Možnosti využití protokolu OData v aplikační platformě .NET

Bc. Milan Gatyás

Diplomová práce
2015

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2014/2015

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Milan Gatyás**
Osobní číslo:	**A13478**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Forma studia:	**kombinovaná**
Téma práce:	**Možnosti využití protokolu OData v aplikační platformě .NET**
Téma anglicky:	**The Possibilities of Using OData Protocols in a .NET Platform and for Developing Business Applications**

Zásady pro vypracování:

1. **Zpracujte literární rešerši na téma Service-Oriented Architecture (SOA).**
2. **Popište OData protokol.**
3. **Analyzujte možnosti poskytování a konzumace OData služeb.**
4. **S pomocí případových studií srovnejte OData služby s SQL přístupem z hlediska použitelnosti, výkonnosti a složitosti vývoje.**
5. **Demonstrujte výsledky a formulujte závěr.**

Rozsah diplomové práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **ERL, Thomas. SOA: servisně orientovaná architektura : kompletní průvodce. Vyd. 1. Překlad Ondřej Baše, Lukáš Krejčí. Brno: Computer Press, 2009, 671 s. ISBN 978-80-251-1886-3.**
2. **OData - The Protocol for REST APIs. [online]. [cit. 2015-01-24]. Dostupné z: http://www.odata.org/**
3. **CHENG, Steven. OData programming cookbook for .NET developers: 70 fast-track, example-driven recipes with clear instructions and details for OData programming with .NET framework. New Edition. Birmingham: Packt Pub, 2012. ISBN 18-496-8592-4.**
4. **MUELLER, John Paul. Microsoft ADO.NET entity framework step by step. Sebastopol (Calif.): O'Reilly Media, 2013. ISBN 0735664161.**
5. **LAKSHMIRAGHAVAN, Badrinarayanan. Practical asp.net web api. Berkeley: Apress, 2013. ISBN 14-302-6175-7.**
6. **OData Support In ASP.NET Web Api. [online]. [cit. 2015-01-24]. Dostupné z: http://www.asp.net/web-api/overview/odata-support-in-aspnet-web-api**
7. **SCOTT SHAW, Kathi Kellenberger. Beginning T-SQL 2012. 2nd ed. New York, N.Y.: Apress, 2012. ISBN 14-302-3704-X.**

Vedoucí diplomové práce: **Ing. Erik Král, Ph.D.**
Ústav počítačových a komunikačních systémů

Datum zadání diplomové práce: **6. února 2015**

Termín odevzdání diplomové práce: **15. května 2015**

Ve Zlíně dne 6. února 2015

L.S.

doc. Mgr. Milan Adámek, Ph.D.
děkan

doc. Mgr. Roman Jašek, Ph.D.
ředitel ústavu

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové/bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně

.....
podpis diplomanta

ABSTRAKT

Diplomová práce se zabývá možnostmi využití Open Data protokolu v aplikačním prostředí .NET. V teoretické části jsou popsány obecné principy servisně orientované architektury, dále je popsán protokol OData, možnosti poskytování a konzumování OData služeb, a možnosti zabezpečení OData služeb. V praktické části jsou uvedeny případové studie pro OData službu, která je implementována pomocí technologie ASP.NET Web Api OData, a případové studie pro implementace klienta služby na platformě .NET pomocí nástroje ODat v4 Client Code Generator.

Klíčová slova: SOA, REST, OData, ASP.NET Web Api OData, OData v4 Client Code Generator.

ABSTRACT

This thesis is concerned about possibilities of usage Open Data protocol on .NET platform. Theoretical part describes general principles of service oriented architecture, OData protocol structure, OData services providing possibilities, OData services consumption possibilities, and OData services security possibilities. Practical part contains case studies in OData services implementation via ASP.NET Web Api OData framework and case studies in OData services consumption on .NET platform via OData v4 Client Code Generator tool.

Keywords: SOA, REST, OData, ASP.NET Web Api OData, OData v4 Client Code Generator.

Děkuji svému vedoucímu diplomové práce Ing. et Ing. Eriku Královi, Ph.D. za jeho cenné rady při vypracovávání této práce.

OBSAH

ÚVOD

Výměna dat je nedílnou součástí aplikací. V praxi je běžné, že aplikace poskytují pro práci s daty své vlastní rozhraní, se kterým se klient musí seznámit, a až poté ho může začít využívat. Pokud klientská aplikace funguje např. jako agregátor různých zdrojů, musí pro každý zdroj implementovat unikátního klienta.

