědí ze třídy *Controller* zatímco v případě SPA je rodičovská třída *ApiController*. Interní chování uvnitř kontroleru se nemění. Jsou napojeny na stejnou servisní vrstvu, která jim obstarává hotová data, jež mají v ideálním případě pouze použít. Platí tak, že kontrolery by měly být hubené a obsahovat co nejméně logiky či žádnou logiku. V tomto projektu veškerá data a operace nad nimi obstarává servisní vrstva a v kontroleru se pouze mapují na konkrétní view model, jenž je předán pohledu a validuje model. Rozdíl je až v návratových typech metod kontrolerů. Metody kontrolerů klasické webové aplikace vrací ve většině případů pohled naplněný daty modelu, nebo přesměrují na

[18] Statická třída nemůže být instanciována a neumožňuje tak použít klíčové slovo *new*. Je globálně viditelná a k metodě na dané statické třídě se přistupuje přes název třídy. K použití metody pro stránkování produktů by kód vypadal následovně:

```
FilterDataHelper<Product, ProductViewModel>.GetPagedFilteredDataResult(...,...);
```

jinou akci. V případě Api kontrolerů se nevrací pohled, ale pouze se posílají data v JSON/XML formátu přes REST API na front-end aplikace.

```
[RoutePrefix("api/product")]
public class ProductController : ApiController
{
    private readonly IProductService productService;
    private readonly IBrandService brandService;
    private readonly ICategoryService categoryService;
    private readonly ICartItemService cartItemService;

    public ProductController(IProductService productService, IBrandService brandService, ...)
    {
        this.productService = productService;
        this.brandService = brandService;
        ...
    }

    [Route("")]
    [AcceptVerbs("GET")]
    public IEnumerable<ProductViewModel> Index()
    {
        IEnumerable<Product> products = productService.GetAll(new string[] { "Brand", "ProductCategory" });
        IEnumerable<ProductViewModel> productViewModels = Mapper.Map<IEnumerable<Product>,
                                                          IEnumerable<ProductViewModel>>(products.Take(20));
       return productViewModels;
    }

    [Route("filter")]
    [AcceptVerbs("POST")]
    public FilteredDataResultViewModel<ProductViewModel> GetFilteredProduct(FilteringOption filterOption)
    {
        FilteredDataResultViewModel<ProductViewModel> resultData =
                                                        new FilteredDataResultViewModel<ProductViewModel>();
        IEnumerable<Product> filteredProducts = productService.GetFilteredProducts(filterOption, new string[] {
                                                                                "Brand", "ProductCategory" });
        resultData = FilterDataHelper<Product, ProductViewModel>.GetPagedFilteredDataResult(filteredProducts,
                                                                                        filterOption);
        return resultData;
    }
```

Kód 24 Ukázka Api kontroleru produktu

Na ukázce je předveden API kontroler, který posílá data na front-end Single page aplikace. Klasický kontroler by byl v podstatě stejný, pouze by dědil ze třídy *Controller*, jeho metody by měly ve většině případů návratový typ *ActionResult* a vracely by pohled s modelem.

5.6.1 Správa uživatelů a autentizace

V projektu diplomové práce se používá framework ASP.NET Identity, který zastřešuje správu uživatelů a uživatelských rolí. Integruje OWIN middleware, jenž mimo jiné slouží k autentizaci uživatelů. K vytváření uživatelů se používá *UserManager* umožňující nastavení bezpečnostních pravidel hesla a loginu uživatele a jejich validaci. Vývojář se nemusí starat o hashování hesla, to obstará sám framework. Pro přihlášení

uživatele se používá *SignInManager*. Pro oba managery je typicky možné vytvořit vlastní třídy manažerů, které z původních podědí, a následně upravit jejich chování.

```
    public class ApplicationUserManager : UserManager<User, int>
    {
        public ApplicationUserManager(IUserStore<User, int> store)
            : base(store) {}
        public static ApplicationUserManager Create(IdentityFactoryOptions<ApplicationUserManager>
                                                                    options, IOwinContext context)
        {
            var manager = new ApplicationUserManager(new UserStore<User, Role, int, UserLogin,
                                                UserRole, UserClaim>(context.Get<EShopDbContext>()));
            // setting username validation policy
            manager.UserValidator = new UserValidator<User, int>(manager)
            {
                AllowOnlyAlphanumericUserNames = false,
                RequireUniqueEmail = true
            };
            // setting passwords validation policy
            manager.PasswordValidator = new PasswordValidator
            {
                RequiredLength = 8,
                RequireNonLetterOrDigit = true,
                RequireDigit = true,
                RequireLowercase = true,
                RequireUppercase = true,
            };
            // setting lockup on repeated login fail
            manager.UserLockoutEnabledByDefault = true;
            manager.DefaultAccountLockoutTimeSpan = TimeSpan.FromMinutes(10);
            manager.MaxFailedAccessAttemptsBeforeLockout = 6;
}
```

Kód 25 Ukázka nastavení bezpečnostních pravidel pro vytváření uživatele v rámci vlastní implementace třídy UserManager

Autentizace uživatele v klasické webové aplikaci a Single Page Aplikaci probíhá relativně odlišným způsobem. Při přihlášení uživatele do klasické webové aplikace se na serveru vytvoří *session* objekt, který uchovává údaje o uživateli, drží tedy stav. Tento stav je v případě ASP.NET Identity ukládán na disk do *cookies*. Po dobu trvání *session* už server stav přihlášení nekontroluje a umožňuje uživateli pohyb na všechna místa webové aplikace, na nichž je potřeba pouhé přihlášení. Další kontroly probíhají až při requestech na stránky, které vyžadují speciální autorizaci – např. role admin, uživatel pavel.novak apod. *Session* objekt je ukončen odhlášením uživatele nebo vypršením časového limitu.

Pro Single Page Aplikaci je to složitější, jelikož na klientské straně není k dispozici *session* objekt ani *cookies* ze serveru – bezstavové. Ověření, zda má uživatel oprávnění navštívit danou stránku nebo provést konkrétní operaci, probíhá při každém requestu. K těmto účelům slouží JSON Web Token[19], zkráceně JWT nebo JW Token. Při přihlášení

[19] JSON Web Token je bezpečnostní token sloužící jako kontejner pro práva uživatele. Jedná se o JSON objekt, který se skládá ze tří částí: *header*, *payload* a *singature*. *Header* obsahuje informaci, že se jedná o JWT, a o způsobu šifrování (např. *HS256*). *Payload* zahrnuje samotné informace práv a rolí uživatele. *Signature* obsahuje informace *hlavičky* a *obsahu* v base64 formátu zašifrované podle šifrovacího algoritmu uvedeného v hlavičce. [43]

se na serveru vytvoří JWT, jenž se pošle na front-end aplikace, kde je nutné jej uložit. Při každém dalším requestu se posílá uložený JWT v hlavičce requestu zpět na server a zde se tokeny porovnají. Po ověření zašle server odpověď na front-end – ten podle výsledku buď zobrazí požadovaný obsah, nebo přesměruje uživatele na stránku s přihlášením. K autentizaci, vytváření a ověřování JWT na platformě ASP.NET slouží OWIN middleware.

Kontrolery a jeho metody se dají omezit autentizačními a autorizačními pravidly.

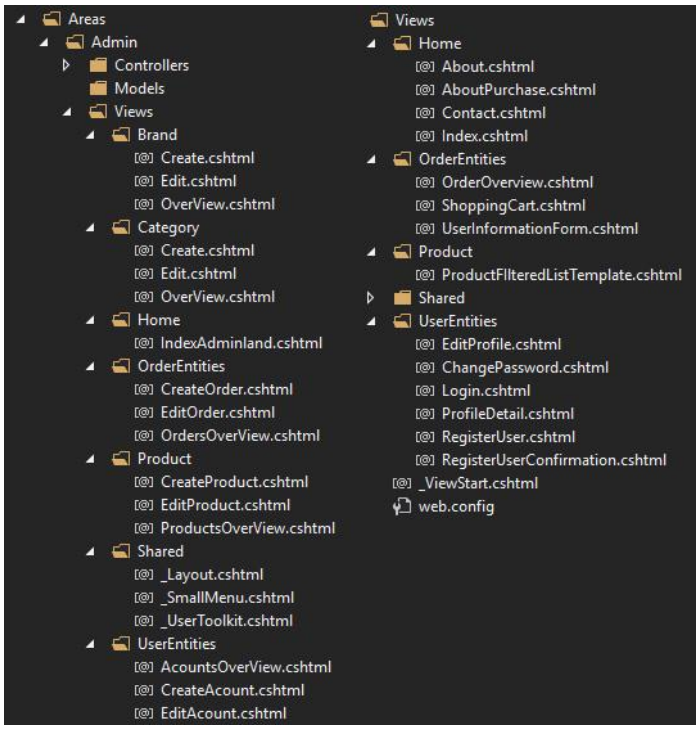
```
[Authorize]
[RoutePrefix("api/product")]
public class ProductController : ApiController
{
    ...
    [Route("")]
    [AcceptVerbs("GET")]
    public IEnumerable<ProductViewModel> Index()
    {
        IEnumerable<Product> products = productService.GetAll(new string[] { "Brand",
                                                                    "ProductCategory" });
        IEnumerable<ProductViewModel> productViewModels = Mapper.Map<IEnumerable<Product>,
                                            IEnumerable<ProductViewModel>>(products.Take(20));
        return productViewModels;
     }

    [Authorize(Roles="Admin")]
    [AcceptVerbs("DELETE")]
    [Route("remove/{id:int}")]
    public HttpResponseMessage DeleteProduct(int id)
    {
        try
        {
            Product curProduct = productService.GetById(id);
            if (curProduct == null) return Request.CreateErrorResponse(HttpStatusCode.NotFound,
                                                                    "Product not found");
            productService.Delete(curProduct);
            productService.SaveChangesToDB();
            return new HttpResponseMessage(HttpStatusCode.OK);
        }
        catch (Exception e)
        {
           return Request.CreateErrorResponse(HttpStatusCode.BadRequest,
                                            "Could not delete product from database");
        }
```

Kód 26 Ukázka autorizace kontroleru a jeho metod

K tomu je možné využít anotace *[Authorize]*. Kontroler nebo metody označené anotací *[Authorize]* smí použít pouze přihlášený, autentizovaný uživatel. V samotné anotaci je možné specifikovat role nebo konkrétní uživatele, pro které je akce povolená. Z ukázky výše je patrné, že se všechny metody kontroleru provedou pouze pro přihlášené uživatele. Smazání produktu je navíc dostupné pouze pro uživatele v roli administrátora. Metody označené atributem *[AllowAnonymous]* budou ostatní autorizační a autentizační pravidla ignorovat a obslouží i nepřihlášeného uživatele.

5.7 Presentation Layer – View

Jak už bylo výše uvedeno, v klasické webové aplikaci kontrolery vrací pohledy s modelem dat. Z těchto pohledů view engine vygeneruje HTML, a to je odesláno na klientskou stranu, kde se zobrazí v prohlížeči. Zdrojový kód pohledu je kombinací klasického HTML kódu a skriptovacího kódu v jazyce C#. Razor view engine umožňuje použití řady vestavěných nástrojů a komponent za pomoci *HTML helperu* (*@Html.*) a *URL helperu* (*@Url.*). Pohledy jsou soubory typu *.cshtml* a jsou uloženy v adresáři *Views* v hlavním projektu aplikace, popřípadě v adresáři *Views* v jednotlivých *Areas*. Mimo soubory pohledů jsou ve složce *Views* umístěny ještě soubory *web.config* a *_ViewStart.cshtml*. *Web.config* je konfigurační soubor, který specifikuje použitý view engine (v tomto případě Razor) a zajišťuje, aby se k pohledu nedalo přistoupit přímo, ale jedině přes kontroler. *_ViewStart.cshtml* specifikuje pouze vstupní šablonu – hlavní, sdílený pohled, do něhož se ostatní pohledy vkládají.



Obrázek 10 Struktura pohledů klasické webové aplikace

Pohledy lze prakticky rozdělit do tří skupin:

- **Hlavní šablona –** hlavní, sdílený pohled, který je zobrazen vždy – statická část webových stránek, jež se nemění. Do jeho těla se vkládají klasické pohledy. Cesta k hlavní šabloně je nastavena v souboru *_ViewStart.cshtm.*

- **Klasický pohled** – klasické pohledy reprezentují obsahy jednotlivých webových stránek a jsou vkládány do hlavní šablony. Pohled je zobrazen vždy jeden, může ale obsahovat variabilní počet částečných (parciálních) pohledů.
- **Parciální pohled** – lze chápat jako znovupoužitelnou komponentu. Nereprezentuje celou webovou stránku jako klasický pohled, ale pouze její část. Je možné jej znovupoužít ve variabilním množství různých pohledů.

Návrh pohledu je demonstrován na ukázce pohledu pro vytvoření nového produktu. Pohled byl pro demonstrativní pohledy mírně upraven, aby pokrýval většinu náležitostí a funkcionalit, s nimiž se může vývojář při jeho návrhu setkat.

```
[AcceptVerbs("GET")]
[Route("new")]
public ActionResult CreateProduct()
{
    ViewBag.CategoriesSelectList = new SelectList(categoryService.getAll(), "CategoryId", "Name");
    ViewBag.BrandsSelectList = new SelectList(brandService.getAll(), "BrandId", "Name");
    return View();
}

[AcceptVerbs("POST")]
[Route("new")]
[ValidateAntiForgeryToken]
public ActionResult CreateProduct(ProductCreateOrEditViewModel productData)
{
    if (ModelState.IsValid)
    {
        try
        {
            Product newProduct = Mapper.Map<ProductCreateOrEditViewModel, Product>(productData);
            productService.Create(newProduct);
            return RedirectToAction("Index");
        }
        catch (Exception e)
        {
            return View(productData);
        }
    }
    else
    {
        return View(productData);
    }
}
```

Kód 27 Ukázka vytvoření nového pohledu v akci kontroleru

V případě pohledů s formuláři jsou potřeba metody dvě. Obě se vztahují ke stejnému pohledu. Jedna vrací samotný pohled a druhá zpracovává uživatelský vstup. V parametru přijímá model a v případě nevalidně vyplněných dat se uživateli opět vrátí pohled s již vyplněnými poli, v opačném případě je model zpracován a uživatel přesměrován. V pohledu se pracuje pouze s daty, které jsou pro něj specifické. Toho je docíleno použitím view modelu, kde jsou nadefinovaná pouze potřebná data. Stejného výsledku lze docílit i pomocí nabindování vybraných atributů pomocí *include* nebo *exclude* v parametru metody:

```
public ActionResult Create([Bind(Include = "Name,Price,Description" )] Product product)
{}
```

```
@model Eshop.ViewModels.ActionViewModels.ProductCreateOrEditViewModel
@{
    Layout = "~/Areas/Admin/Views/Shared/_Layout_Adminland.cshtml";
}
@helper InputField(string propertyName)
{
    <div class="form-group">
        @Html.Label(propertyName, new { @class = "control-label col-md-2" })
        <div class="col-md-10">
            @Html.TextBox(propertyName, new { @class = "form-control" })
            @Html.ValidationMessage(propertyName, new { @class = "text-danger" })
        </div>
    </div>
}

<h2>Nový produkt</h2>
@using (Html.BeginForm("Create", "Product", new { area = "Admin" }, FormMethod.Post, new { @class = "form-horizontal",
                                                                                           enctype = "multipart/form-data" }))
{
    @Html.AntiForgeryToken()
    <div class="form-horizontal">

        @InputField("Name");

        <div class="form-group">
            @Html.LabelFor(model => model.Price, new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.TextBoxFor(model => model.Price, new { @class = "form-control", @type = "number" })
                @Html.ValidationMessageFor(model => model.Price)
            </div>
        </div>

        @InputField("Description");
        @InputField("ShortDescription");
        @InputField("StorageState")

        <div class="form-group">
            @Html.LabelFor(model => model.ProductCategory, "Kategorie", new { @class = "control-label col-md-
                                                                                        2" })
            <div class="col-md-10">
                @Html.DropDownListFor(model => model. ProductCategory, ViewBag.CategoriesSelectList as SelectList)
                @Html.ValidationMessageFor(model => model. ProductCategory)
            </div>
        </div>
        <div class="form-group">
            @Html.LabelFor(model => model.Brand, "Značka", new { @class = "control-label col-md-2" })
            <div>
                @Html.DropDownListFor(model => model.Brand, ViewBag.BrandsSelectList as SelectList)
                @Html.ValidationMessageFor(model => model.Brand)
            </div>
        </div>

        <div class="form-group ">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Create" class="btn btn-default" />
            </div>
        </div>
    </div>
@Html.ActionLink("Zrušit", "Index","Product", new { area = "Admin" })
}
@Scripts.Render("~/bundles/jqueryval")
@section Scripts{
    <script src="http://ajax.googleapis.com/ajax/libs/jqueryui/1.8/jquery-ui.min.js"></script>
    <script language="javascript" type="text/javascript">
        function exampleFunction() { console.log("just show case"); }
    </script>
}
```

Kód 28 Ukázka pohledu pro vytvoření produktu

Pohled je silně typový, a nabízí tak *intellisence* pro model typu *ProductCreateOrEditViewModel*. Dále je specifikovaná cesta k hlavní šabloně. Běžně není potřeba uvádět, protože se používá výchozí šablona specifikovaná v souboru *_ViewStart.cshtml*, ale protože má administrativní část aplikace odlišný design, je nutné cestu k jiné šabloně definovat. HTML helpery výrazně usnadňují práci a nabízejí řadu již předpřipravených komponent a funkcí pro urychlení a zjednodušení tvorby pohledu. V ukázce je použit HTML helper pro tvorbu formuláře, jednotlivých formulářových prvků, jejich popisků a validaci. Některé formulářové prvky se několikrát opakují, mapují se pouze na jiné atributy modelu. Aby nebyl kód pohledu zbytečně dlouhý, byla pomocí helperu vytvořena metoda *InputField(string propertyName)*, jíž lze použít jako šablonu pro různé formulářové prvky – viz ukázka výše. Po odeslání formuláře je volána *POST* metoda *Create* na kontroleru *Product* v sekci *Admin*, do které je odeslán model s vyplněnými daty ke zpracování. Obalující blok *@using{}* vygeneruje uzavírací tag *</form>* – možné docílit manuálním uzavřením formuláře pomocí *@Html.EndForm()*.

Validační zprávy lze definovat přímo v pohledu nebo v anotacích u jednotlivých atributů modelu. Při odeslání formuláře server zkontroluje validitu modelu a případně zobrazí validační zprávy u formulářových polí. Nicméně je dobrá praxe validovat už na straně klienta, aby se počet requestů na server omezil.

Řádek s *@Html.AntiForgeryToken()* zajišťuje ochranu proti *Cross-Site Request Forgery*. *POST* metoda *Create* musí v obsahovat taktéž anotaci *[ValidateAntiForgeryToken]*, aby bylo ochrany docíleno.

Dále je v pohledu použit helper pro vytvoření odkazu. *@Html.ActionLink()* vytvoří HTML tagy *<a>*. V parametrech přebírá text k zobrazení, název akce, kterou má provést po kliknutí. Dále přebírá název kontroleru, jemuž akce náleží a v případě potřeby je možné specifikovat *Area* nebo CSS třídy a další.

Ve spodní části ukázky je předvedeno použití JS scriptů z různých zdrojů. *@Scripts.Render()* načte zdrojové scripty ze specifického JS souboru obsaženém v projektu. Do bloku *@section.Scripts* je možné vložit vlastní Javascriptový zdrojový kód přímo nebo jednoduše odkázat na nějaké externí zdroje.

5.8 Front-End Sinle Page Aplikace – React Applikace

V rámci Single Page Aplikace vytvářené pro diplomovou práci je *View* z architektury MVC reprezentován separátním projektem postaveném na JS knihovně React. Aplikace běží na vlastním serveru. K těmto účelům je pro vývoj použit *webpack-dev-server*, což je malý Express.js server běžící na Node.js. Pro kompilaci zdrojového kódu a načítání modulů různého typu je v projektu použit *Webpack*. Dále je integrována knihovna *Redux*, která se stará o správu stavu aplikace.