Takový přístup k poskytování a konzumaci dat není ovšem vhodný, jelikož konzument dat ztrácí čas a prostředky, které musí vynaložit pro seznámení se s datovým rozhraním. Rovněž producent musí vynaložit úsilí pro specifikaci infrastruktury poskytující data. Přirozenější variantou je definovat datový zdroj a jeho klienta pomocí jednoznačných, dobře známých, a interoperabilních pravidel.

Na základě standardizovaně definovaného datového zdroje je potom možné zjednodušit vlastní zpracování dat. Z popisu zdroje je možné vygenerovat značnou část klientské infrastruktury, a vývojář poté využívá pouze toto vygenerované klientské rozhraní.

Diplomová práce pojednává o jedné z možností, jak poskytovat a konzumovat data standardizovanou a interoperabilní cestou. Touto cestou je využití protokolu OData.

I. TEORETICKÁ ČÁST

1 SERVISNĚ ORIENTOVANÁ ARCHITEKTURA

Servisně orientovaná architektura (SOA) je model, ve kterém je logika automatizace rozdělena na *logické jednotky*. Dohromady tyto jednotky tvoří vyšší celky. Individuálně mohou být tyto jednotky rozdělené. SOA podporuje, aby jednotlivé jednotky existovaly autonomně, ale nebyly od sebe izolovány. Musí dodržovat určitou sadu předpisů, aby mezi sebou mohly komunikovat. Cílem je nalézt takovou sadu předpisů, která by umožnila jednotlivým jednotkám rozvíjet se nezávisle, a zároveň zajistila dostatečné množství společných rysů a standardizace. V SOA jsou tyto logické jednotky známé jako *služby*.

Služby mohou používat jiné služby nebo programy. Vztah mezi službami je založen na tom, že služby o sobě musí navzájem vědět. K tomuto účelu slouží *popisy služeb*. Popis stanovuje především název služby, umístění služby, a její požadavky na výměnu dat. Výměna dat mezi službami je zprostředkována pomocí výměny *zpráv*.

Mezi hlavní charakteristiky servisní orientace tedy patří:

- *Volná vazba* – minimalizace závislosti mezi jednotlivými službami.
- *Kontrakt služby* – popis služby jednoznačně určuje pravidla pro komunikaci.
- *Samospráva* – kontrola nad logikou, kterou služba zapouzdřuje.
- *Abstrakce* – zapouzdření vnitřní logiky vůči vnějšímu prostředí.
- *Znuvupoužitelnost* – logika problémů je dělena do logických jednotek (služeb) za účelem podpory znuvupoužitelnosti.
- *Kompozice* – jednotlivé služby lze komponovat do vyšších celků.
- *Bezstavovost* – minimalizace službou uchovávaných informací pro provedení určité aktivity.
- *Zjistitelnost* – službu je možné vyhledat a zhodnotit dostupným vyhledávacím mechanismem.

Termín servisně orientovaná služba existoval již před příchodem webových služeb. Jsou to však právě webové služby, které jsou pro implementace servisně orientovaných služeb používány nejčastěji [1].

1.1 WSDL a SOAP

Webové služby implementované s pomocí technologií *WSDL* (Web Services Description Language) a *SOAP* (Simple Object Access Protocol) jsou v současné době asi nejvíce rozšířený způsob poskytování webových služeb.

Tyto služby využívají pro svůj popis jazyk *XML*[1] (EXtensible Markup Language). Služby definují standardní mechanismus pro zpřístupňování a konzumaci dat pomocí výměny XML zpráv. Tyto zprávy mohou být přenášeny různými cestami, např. skrze *TCP* (Transmission Control Protocol), *HTTP* (HyperText Transfer Protocol), nebo *SMTP* (Simple Mail Transfer Protocol).

Pro umožnění přenosu zpráv mezi poskytovatelem a konzumentem dat je potřebné, aby se konzument seznámil s rozhraním poskytovatele. K tomuto účelu slouží WSDL dokument. WSDL dokument sdružuje zprávy do operací, a operace do rozhraní. Zároveň uvádí vazby mezi rozhraními, požadovaným protokolem, a koncovou adresou pro dané rozhraní. Kompletní WSDL dokument poskytuje všechny informace, jež jsou potřebné pro oslovení webové služby. Poskytuje tzv. *kontrakt* služby.