5.8.1 Konfigurace

Jelikož se jedná o poměrně malý projekt není konfigurace příliš rozsáhlá. Konfigurace samotného projektu se provádí v souboru *package.json*. Jak už bylo uvedeno výše, v projektu se používá *Webpack*, jenž má vlastní konfigurační soubor *webpack.config.js*.

package.json

Každý *npm* balíček obsahuje *package.json* soubor nacházející se obvykle v kořenovém adresáři projektu. Tento JSON soubor obsahuje různá metadata relevantní k projektu. Umožňuje tak *node package manageru* identifikovat projekt a řídit jeho závislosti na jiné balíčky. *Package.json* obsahuje informace jako verze, název a popis projektu, dále také licenční informace, konfigurační data a další. Název a verze projektu jsou povinná pole, bez nich *npm* projekt nenainstaluje.

```
{
  "name": "reactapp",
  "version": "1.0.0",
  "description": "online catalog of musical instruments",
  "main": "index.js",
  "scripts": {
    "start": "webpack-dev-server --hot",
    "build": "webpack -p",
    "develop": "webpack -d --watch"
  },
  "keywords": [],
  "author": "Přemysl Šulc",
  "license": "ISC",
  "dependencies": {
    "react": "^15.4.2",
    "react-dom": "^15.4.2",
    "react-redux": "^5.0.6",
    "react-router": "^4.1.1"
    ...
  },
  "devDependencies": {
    "babel-core": "^6.22.1",
    "babel-loader": "^6.2.10",
    "babel-plugin-transform-object-rest-spread": "^6.26.0",
    "babel-preset-es2015": "^6.24.1",
    "babel-preset-react": "^6.22.0",
    "css-loader": "^0.28.7",
    "file-loader": "^0.11.2",
    "html-webpack-plugin": "^2.28.0",
    "json-loader": "^0.5.4",
    "style-loader": "^0.18.2",
    "url-loader": "^0.5.9",
    "webpack": "^1.14.0",
    "webpack-dev-server": "^1.16.2"
  }
}
```

Kód 29 Ukázka package.json souboru

Na ukázce výše je vyobrazen obsah souboru *package.json*. Soubor obsahuje základní informace o projektu jako jeho název, verzi, popis a další. Sekce *scripts* umožňuje skládat variabilní počet scriptů dohromady a spustit je pod vlastním příkazem. Například spuštěním příkazu *npm start* se spustí *dev-server* a *hot module*, který sleduje změny v kódu. Při uložení kód překompiluje a změny rovnou promítne v prohlížeči. Sekce *dependencies* a *devDependencies* specifikují balíčky, jež se v rámci projektu používají. Při spuštění příkazu *npm install* budou všechny balíčky uvedené v závislostech doinstalovány. Mezi těmito sekcemi je ovšem rozdíl. Jak už názvy napovídají, *devDependencies* zahrnují balíčky

nutné pro vývoj projektu, zatímco v *dependencies* jsou pouze takové, které jsou potřeba k běhu aplikace na produkci.

webpack.config.json

V ukázce níže je obsažena veškerá konfigurace Webpacku, která je pro projekt esenciální. Sekce *entry* definuje vstupní, hlavní JavaScriptový soubor aplikace. *Output* naopak určuje adresář, do něhož se vygeneruje výsledný *bundle* a jeho název. V sekci *devServer* probíhá nastavení nutných náležitostí serveru, na kterém běží vývoj. Dále jsou už pouze nastaveny všechny *loadery* a *pluginy*, které Webpack pro projekt používá.

```
var config = {
  entry: [
    './app/index.js'
  ],
  output: {
    path: './dist',
    filename: "bundle.js",
    publicPath: '/'
  },

  devServer: {
    inline: true,
    port: 6060,
    headers: { "Access-Control-Allow-Origin": "*" },
    historyApiFallback: true
  },
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        exclude: /node_modules/,
        loader: 'babel',

        query: {
          presets: ['es2015', 'react'],
          plugins: ['transform-decorators-legacy']
        }
      },
      {
        test: /\.json$/,
        loader: 'json-loader'
      },
      {
        test: /\.(woff|woff2|eot|ttf|svg|jpg)$/,
        loader: 'file?name=fonts/[name].[ext]'
      },
      { test: /\.(png|woff|woff2|eot|ttf|svg|jpg)$/, loader: 'url-loader?limit=100000' },
      { test: /\.css$/, loader: "style-loader!css-loader" },
      { test: /\.svg$/, loader: 'url?limit=65000&mimetype=image/svg+xml&name=public/fonts/[name].[ext]' },
      { test: /\.woff$/, loader: 'url?limit=65000&mimetype=application/font-woff&name=public/fonts/[name].[ext]' },
      { test: /\.woff2$/, loader: 'url?limit=65000&mimetype=application/font-woff2&name=public/fonts/[name].[ext]' },
      { test: /\.[ot]tf$/, loader: 'url?limit=65000&mimetype=application/octet-stream&name=public/fonts/[name].[ext]' },
      { test: /\.eot$/, loader: 'url?limit=65000&mimetype=application/vnd.ms-fontobject&name=public/fonts/[name].[ext]'}
    ]
  },
  plugins: [HTMLWebpackPluginConfig]
};
```

Kód 30 Ukázka konfigurace Webpacku

5.8.2 Routování

Single page aplikace je sice z principu tvořena pouze jedinou stránkou, avšak z pohledu navigace se na venek chová jako klasická, vícestránková aplikace. Uživatel pomocí navigačních prvků přechází mezi pohledy a v závislosti na tom se adekvátně mění i aktuální URL adresa. Avšak React knihovna v sobě nemá zabudované nástroje na routování, a proto je pro tyto účely nutné použít externí knihovnu. Alternativ je více, autor ale sáhl po nejpopulárnější a nejpoužívanější knihovně *React-router*. Pro příkladovou aplikaci je použita verze *2.**. Nynější verze *4.** prodělala poměrně rozsáhlé změny, a proto se autor rozhodl zůstat u starší verze knihovny.

```
<Provider store={store}>
    <Router history={browserHistory}>
        <Route path='/' component={Layout}>
            <IndexRoute path="/" component={HomePage} />
            <Route path="/about" component={About} />
            <Route path="/complaint" component={Complaint} />
            <Route path="/delivery" component={Delivery} />
            <Route path="/service" component={ServiceAndServicies} />
            <Route path="/administration/products/overview" component={ProductAdministration}
                                                                onEnter={checkAuth} />
            <Route path="/administration/products/:id" component={AddOrEditProductForm} onEnter={checkAuth} />
            <Route path="/administration/categories/overview" component={CategoryAdministration}
                                                                    onEnter={checkAuth} />
            <Route path="/administration/categories/:id" component={AddOrEditCategoryForm}
                                                               onEnter={checkAuth} />
            <Route path="/administration" component={AdministrationIndex} onEnter={checkAuth}/>
            <Route path="/product/:id" component={ProductDetail} />
            <Route path="/:category" component={ProductIndexTemplate} />
            <Route path="/register" component={Register} />
            <Route path="/signin" component={LoginPage} />
            <Route path="/not-found" component={NotFound} />
            <Redirect from='*' to='/not-found' />
        </Route>
    </Router>
</Provider>
```

Kód 31 Ukázka routování

Router je většinou umístěn do hlavní komponenty layoutu v místě, kde je potřeba měnit obsah stránky. V rámci příkladové aplikace zastupuje obsahovou část mezi hlavičkou a patičkou stránky – jejich obsah se v rámci navigace nemění.

K navigaci mezi pohledy slouží komponenta *<Link to=''>*, v novějších verzích knihovny také komponenta *<NavLink to=''>*, které se v atributu ***to*** definuje URL adresa, na kterou má uživatele přesměrovat. Router porovná cílovou URL adresu s cestami nadefinovaných *Route* komponent a v případě shody vloží do pohledu patřičnou komponentu a přistoupí na žádané URL. Pokud pro kýžené URL neexistuje žádná *Route* komponenta, je uživatel přesměrován na pohled *NotFound*. Při vstupu na administrátorské pohledy se volá metoda *checkAuth()*, která ověřuje, zda je uživatel opravdu přihlášen. Pokud není, je odkázán na pohled přihlášení.

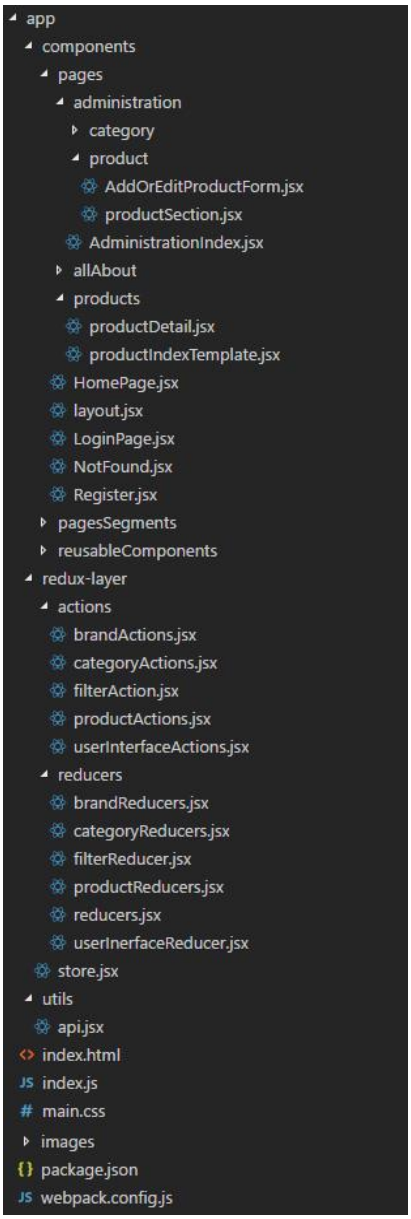
React-router dokáže spravovat historii navigace zásluhou *browserHistory*, který přebírá v *props*. Díky tomu je možné pro navigaci historií používat v prohlížeči tlačítko *backwards* a *forward*.

Celý *Router* je obalen komponentou *Provider*. Jedná se o komponentu z knihovny *React-redux*, jenž do *props* přebírá objekt *store*, ve kterém je uložen stav aplikace. Ta umožňuje všem komponentám zanořeným uvnitř přistupovat k objektu *store* a číst nebo manipulovat s daty aplikace.

5.8.3 Struktura projektu

Aplikace je strukturována do tří hlavních stavebních vrstev:

- **Components** – vrstva komponent, ze kterých se skládá uživatelské rozhraní aplikace.
- **Redux Layer** – vrstva zodpovědná za zprávu stavu aplikace.
- **API Layer –** vrstva zodpovídající za komunikaci s back-endem aplikace.



Obrázek 11 Struktura front-end JS aplikace

V kořenové složce projektu se dále nacházejí konfigurační soubory projektu, adresáře s obrázky, CSS soubory, fonty, vstupní JavaScriptový soubor a hlavní HTML soubor. Jelikož se jedná o Single Page Application, aplikace obsahuje pouze jediný HTLM soubor – *index.html*. Uvnitř jeho těla najdeme pouze *div* element sloužící jako kontejner celého obsahu aplikace a referenci na vstupní JavaScriptový soubor aplikace – viz ukázka. Webpack je možné nastavit tak, aby soubor *index.html* vytvořil sám, a to včetně reference na vstupní JavaScriptový soubor *index.js*.

```
<body>
  <div id="app_container"></div>
  <script src="/index.js"></script>
</body>
```

Kód 32 Ukázka souboru index.html

```
import ReactDOM from 'react-dom';
import Layout from './components/pages/layout.jsx';

ReactDOM.render(
    <Layout />
  document.getElementById('app_container')
);
```

Kód 33 Ukázka vstupního JS souboru

Vstupní JavaScriptový soubor *index.js* se stará o inicializaci vykreslení celé aplikace. Samotný soubor pouze importuje potřebné balíčky a kořenovou komponentu aplikace, jíž vykreslí v prohlížeči voláním metody *ReactDOM.render()*. Metoda přebírá v parametrech komponentu k vykreslení a DOM element, do něhož má obsah vložit.

5.8.3.1 API Layer

API vrstva slouží jako spojovací most mezi klientskou částí a serverem. Zde se nacházejí veškeré metody, jež pomocí asynchronních volání zasílají HTTP requesty na back-end aplikace. Z tohoto místa se odesílají požadavky na data z databáze, zároveň je však tato vrstva zodpovědná za odesílání nových dat na server nebo autentizačních požadavků a jiné. API volání jsou roztříděná podle jednotlivých kontrolerů, kterým náleží.

```
import axios from "axios";
const urlBase = "http://localhost:52491/api/";
module.exports = {
    // ========== PRODUCTS ========== //
    getFilteredProducts: function(filterOption) {
        return axios.post(urlBase + "product/filter", filterOption)
        .then(function (response) {
            return response.data
        }).catch(function(err){
            console.log("FrontEnd API layer error: filtering products failed")
        })
    },
    createProduct: function (data) {
        return axios.post(urlBase + "/product/new", data)
            .then(resource => {
                return resource;
            })
            .catch(function (error) {
                console.log("FrontEnd API layer error : product was not created");
            });
    },
    fetchProductDetail: function (id) {
        return axios.get(urlBase + `product/${id}`)
            .then(function (response) {
                return response.data;
            }).catch(function (response) {
                console.log("FrontEnd API layer error: product detail data wasnt fetched");
            });
    },
```

Kód 34 Ukázka API vrstvy

Na ukázce výše je zobrazena část API vrstvy pro entity produktu. K odeslání asynchronních HTTP requestů je použita populární knihovna *Axios*. Alternativou této knihovny může být často používaná knihovna R*equest*.

5.8.3.2 Redux Layer

Ve vrstvě *Redux Layer* je do projektu integrována knihovna *React-redux*. Tato vrstva zodpovídá za správu stavu celé aplikace a nabízí tak API pro čtení a modifikaci aplikačních dat. Knihovna *React-redux* se skládá ze tří základních konstruktů, jenž mají každý z nich svou funkci a odpovědnost. Podle těchto zodpovědností je rozdělena i *Redux Layer* na dílčí podvrstvy:

- **store**
- **actions**
- **reducers**

5.8.3.2.1 Store

V objektu *store* jsou udržována kompletní data aplikace – stav aplikace. Při modifikaci stavu aplikace se nikdy nesmí modifikovat aktuální objekt *store*, je třeba vytvořit nový, s aktuálními daty. Z toho plyne jedna z výhod reduxu, a to, že je možné jednotlivé stavy ukládat, držet v paměti a v případě potřeby mezi nimi přepínat. *Store* je předáván komponentě *<Provider></Provider>*, která je taktéž součásti knihovny *React-redux* a ta zpřístupňuje stav aplikace všem zanořeným komponentám uvnitř. Objekt *store* se vytváří voláním metody *createStore()*, jež v parametru přebírá hlavní *reducer*. V ukázce níže je demonstrován modul pro vytvoření objektu *store*.

```
import { createStore } from 'redux';
import { reducers } from './reducers/reducers.jsx';

export let store = createStore(
    reducers,
);

store.subscribe(() => {
    console.log("state has changed: ",  store.getState());
})
```

Kód 35 Ukázka Store modulu

5.8.3.2.2 Reducers

Na tomto místě se provádí samotná změna stavu aplikace a *reducer* je funkce provádějící tuto změnu. V parametrech obdrží aktuální stav aplikace a objekt akce. V objektu akce jsou zpracovaná data a identifikátor akce, která se má nad daty provést. *Reducer* je v podstatě velký switch, který podle identifikátoru pozná, jakou změnu nad stavem má provést. Výstupem *reducer* funkce je zcela nový stavový objekt, jenž pak aplikace používá. Pro *reducer* platí několik pravidel. Měl by mít co nejrovnější strukturu (ideálně nanejvýš jedna úroveň zanoření dat) a taktéž by měl pouze vytvářet nový stavový objekt. Zpracování dat je úlohou jednotlivých akcí. Z toho důvodu se hlavní *reducer* rozpadá na více menších a jednodušších *reducerů*, kde každý zodpovídá pouze za dílčí část aplikačního stavu. V modulu s hlavním *reducerem* se následně složí dohromady pomocí funkce *combineReducers()* a použijí jako jeden celek – hlavní *reducer*.

```
const INITIAL_STATE = {
    products: [],
    error: null,
    product: {
        Name: "",
        Price: 0,
        Description: "",
        StorageState: "",
        Category: {},
        Brand: {}
    }
};
/**
* @param {Object} state - Default aplication state
* @param {Object} action - Action from action creator
* @returns {Object} New state
*/
export function productReducer(state = INITIAL_STATE, action) {
    switch (action.type) {
        case "FETCH_PRODUCTS":
            return {
                ...state,
                products: action.payload
            };
        case "FETCH_PRODUCT_DETAIL":
            return {
                ...state,
            product: action.payload
        };
        case "ADD_PRODUCT":
            let newState = Object.assign({}, state);
            newState.products = state.products.concat(action.payload)
            return newState;
       ...
```

Kód 36 Ukázka reduceru pro produkt

5.8.3.2.3 Actions

Akce je z pohledu knihovny *React-redux* objekt, který předává reduceru hotová data k uložení a identifikátor akce, podle kterého reducer vybere správnou operaci k provedení. Aby bylo vše přehledné, je pro každý reducer vytvořen vlastní modul s funkcemi, jež předávají objekt akce do reducer vrstvy. Tato vrstva je tedy zodpovědná za zpracování dat a iniciaci uložení nového stavu reducerem. Celý proces se iniciuje voláním funkce *dispatch()* na objektu *store*, kdy se v parametru metody předává právě objekt akce.

```
import { store } from './../store.jsx';
import api from './../../utils/api.jsx';

export function fetchProductDetail(id, th, callback){
    api.fetchProductDetail(id)
    .then(function(data){
        store.dispatch({
            type: "FETCH_PRODUCT_DETAIL",
            payload: data
        })
        callback(th);
    }).catch(function(err){
        store.dispatch({
            type: "CATCH_ERROR",
            payload: err
        })
    })
}
export function createProduct(product, callback) {
    api.createProduct(product)
        .then(function (response) {
            store.dispatch({
                type: "ADD_PRODUCT",
                payload: product
            })
            callback();
        }).catch(function (err) {
            store.dispatch({
                type: "CATCH_ERROR",
                payload: err
            })
        });
}
```

Kód 37 Ukázka Actions vrstvy

Na ukázce výše je ukázána část modulu akcí pro *productReducer*. Samotné funkce nejprve provedou zpracování daty a teprve posléze předá objekt akce na reducer vrstvu pomocí volání *store.dispatch({...})*. Metoda *createProduct()* nejprve odešle data z formuláře na server a v případě úspěšného uložení, kdy server pošle *Status Code 200*, se požádá reducer o uložení nových dat do *store*. Důležité je držet synchronní data s databází.

5.8.3.3 Components

Poslední vrstva front-endové části SPA je vrstva samotných komponent. Jedná se o komponenty, ze nichž je složeno celé uživatelské rozhraní. Slouží tak k vizuální prezentaci prvků stránek, ale taktéž k obsluze uživatelských vstupů a událostí. Komponenty je možné lehce strukturovat, zanořovat a sestavovat z jiných komponent. Obecně je dobrá praxe komponenty udržovat malé a jednoduché, z nichž se poté poskládají složitější celky. Komponenty uvnitř této vrstvy jsou roztříděné do tří kategorií:

- **ReusableComponents** – malé komponenty, ze kterých se skládají větší a které jsou uvnitř projektu použité vícekrát na různých místech. Jedná se například o stránkování, tabulky, filter, menu položky apod.
- **Pages** – komponenty, které reprezentují pohledy celých stránek – domovská stránka, detail produktu, kontakt, registrace a jiné.
- **PagesSegments** – části webové stránky – navigační menu, hlavička, patička apod.