Kromě definice kontraktu služby je potřebná ještě definice pro zprávy, pomocí kterých jsou přenášena vlastní data. Tuto definici poskytuje SOAP. SOAP využívá jazyka XML pro definici rozšiřitelného komunikačního rozhraní, které může být implementováno nad různými druhy podkladových přenosových infrastruktur. SOAP byl navržen tak, aby nebyl závislý na žádném programovacím modelu. Díky tomu je interoperabilní [2][3].

1.2 REST webové služby

REST (Representational State Transfer) Je architektonický styl, který definoval Roy Fielding ve své disertační práci. Tento styl definuje především následující omezení:

- *Oddělení zodpovědností* (Separation of concerns) – oddělení zodpovědnosti pro rozhraní konzumenta dat od zodpovědností mechanismu úschovy dat.
- *Bezstavovost* – každý požadavek na poskytovatele dat musí obsahovat všechny potřebné informace pro vyřízení požadavku.

[1] Pro více informací o jazyku XML viz http://www.w3.org/TR/REC-xml/

- *Kešování* – při opakovaném požadavku na stejný datový zdroj je možné konzumentovi vrátit data z kešovací paměti.
- *Stejnorodé rozhraní* – všechny komponenty mají stejné rozhraní pro komunikaci.

Tato omezení splňuje například protokol HTTP, který je nejpoužívanějším prostředkem při implementaci stylu REST. Stěžejným požadavkem na tento styl je důraz na stejnorodé rozhraní, díky kterému je zpracování datového zdroje jednodušší a přehlednější. V případě HTTP to znamená využívání HTTP metod (GET, POST, PUT, …).

Webové služby, které se řídí stylem REST, se nazývají REST webové služby (RESTful Web Services). REST webové služby jsou datově orientované (Resource-Oriented). Ústředním motivem zde nejsou metody služby, nýbrž jednotlivé datové entity služby [4][5].

1.3 WSDL+ SOAP nebo REST přístup

Pomocí WSDL jsou zveřejňovány specifické operace, jenž se tvůrce služby rozhodl zveřejnit. Vlastní struktura a vazby mezi daty jsou klientovi skryté. Tento styl se nazývá *RPC* (Remote Procedure Call). Službu tohoto typu je vhodné použít, pokud je potřebné zveřejnit pouze pevně dané funkcionality, a interní tvar dat by měl být zapouzdřen. Takovéto chování nemusí být zcela vhodné, pokud je potřebné poskytovat *datově orientovanou službu*. Pro každou entitu je totiž potřebné definovat sadu metod, které entitu zveřejní klientovi. Rovněž je obtížné vyhovět všem požadavkům klienta, jelikož např. může chtít vyhledávat lokace namísto názvu pomocí zeměpisných souřadnic.

V případě datově centrických služeb je vhodné použít principu REST. REST je vhodný především pro provádění *CRUD* (Create, Read, Update, Delete) operací nad entitami. Standardním schématem je situace, kdy jsou jednotlivé entity k dispozici dle jedinečných *URL* (Uniform Resource Locators), a jednotlivé operace jsou prováděny pomocí HTTP metod. Klient v tomto případě vidí strukturu entit, a rovněž má k dispozici předem danou sadu operací, které může nad těmito entitami provádět. Nemusí tedy studovat metody služby. REST webové služby obecně nedeklarují metadata podobné WSDL dokumentu, a klienta tudíž nelze automaticky vygenerovat. Rovněž není možné bez dokumentace (nebo znalosti služby) využívat jiné, než CRUD operace [6].

Pokud je vhodné využít REST webovou službu, ale rovněž je potřebné generovat metadata, musí se použít protokol, který poskytne dodatečné funkcionality. Tímto protokolem je protokol *OData* (Open Data Protocol).

2 PROTOKOL ODATA

OData je aplikační protokol pro tvorbu REST webových služeb. Protokol navrhla firma Microsoft, která ho vyvíjela do verze 3. V současné době je spravován organizací OASIS[2], která uvolnila specifikaci pro verzi 4. Verze 1 až 3 jsou zpětně kompatibilní, verze 4 již nikoliv. V textu bude popsána verze protokolu 4.