V následující ukázce je předložen zdrojový kód komponenty pro výpis produktů různých kategorií. Produkty jsou zobrazeny včetně miniatury detailu a je možné je stránkovat a filtrovat. Už z tohoto popisu je patrné, že se bude jednat pouze o kontejner komponentu, která se skládá z menších celků. V tomto případě se jedná o komponenty *ProductFilter*, *Paginator* a *FilteredProductList*, jež jsou do této komponenty importovány a zanořeny. Tyto komponenty se samy skládají z dalších dílčích komponent.

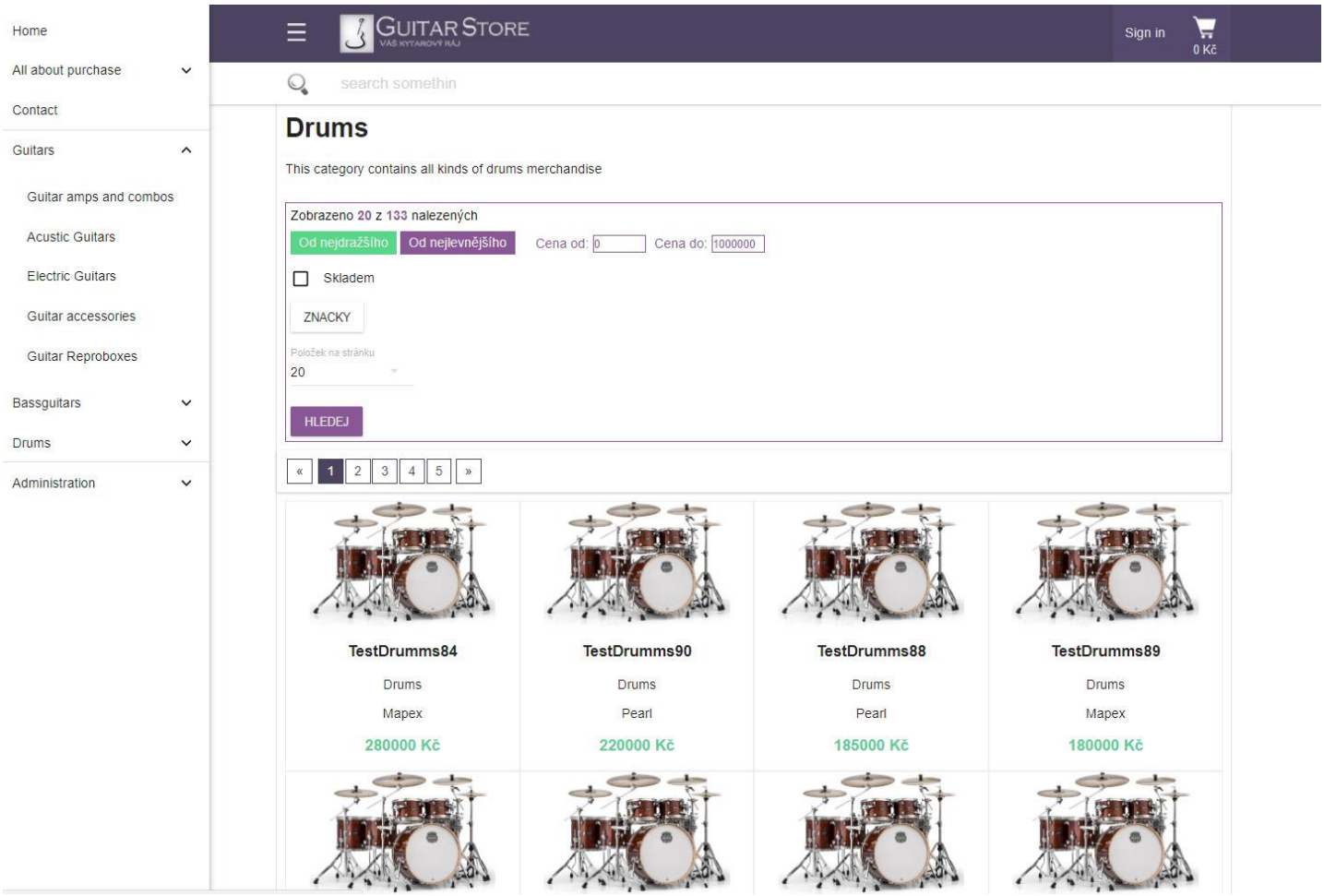
Na začátku zdrojového kódu je pomocí dekorátoru *@connect* vložena část dat z objektu *store* do *props* komponenty. Díky tomu má komponenta přístup k aktuálním datům aplikace. Komponentu, jež je ve skutečnosti třída, je dále nutné exportovat, aby ji bylo možné importovat a použít v nějaké další komponentě.

Uvnitř komponenty se nacházejí některé metody životního cyklu. *Construktor* nastavuje počáteční stav komponenty. V metodách *componentDidMount* a *componentDidUpdate* se hlídá, zda se nezměnila URL adresa, ze které se bere informace, o kategorii, pro kterou se mají zobrazit produkty. Při změně se odešle API volání na back-end, ve němž jsou specifikovány parametry filtru – kategorie, počet záznamů na stránku, způsob řazení, index stránky apod. Server následně nově vyfiltrované produkty odešle zpět a zde se uloží do *store*. Tím je vyvolán *render*, jenž stránku znovu překreslí.

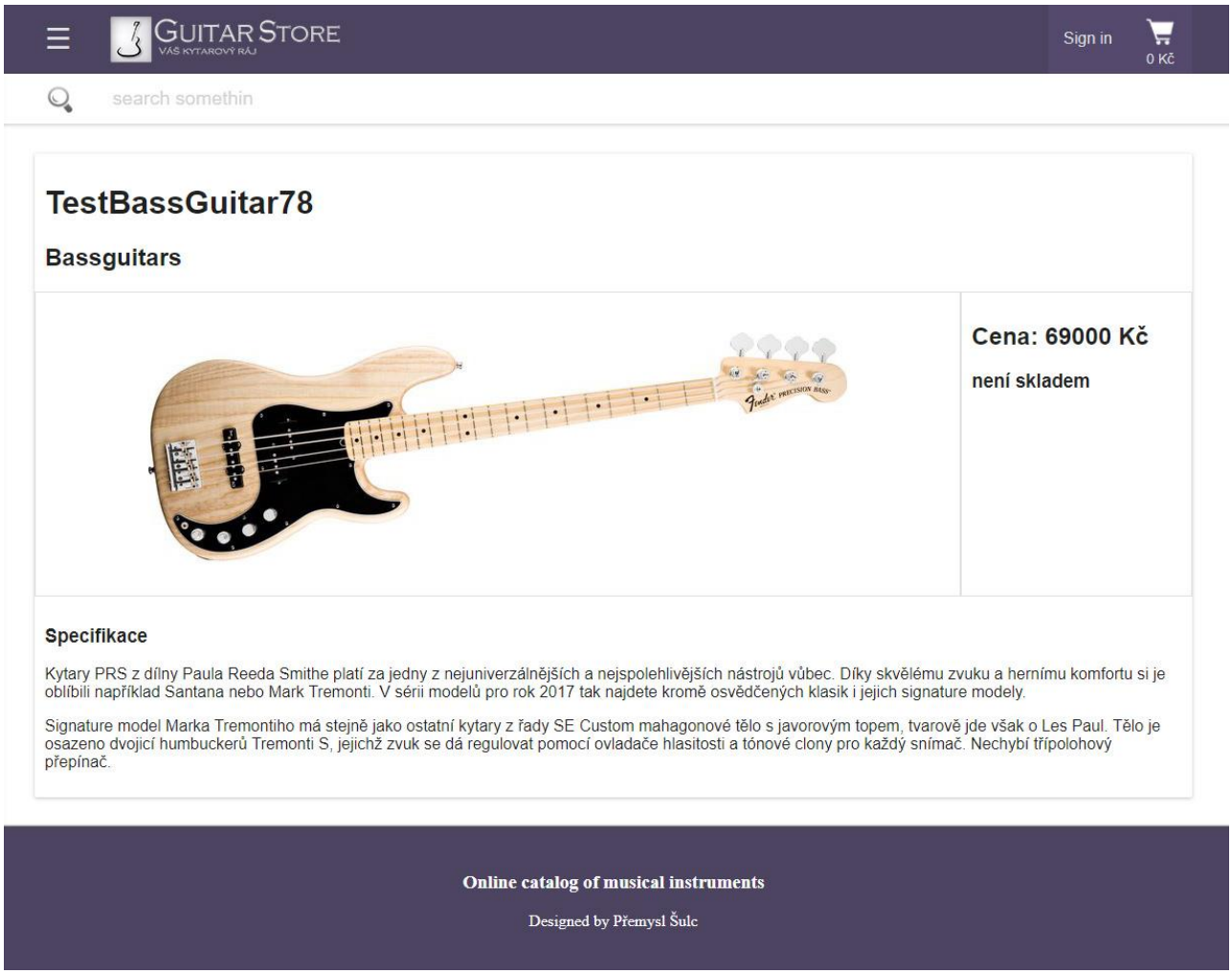
```
@connect((store) => {
    return {
        products: store.products.products,
        category: store.categories.category,
        filter: store.filter
    }
})
export default class ProductIndexTemplate extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            paramsIdentifier: ''
        }
    }
    componentDidMount(){
        const params = this.props.match.params.category
        if(params !== this.state.paramsIdentifier ){
            setFilterCategoryIdentifier(params);
            fetchFilteredItems("PRODUCTS");
            fetchCategory(params)
            this.state.paramsIdentifier = params;
        }
    }
    componentDidUpdate(){
        const params = this.props.match.params.category
        if(params !== this.state.paramsIdentifier ){
            this.state.paramsIdentifier = params;
        }
    }
    render() {
        const categoryName = this.props.category && this.props.category.Name;
        const categoryDescription = (this.props.category && this.props.category.Description)
                                    ? this.props.category.Description
                                    : "There should be some category description";
        return (
            <div>
                <div style={{padding: 5}}>
                    <h1>{categoryName}</h1>
                    <p>{categoryDescription}</p>
                </div>
                <ProductFilter categoryIdentifier={this.state.paramsIdentifier} />
                <Paginator filterIdentifier={"PRODUCTS"}/>

                <div>
                     <ProductFilteredList products={this.props.products} />
                     <Paginator filterIdentifier={"PRODUCTS"}/>
                </div>
            </div>
        )
    }
}
```

Kód 38 Ukázka komponenty šablony pro výpis produktů pro různé kategorie

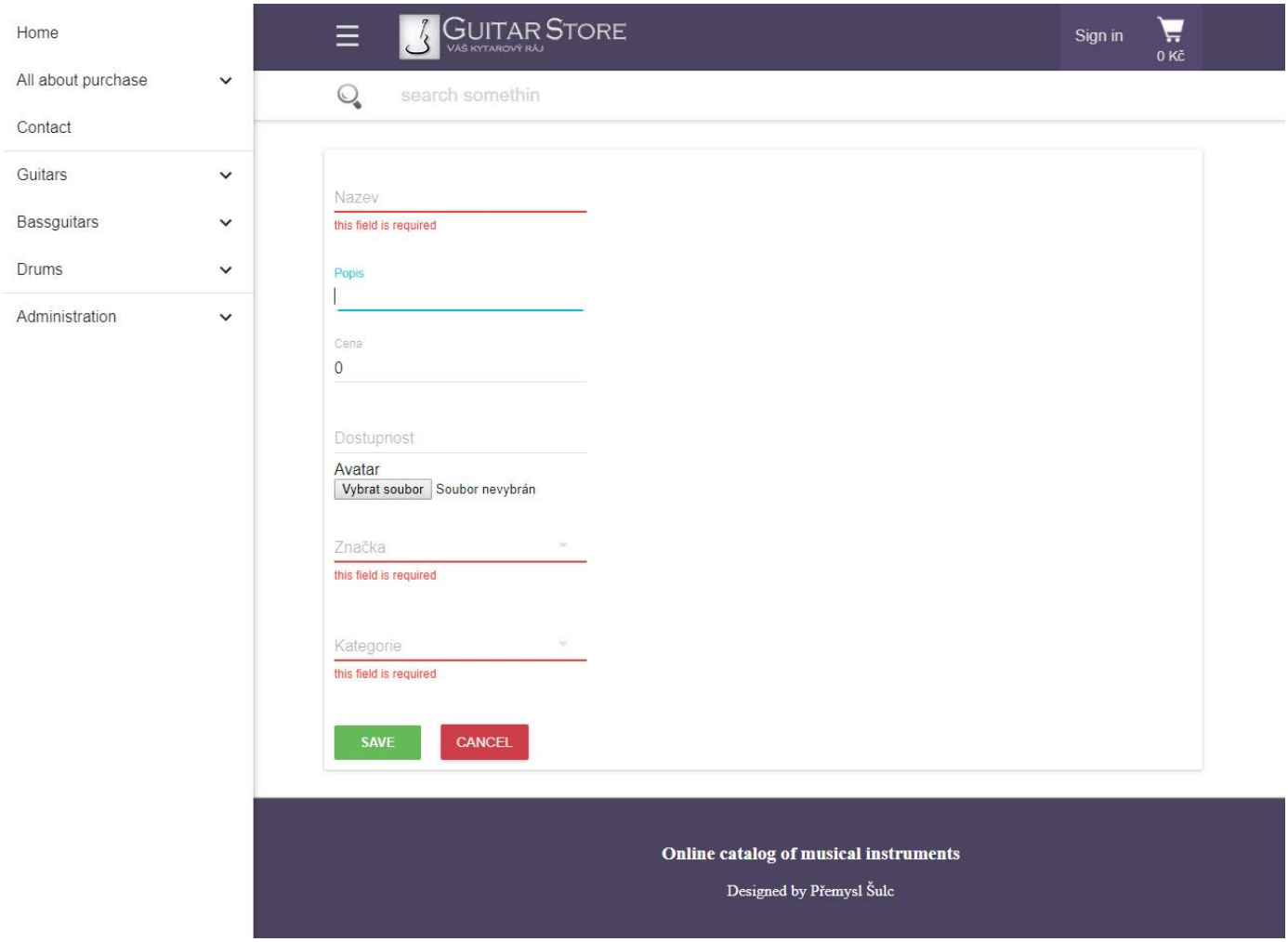
Snímky z aplikace



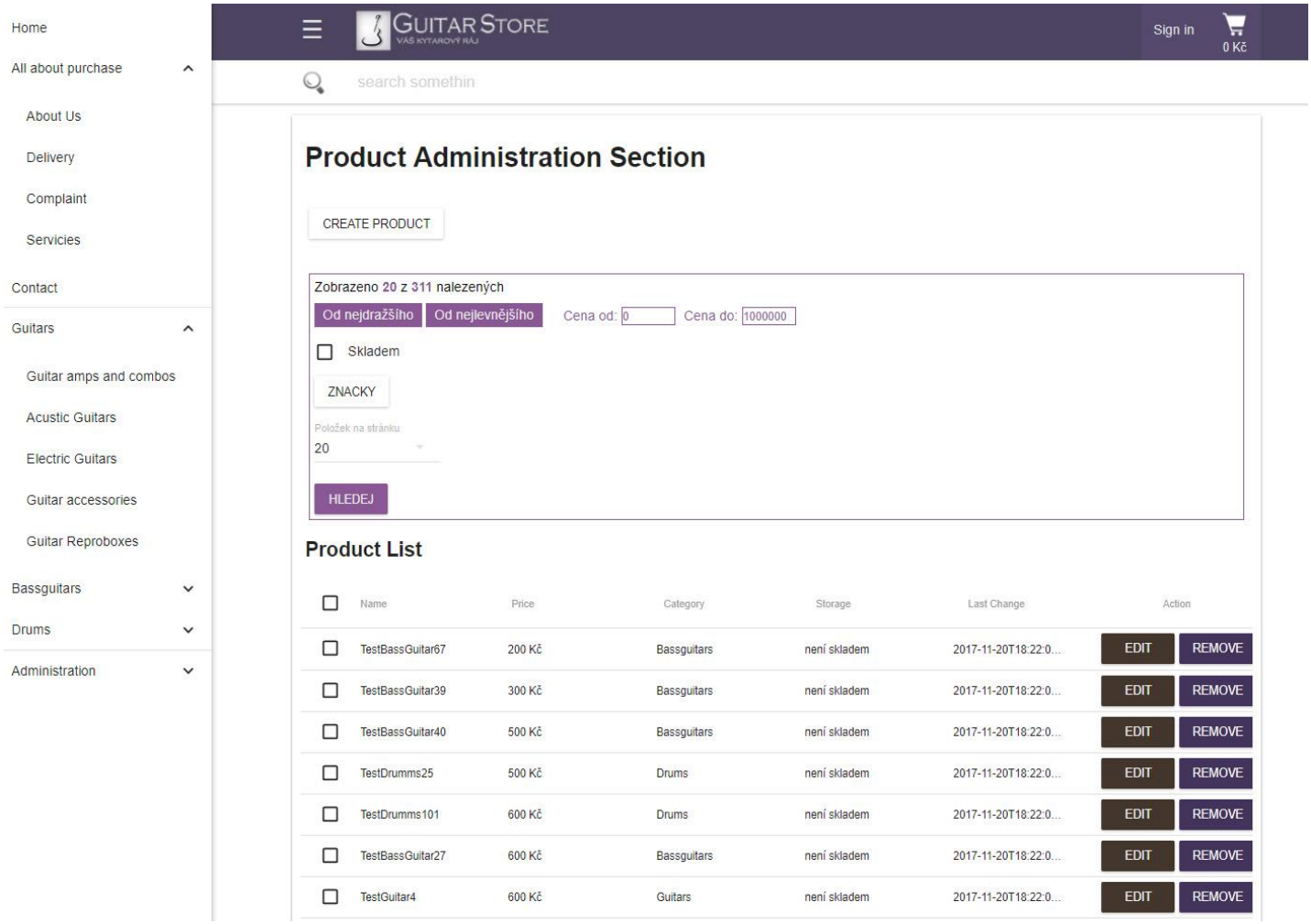
Obrázek 12 Ukázka výpisu produktů z kategorie bicích



Obrázek 13 Ukázka detailu produktu



Obrázek 14 Ukázka formuláře pro vytvoření / editaci produktu



Obrázek 15 Ukázka administrativního přehledu produktů

6 Shrnutí

Tato diplomová práce se zaobírala otázkami vývoje Single page aplikací a klasických serverových aplikací postavených na platformě ASP.NET MVC. V úvodní kapitole jsou představeny platformy .NET a ASP.NET. Jelikož je ASP.NET MVC součástí těchto větších celků, je záhodné, aby měl čtenář k dispozici dostatečně rozsáhlé podklady k pochopení problematiky v širším kontextu.

Následující kapitola se zabývá představením samotné architektury MVC, frameworku ASP.NET MVC a všech stěžejních nástrojů používaných v rámci vývoje na této platformě. Architektura MVC je rozdělena na složky Model, View a Controller, které jsou detailně analyzovány.

Další stěžejní částí diplomové práce jsou kapitoly věnující se tématu Single Page Application a knihovně React. Technologie SPA je zde detailně popsána a jsou objasněny principy aplikací postavených na této technologii. V neposlední řadě jsou uvedeny výhody a nevýhody oproti klasickým webovým aplikacím a definovány nejpoužívanější nástroje pro vývoj SPA. Kapitola o knihovně React uvádí čtenáře do problematiky návrhu aplikací za použití zmíněné knihovny v kombinaci s architektonickým vzorem Redux. Dále jsou diskutovány často používané nástroje, jež se při práci s Reactem používají. Jedná se například o Webpack, Node Package Manager, Node.js, Babel a další.

Závěrečná kapitola diplomové práce se zabývá ukázkou celého vývoje na komplexní úloze. Příkladová aplikace reprezentuje online katalog hudebních nástrojů a pokrývá všechny základní scénáře, které vývojář řeší při stavbě jakékoli webové aplikace. Práce představuje architekturu projektu, popisuje dílčí implementaci klientské i serverové části, implementaci jednotlivých vrstev aplikace a jejich propojení.