2.1 Struktura protokolu

Poskytovaná infrastruktura a data jsou popsána pomocí *EDM* (Entity Data Model). EDM deklaruje především následující prvky:

- *Kolekce entit* (Entity set), které definují přístupové body k entitám.
- *Entity*, jež představují poskytované typy dat. Jsou zde popsány vlastnosti entit a jejich datové typy, možnost absence hodnoty, klíč entity, apod.
- *Navigační vlastnosti* (Navigation property), které deklarují vztahy mezi entitami (na základě cizích klíčů) a jejich kardinalitu.
- *Akce*, pomocí kterých je možné provádět operace, které manipulují s daty.
- *Funkce*, které poskytují operace, jež data zpřístupňují, ale nemění je.

Komunikace se službou probíhá na základě jedinečných kanonických identifikátorů, jež reprezentují konkrétní prvky služby. Identifikátory jsou ve formě URL.

Protokol poskytuje univerzální popis infrastruktury služby a vlastních dat. Díky tomu je platformě nezávislý a umožňuje vysokou interoperabilitu. Poskytovaná data jsou ve formátu JSON (JavaScript Object Notation) nebo Atom [7][8].

2.1.1 CSDL

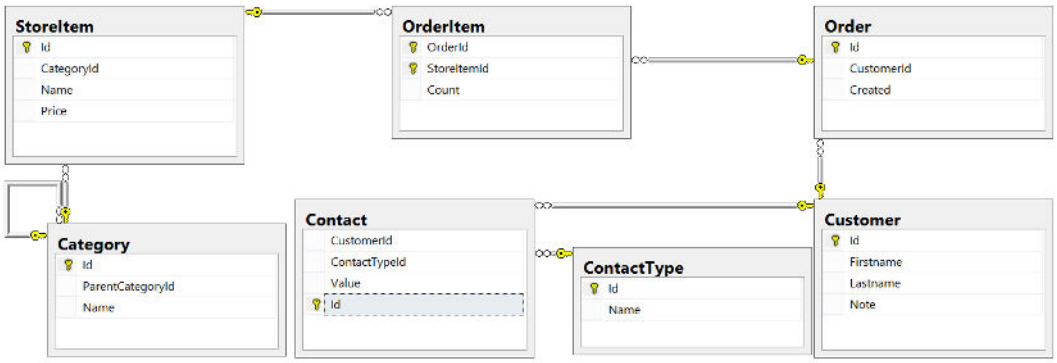
EDM je reprezentován pomocí *CSDL* (Common Schema Definition Language) dokumentu. Tento dokument je k dispozici pod adresou *[adr]/$metadata*. Řetězec *[adr]* bude v textu reprezentovat adresu kořene služby. Vzorový dokument by mohl mít podobu

[2] https://www.oasis-open.org/

```
<edmx:Edmx xmlns:edmx="http://docs.oasis-open.org/odata/ns/edmx"
    Version="4.0">
<edmx:DataServices>
<Schema xmlns="http://docs.oasis-open.org/odata/ns/edm"
    Namespace="Server.Edm">
    <EntityType Name="OrderItem">
        <Key>
            <PropertyRef Name="OrderId"/>
            <PropertyRef Name="StoreItemId"/>
        </Key>
        <Property Name="OrderId" Type="Edm.Int32" Nullable="false"/>
        <Property Name="StoreItemId" Type="Edm.String" Nullable="false"/>
        <Property Name="Count" Type="Edm.Int32" Nullable="false"/>
        <NavigationProperty Name="Order" Type="Server.Edm.Order"/>
        <NavigationProperty Name="StoreItem" Type="Server.Edm.StoreItem"/>
    </EntityType>
    <EntityType Name="StoreItem">
        <Key>
            <PropertyRef Name="Id"/>
        </Key>
        <Property Name="Id" Type="Edm.String" Nullable="false"/>
        <Property Name="CategoryId" Type="Edm.Int32" Nullable="false"/>
        <Property Name="Name" Type="Edm.String" Nullable="false"/>
        <Property Name="Price" Type="Edm.Decimal" Nullable="false"/>
        <NavigationProperty Name="Category" Type="Server.Edm.Category"/>
        <NavigationProperty Name="OrderItems"
            Type="Collection(Server.Edm.OrderItem)"/>
    </EntityType>
</Schema>
<Schema xmlns="http://docs.oasis-open.org/odata/ns/edm" Namespace="Default">
    <EntityContainer Name="Container">
        <EntitySet Name="OrderItems" EntityType="Server.Edm.OrderItem">
            <NavigationPropertyBinding Path="Order" Target="Orders"/>
            <NavigationPropertyBinding Path="StoreItem" Target="StoreItems"/>
        </EntitySet>
        <EntitySet Name="StoreItems" EntityType="Server.Edm.StoreItem">
            <NavigationPropertyBinding Path="Category" Target="Categories"/>
            <NavigationPropertyBinding Path="OrderItems"
Target="OrderItems"/>
        </EntitySet>
    </EntityContainer>
</Schema>
</edmx:DataServices>
</edmx:Edmx>
```

ER diagram datového zdroje viz obrázek 1. CSDL dokument je XML dokument, jehož kořenem je element *Edmx*. Atribut *Version* určuje verzi datové služby.