Při vývoji serverové části aplikace se architektura MVC ukázala jako nedostačující a bylo nutné jí rozšířit o další logické vrstvy, a to o datovou, servisní, prezenční apod. Pro ulehčení vývoje je v rámci projektu použito několik dalších knihoven. Pro autentizaci a správu uživatelů se využívá ASP.NET Identity, operace nad daty v databázi zastřešuje Entity Framework, mapování doménových objektů na view modely obstarává knihovna Automapper a závislosti mezi objekty spravuje Autofac kontejner. Prvotní konfigurace projektu a implementace zmíněných nástrojů je časově poměrně náročná, nicméně se určitě vyplatí. Proto je v závěrečné kapitole vše důkladně popsáno.

Front-end příkladové aplikace je vytvořen pomocí knihovny React. Tato knihovna je velmi oblíbená, vývojář má k dispozici kvalitní dokumentaci a téměř neomezené množství studijních materiálů. Díky tomu byla implementace výrazně jednodušší. Správa historie a navigace je řešena za použití knihovny React-router. Autor se také zabýval otázkou řízení stavu aplikace. Z tohoto důvodu se rozhodl zaimplementovat knihovnu Redux, která se ukázala jako velmi efektivní nástroj pro správu stavu aplikace.

Při vývoji front-endové i back-endové části aplikace je kladem velký důraz na správné strukturování projektu. Díky tomu je aplikace přehledná a samotný vývoj je pak rychlejší. Kvalitní architektura přináší řadu výhod, mezi které patří: oddělení zodpovědností, znovupoužitelnost segmentů aplikace, přehlednost, snadná udržitelnost, rychlejší chod aplikace, jednoduchá testovatelnost, méně redundantního kódu a další. Tyto aspekty jsou do velké míry pro vývoj aplikací klíčové.

7 Závěr a doporučení

Cílem této diplomové práce bylo představit dva různé přístupy pro tvorbu webových aplikací. Jedná se o vývoj klasických webových aplikací na platformě ASP.NET MVC, přičemž jsou webové stránky generované na serveru. Tento způsob je porovnán s moderní a stále populárnější formou vývoje takzvaných Single Page Application, kdy celou aplikaci tvoří jediná webová stránka. Její obsah je plněn na straně klienta za použití JavaScriptu. Tuto úlohu v rámci diplomové práce zastává JavaScriptová knihovna React, jež aktuálně patří k nejoblíbenějším a nejrychleji se rozšiřujícím. Cílem této práce není pouze představit zmiňované technologie, ale poskytnout čtenáři dostatečně rozsáhlé podklady, aby plně porozuměl problematice a byl schopen sám vybrat vhodnější řešení. Jednotlivé kapitoly jsou podepřeny příkladovými ukázkami zdrojových kódů a kódy z vlastní aplikace. Technologie jsou v práci představeny a současně jsou uvedeny užitečné rady a nejlepší praktiky, jakým způsobem technologie používat a jak správně strukturovat komplexnější řešení. Práce by měla sloužit jako průvodce a usnadnit čtenáři případný vývoj.

Tematicky je diplomová práce velmi rozsáhlá a bylo poměrně obtížné představit vše podstatné. Práce se zabývá specifikací frameworků .NET, ASP.NET a ASP.NET MVC. Dále je analyzována technologie Single Page Application, knihovna React iarchitektura MVC. V rámci představené aplikace je ukázáno použití jednotlivých technologií při vývoji klasických webových aplikací a Single page aplikací.

Architektura MVC je velmi zajímavá a snadná na implementaci. Její použití je poměrně běžné na úrovni středně velkých a větších firemních projektů. Samotné rozdělení MVC se avšak ukázalo jako nedostačující pro většinu aplikací a je záhodné jej rozšířit o další logické vrstvy (například datová, servisní apod.). S frameworkem ASP.NET MVC se velmi dobře pracuje, nabízí řadu vestavěných i externích nástrojů pro vývoj kvalitních serverových řešení. Například pro práci s databází je použit Entity Framework, jenž lze nahradit často používaným NHibernate. Za účelem autentizace, správy uživatelů a uživatelských rolí je v projektu aplikován framework ASP.NET Identity.

Analýza ukázala, že Single page aplikace nabírají rychle na popularitě a jsou označovány za budoucnost moderního webového vývoje. Z průzkumů je zřejmé, že počet jednostránkových aplikací výrazně narůstá. O SPA je velký zájem, proto se díky práci rozsáhlých komunit tyto technologie velmi rychle rozvíjejí. Vývojáři mají k dispozici téměř neomezené studijní materiály a nástroje, jež podstatně usnadňují práci, což je samo

o sobě nespornou výhodou. Proces studia může být zpočátku pomalejší, neboť je nutné zvládat práci s větším množstvím nástrojů a technologií, které se v této době extrémně rychle mění a vyvíjejí. To může být pro řadu vývojářů rozhodujícím kritériem. Pro vytvoření front-endové části příkladové aplikace si musel autor osvojit znalosti pro práci s nástroji jako je: Webpack, Node Package Manager, knihovny React, Redux, React-router, Material-UI a dalšími.

Single page aplikace přináší oproti klasickým webovým aplikacím řadu výhod. Jsou velmi rychlé, a poskytují tak uživateli plynulý a příjemný zážitek při jejich procházení. Veškeré zdrojové kódy nutné k chodu aplikace jsou staženy během inicializačního načtení stránek. Server při každém requestu neposílá kompletní, velké HTML včetně CSS, ale pouze data. Počet requestů na server je nejen snížen, ale především je omezen objem dat putujících zpět na stranu klienta. V důsledku toho je menší traffic a celková zátěž serveru. Single Page Application jsou snadno udržitelné. Díky oddělenému front-endu a back-endu aplikace je možné jednotlivé části vyvíjet a testovat separátně. Při vývoji FE lze BE dokonce nahradit specializovaným API, které simuluje BE a posílá klientovi falešná data. K těmto účelům může být využito například *Apiary*. Další výhodou separace je znovupoužitelnost back-end části aplikace – ten samý back-end je možné použít pro webovou i mobilní aplikaci. Single page aplikace mohou fungovat částečně v offline režimu, jelikož jsou potřebné zdroje staženy při prvním načtení stránky a uloženy lokálně.

Na druhou stranu single page aplikace přinášejí oproti klasickým, vícestránkovým aplikacím některá omezení. Pro jejich fungování je nutná podpora JavaScriptu. Klasické aplikace jsou obecně bezpečnější a méně náchylné ke Cross-Site Scripting útokům. Dříve sužoval SPA problém se SEO a spravováním historie. Tyto problémy již dnes nejsou aktuální a nedají se příliš pokládat za relevantní limitaci. Potíž může nastat v případě pomalého internetového připojení a tlusté klientské části. První vykreslení pak trvá poměrně dlouho, což může potenciální návštěvníky odradit, v krajním případě dokonce i znemožnit přístup na stránky.

Z výše uvedeného textu je patrné, že technologie Single Page Application přináší mnoho podstatných výhod, rychle se vyvíjí a její klady převažují. Nicméně i přes obsáhlý seznam benefitů tohoto přístupu nelze jednoznačně označit SPA za nejlepší možné řešení pro všechny druhy aplikací. Single page aplikace nabízí uživateli velmi moderní, rychlé a vysoce interaktivní prostředí. Oproti tomu klasické webové aplikace naopak přinášejí

standardní, pomalejší, ale velmi bezpečné řešení. Nutné je mít na paměti, že požadovanou aplikaci lze vždy vytvořit oběma způsoby. Při výběru vhodné technologie záleží na prioritách, požadavcích, účelu samotných webových stránek a know-how či zkušenostech vývojového týmu. Pokud má výsledná aplikace zaujmout návštěvníka a poskytnout příjemný a svižný zážitek, je SPA vhodnou volbou. V případě velmi rozsáhlého projektu, jenž má prioritu v bezpečnosti a kvalitní uživatelský zážitek není stěžejní, může být vhodnější varianta klasické webové aplikace. Faktorů, podle nichž je třeba vybírat vhodnou technologii je více a v tuto chvíli se stále názory vývojářů rozcházejí. Nicméně zkušenosti se Single page aplikacemi rostou a k tomuto řešení se postupně přiklání více a více vývojářů.

Autor zastává názor, že technologie Single Page Application má velký potenciál, tudíž se bude pravděpodobně tímto směrem ubírat vývoj moderních webových aplikací. To je důvodem, proč si tento proud vývoje zaslouží velkou pozornost a jeho znalost je prakticky nutností k udržení kroku s moderními technologiemi.

8 Zdroje

[1] Co je .NET framework [online]. 2010 [cit. 2016-06-22]. Dostupné z: http://svet-hostingu.cz/2009/10/12/co-je-net-framework/

[2] Pro C# 5.0 and the .NET 4.5 Framework. TROELSEN, Andrew. Pro C# 5.0 and the .NET 4.5 Framework. 6th Edition. Calif: Apress, 2012, ISBN 978-1-4302-4234-5.

[3] MEHRA, Puran. .NET Framework and Architecture [online]. 2010 [cit. 2016-06-22]. Dostupné z: http://www.c-sharpcorner.com/UploadFile/puranindia/net-framework-and-architecture/

[4] *Under the Hood of .NET Memory Management*. HARRISON, Nick a Chris FARRELL. Simple Talk Publishing, 2011, s. 28-29. ISBN 978-1-906434-74-8.

[5] HANSELMAN, Scott. *An update on ASP.NET Core 1.0 RC2* [online]. **2016** [cit. 2016-06-22]. Dostupné z: http://www.hanselman.com/blog/AnUpdateOnASPNETCore10RC2.aspx

[6] .NET Framework: Architecture. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2016 [cit. 2016-06-22]. Dostupné z: https://en.wikipedia.org/wiki/.NET_Framework

[7] Common Language Specification. *MSDN* [online]. [cit. 2016-06-22]. Dostupné z: https://msdn.microsoft.com/en-us/library/12a7a7h3(v=vs.100).aspx

[8] GARG, Vikas. *.Net Architecture and .Net Framework Basics* [online]. 2011 [cit. 2016-06-22]. Dostupné z: http://www.c-sharpcorner.com/uploadfile/09f663/net-architecture-and-net-framework-basics/

[9] MACDONALD, Mathew. *Beggining ASP.NET 4.5 in C#*. Calif: Appres, 2012, s 8-13. ISBN 978-1-4302-4252-9.