V elementu *DataServices* se nachází element *Schema*. Obsahuje typy a strukturu poskytovaných dat. Rovněž poskytuje informace o poskytovaných kolekcích. Činí tak pomocí *EntitySet* elementů. Dále jsou uvedeny vztahy mezi kolekcemi pomocí elementů *NavigationPropertyBinding*. Pro přehlednost byly uvedeny pouze dvě tabulky. Výpis celého dokumentu ukázkové služby viz příloha PI [9][10].



Obrázek 1 ER diagram datového zdroje služby

2.1.2 Datové typy

Kromě entit služba může deklarovat i jiné datové typy. Entity jsou součástí skupiny *strukturních typů*. Spolu s entitami do této kategorie patří *komplexní typy*. Strukturní typy obsahují *strukturní vlastnosti*, které jsou typu *primitivního*, *komplexního*, nebo *enumerativního*.

Primitivní typy jsou dále nedělitelné, výpis typů viz tabulka 1.

Binary	Boolean	Byte	Date
DateTimeOffset	Decimal	Double	Duration
Guid	Int16	Int32	Int64
SByte	Single	Stream	String
TimeOfDay	*Geography	*Geometry	

Tabulka 1 Primitivní datové typy [11]

Typy označené hvězdičkou určují specifickou rodinu datových typů pro práci s geografickými, nebo geometrickými daty. Za zmínku stojí *absence* datového typu *DateTime*. Protokol tento datový typ *nepodporuje*, a doporučuje místo něj používat datový typ *DateTimeOffset*. To může způsobit nepříjemnosti především u již existujících databází, které obsahují sloupce s tímto datovým typem. Toto omezení se nevztahuje na verze protokolu 1 až 3. Zde je primitivní typ DateTime podporován [11].

Strukturní vlastnosti popisují strukturní typy. Uvádějí se elementem *Property*. Vlastní informace sdělují pomocí atributů. Přehled nejdůležitějších atributů viz tabulka 2.

Atribut	Význam
Name	Název vlastnosti
Type	Typ vlastnosti
Nullable	Možnost absence hodnoty
*MaxLength	Maximální délka proudu, řetězce, nebo binárního řetězce
*Precision	Přesnost číselného nebo datového typu
*DefaultValue	Implicitní hodnota

Tabulka 2 Vybrané atributy strukturní vlastnosti [12]

Atributy s hvězdičkou nejsou povinné. Služba je nemusí uvádět, i když jsou v podkladovém zdroji definovány. [12].

Navigační vlastnosti popisují vztahy mezi entitami. Uvádějí se pomocí elementu *NavigationProperty*. Přehled vybraných atributů viz tabulka 3.

Atribut	Význam
Name	Název vlastnosti
Type	Typ svázané entity
Nullable	Možnost absence hodnoty
*OnDelete	Operace nad kolekcí podřízených entit při odstranění nadřízené entity

Tabulka 3 Vybrané atributy navigační vlastnosti [13]

Atribut označený hvězdičkou nemusí být službou podporován. Atribut *Type* může mít tvar entity, nebo kolekce entit. Tvar entity je použit, pokud je svázaná entita *nadřízená*. V ukázkové službě je to např. navigační vlastnost *Order* entity *OrderItem*. Kolekce entit se naopak použije, je-li svázaná entita *podřízená*. V ukázce vlastnost *OrderItems* entity *StoreItem*. Atribut *Nullable* je uvažován pouze u vlastností, jež mají typ entity. Pokud není přítomen, považuje se jeho hodnota jako kladná, a svázaná entita tudíž nemusí existovat. Kolekce entit bude vždy nenulová. Může být ovšem prázdná [13].