[10] NAGEL, Christian, Bill EVJEN, Jay GLYNN, Karli WATSON a Morgan SKINNER. *Professional C# 4 and .Net 4*. Indiana: Wiley Publishing, 2010. ISBN 978-0-470-50225-9.

[11] *ITnetwork: 1. díl – Úvod do C# a .NET frameworku* [online]. 2014 [cit. 2016-06-27]. Dostupné z: http://www.itnetwork.cz/csharp/zaklady/c-sharp-tutorial-uvod-do-jazyka-a-dot-net-framework

[12] TROELSEN, Andrew a Philip JAPIKSE. *C# 6.0 and the .NET 4.6 Framework*. Seventh edition. New Youk: Apress, 2015, s. 64-66. ISBN 978-1-4842-1333-9.

[13] .NET Framework version history. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2016-07-06]. Dostupné z: https://en.wikipedia.org/wiki/.NET_Framework_version_history#.NET_Framework_3.5

[14] *Historie .NETu* [online]. 2014 [cit. 2016-06-27]. Dostupné z: http://www.itnetwork.cz/csharp/historie-dotnetu

[15] Verze a závislosti rozhraní .NET Framework. *Microsoft MSDN* [online]. 2015 [cit. 2016-07-06]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/bb822049(v=vs.110).aspx

[16] SHALEINDRA, Chauhan. Understanding Detailed Architecture of ASP.NET 4.5: Components of Asp.NET 4.5 Architecture. In: DotNetTricks [online]. 2013 [cit. 2016-07-10]. Dostupné z: http://www.dotnet-tricks.com/Tutorial/aspnet/SaJc221013-Understanding-Detailed-Architecture-of-ASP.NET-4.5.html

[17] AJAX Introduction. W3Schools [online]. [cit. 2016-07-10]. Dostupné z: http://www.w3schools.com/ajax/ajax_intro.asp

[18] - MIKOWSKI, Michael S. a Josh C. POWELL. Single Page Web Application: JavaScript end-to-end. Shelter Island, NY 11964: Manning Publications Co., 2014, s. 1-9. ISBN 9781617290756. http://choonsiong.com/public/books/tmp/Single%20Page%20Web%20Application.pdf

[19] - Essential Ajax. HOLZNER, Steven. *AJAX: A Beginner's Guider*. New York: The McGraw-Hill Companies, 2009, s. 1-7. ISBN 0-07-159531-7. https://thesharad.files.wordpress.com/2010/10/ajax-a-beginners-guide.pdf

[20] - The Road to Single Page Application Development: Introducing Single Page Applications. FINK, Gil a Ido FLATOW. *Pro Single Page Application Development:*

Using Backbone.js and ASP.NET. New York, 233 Spring Street: Apress, 2014, s. 1-13. ISBN 9781430266730. http://pepa.holla.cz/wp-content/uploads/2015/10/Pro-Single-Page-Application-Development.pdf

[21] - *Standard ECMA-262: ECMAScript® 2017 Language Specification*. 8th edition. Geneva: ECMA International, 2016.

[22] - Node.js. HAVERBEKE, Marijn. *Eloquent JavaScript: A Modern Introduction to Programming*. Creative Commons, 2014, s. 376-377. ISBN 978-1593275846.

[23] - Node fundamentals: Welocome to Node.js. CANTELON, Mike, Marc HARTER, T.J. HOLOWYCHUK a Nathan RAJLICH. Node.js in action. New York: Manning Publications Co., 2014, s. 3-5. ISBN 9781617290572.

[24] - Tutorialspoint.com: node.js -npm [online]. 2017 [cit. 2017-07-11]. Dostupné z: https://www.tutorialspoint.com/nodejs/nodejs_npm.htm

[25] - NPM: What is npm? Npmjs.js [online]. 2017 [cit. 2017-07-11]. Dostupné z: https://docs.npmjs.com/getting-started/what-is-npm

[26] - Webpack.js [online]. 2017 [cit. 2017-07-11]. Dostupné z: https://webpack.js.org/concepts/

[27] - FEDOSEJEV, Artemij. React.js Essentials. Birmingham: Packt Publishing, 2015. ISBN 978-1-78355-162-0.

[28] - Components, Props and State. React VR | A framework for building VR apps using React [online]. 2017 [cit. 2017-07-25]. Dostupné z: https://facebook.github.io/react-vr/docs/components-props-and-state.html

[29] - React.Component. React – A JavaScript library for building user interfaces [online]. 2017 [cit. 2017-07-25]. Dostupné z: https://facebook.github.io/react/docs/react-component.html

[30] - In Depth Overview. Flux | Application Architecture for Building User Interfaces [online]. 2015 [cit. 2017-07-25]. Dostupné z: https://facebook.github.io/flux/docs/in-depth-overview.html#content

[31] - Configuring ASP.NET Web API 2. Technical documentation, API, and code examples | Microsoft Docs [online]. 2014 [cit. 2017-07-29]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/web-api/overview/advanced/configuring-aspnet-web-api

[32] - MULLOY, Brian. Web Api Design: Crafting Interfaces that Developers Love [online]. Apigee, 2012 [cit. 2017-07-30]. Dostupné z: https://pages.apigee.com/rs/apigee/images/api-design-ebook-2012-03.pdf

[33] - Official Google Webmaster Central Blog: Deprecating our AJAX crawling scheme. Official Google Webmaster Central Blog[online]. 2015 [cit. 2017-08-06]. Dostupné z: https://webmasters.googleblog.com/2015/10/deprecating-our-ajax-crawling-scheme.html

[34] - JavaScript Versions. W3Schools Online Web Tutorials [online]. [cit. 2017-08-06]. Dostupné z: https://www.w3schools.com/js/js_versions.asp

[35] - GALLOWAY, Jon, Brad WILSON, K.Scott ALLEN a David MATSON. Professional ASP.NET MVC 5. Canada: John Wiley, 2014. ISBN 978-1-118-79475-3.

[36] - DbContext. Entity Framework Tutorial [online]. 2016 [cit. 2017-09-12]. Dostupné z: http://www.entityframeworktutorial.net/EntityFramework4.3/dbcontext-vs-objectcontext.aspx

[37] - Language Integrated Query (LINQ). Technical documentation, API, and code examples | Microsoft Docs [online]. [cit. 2017-09-16]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/

[38] - ASP.NET MVC Overview. Learn to Develop with Microsoft Developer Network | MSDN [online]. [cit. 2017-09-16]. Dostupné z: https://msdn.microsoft.com/en-us/library/dd381412(v=vs.108).aspx

[39] - ASP.NET - Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET. Learn to Develop with Microsoft Developer Network | MSDN [online]. 2013 [cit. 2017-09-16]. Dostupné z: https://msdn.microsoft.com/en-us/magazine/dn463786.aspx?f=255&MSPPError=-2147217396

[40] - GULZAR, Nadir. Fast Track to Struts: What it Does and How [online]. In: . 2002 [cit. 2017-09-17]. Dostupné z: http://media.techtarget.com/tss/static/articles/content/StrutsFastTrack/StrutsFastTrack.pdf

[41] - ASP.NET Web Configuration Guidelines. Learn to Develop with Microsoft Developer Network | MSDN [online]. [cit. 2017-11-01]. Dostupné z: https://msdn.microsoft.com/en-us/library/ff400235.aspx

[42] - Global.asax File. Learn to Develop with Microsoft Developer Network | MSDN [online]. [cit. 2017-11-01]. Dostupné z: https://msdn.microsoft.com/en-us/library/1xaas8a2(v=vs.71).aspx

[43] - PEYROTT, Sebastián. The JWT Handbook [online]. AuthO, 2016 [cit. 2017-11-03]. Dostupné z: http://pepa.holla.cz/wp-content/uploads/2017/06/jwt-handbook.pdf

[44] - SAHAY, Rauhl. Hands-on with ASP.NET MVC: Covering MVC 6. Quills Ink Publishing, 2014. ISBN 978-93-84318-52-9.

9 Zadání diplomové práce

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2015/2016

Studijní program: Systémové inženýrství a informatika
Forma: Prezenční
Obor/komb.: Informační management (im2-p)

Podklad pro zadání DIPLOMOVÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Be. Šulc Přemysl	O. Zeminy 73, Nová Paka - Heřmanice	I1450

TÉMA ČESKY:

Vývoj webových aplikací na platformě ASP.NET MVC a Single Page Application.

TÉMA ANGLICKY:

Development of web applications on ASP.NET MVC platform and Single Page Application

VEDOUCÍ PRÁCE:

Ing. Jiří Štěpánek - KIT

ZÁSADY PRO VYPRACOVÁNÍ:

Cíl práce:
Analyzovat a porovnat vývoj klasických webových aplikací na platformě ASP.NET MVC s vývojem Single Page Aplikací. Shrnout principy obou technologií a nastínit jejich klady a zápory. Předvést jejich implementaci na konkrétních příkladech.

Osnova práce:
1) Úvod
2) Framework .NET
3) Framework ASP.NET MVC
4) Single Page Application
5) React
6) Praktické použití probíraných technologií
7) Závěr

SEZNAM DOPORUČENÉ LITERATURY:

Andrew Troelsen - Pro C# 5.0 and the .NET 4.5 Framework
Harrison & Farrel - Under the Hood of .NET Memory Management
Mathew MacDonald - ASP.NET 4.5 in C#
Chauhan Shaleindra - Understanding Detailed Architecture of ASP.NET 4.5
Rauhl Sahay - Hands-on with ASP.NET MVC: Covering MVC 6
Galloway, Wilson, Scott, Matson - Professional ASP.NET MVC 5
Mikowski & Powell - Single Page Web Application: JavaScript end-to-end
Fink & Flatow - The Road to Single Page Application Development

Podpis studenta: P. Šulc Datum: 12.10.2015

Podpis vedoucího práce: Štěpánek Datum: 12.10.2015