Entity jsou strukturní typy, jež reprezentují *jedinečně identifikovatelné prvky*. Pomocí entit se tedy modelují entity podkladového datového zdroje. Uvádějí se elementem *EntityType*. Entity obsahují navigační a strukturní vlastnosti. Obsahují rovněž element *Key*, který urču-

je alespoň jednu referenci na vlastnost-i tvořící klíč entity. Vlastní element obsahuje atributy s dalšími informacemi. Jejich výběr viz tabulka 4.

Atribut	Význam
Name	Název entity
BaseType	Předek entity
Abstract	Zda je entita abstraktní
HasStream	Zda je entita multimediální
*OpenType	Zda je entita otevřená

Tabulka 4 Vybrané atributy entity [14]

Jak je z atributu *BaseType* patrné, protokol definuje dědičnost entit. Potomek dědí klíč, strukturní a navigační vlastnosti předka. Atribut se neuvádí, pokud entita nemá předka. Entita může být rovněž abstraktní. V takovém případě má atribut *Abstract* nastaven na hodnotu *true*, a neobsahuje element *Key*. Při absenci atributu není entita pokládána za abstraktní. Atribut *HasStream* určuje, zda entita obsahuje vlastnost multimediálního typu. Atribut *OpenType* nemusí být službou podporovaný. Otevřené entity mohou obsahovat dynamické vlastnosti, jež nejsou uvedeny v CSDL dokumentu. Jedná se o dvojice klíč hodnota, kde klíče jsou unikátní. Při absenci atributu je entita považována za uzavřenou [14].

Komplexní typy jsou strukturní typy, které *nejsou jedinečně identifikovatelné*. Díky tomu nad nimi nemohou být samostatně prováděné CRUD operace. Uvádějí se elementem *ComplexType*. Obsahují strukturní vlastnosti primitivních, komplexních nebo enumerativních typů (případně jejich kolekce). Tyto typy mohou být součástí entity. V takovém případě jsou její strukturní vlastností. Rovněž se mohou použít jako argument, potažmo návratové hodnoty funkcí a akcí. Komplexní typy mohou dědit, mohou být abstraktní, a mohou být otevřené [15]. Vzorový element může mít podobu

```
<ComplexType Name="AddOrderItemModel">
    <Property Name="orderId" Type="Edm.Int32" Nullable="false"/>
    <Property Name="orderItemId" Type="Edm.Int32" Nullable="false"/>
</ComplexType>
```

Enumerativní typy jsou typy reprezentující výčet *předem definovaných hodnot*. Deklarují se elementem *EnumType*. Tento element může obsahovat logický atribut *IsFlags*, jenž sig-

nalizuje, že strukturní vlastnost může obsahovat více hodnot enumerativního typu současně. Jednotlivé výčtové položky jsou uvedeny jako vnořené elementy s názvem *Member*. Tyto elementy obsahují atribut *Name*, případně atribut *Value* [16]. Vzorový typ může mít podobu

```
<EnumType Name="PaymentMethod">
  <Member Name="Cash" />
  <Member Name="CreditCard" />
</EnumType>
```

Akce jsou uživatelsky definované operace, které mohou modifikovat datový zdroj služby. Uvádí se elementem *Action.* Funkce jsou operace, jejichž úkolem je vracet data. Uvádí se elementem *Function*. Atributem *Name* je určen název operace. Vnořeným elementem *ReturnType* je možné uvést návratovou hodnotu operace. Operace může vracet primitivní, nebo strukturní typ (případně jejich kolekce). Vnořeným elementem *Parameter* se deklarují parametry operace. Operace mohou být *vázané*, nebo *nevázané*. Vázané operace obsahují vazbu na určitou entitu, potažmo její kolekci. V takovém případě se volají jako metody dané entity/kolekce. První parametr vázané operace je její vazba. Nevázané operace jsou volány jako statické funkce z kořenové adresy služby. Vazba operace se deklaruje logickým atributem *IsBound* [17]. Vzorové operace by mohli mít podobu

```
<Function Name="GetTotalCost" IsBound="true">
    <Parameter Name="bindingParameter" Type="Collection(Server.Edm.Order)"/>
    <Parameter Name="orderId" Type="Edm.Int32" Nullable="false"/>
    <ReturnType Type="Edm.Double" Nullable="false"/>
</Function>
<Action Name="AddOrderItem" IsBound="true">
    <Parameter Name="bindingParameter" Type="Collection(Server.Edm.Order)"/>
    <Parameter Name="addOrderItemModel" Type="Server.Edm.AddOrderItemModel"/>
</Action>
```

Jak je z ukázky patrné, operace jsou vázané na kolekci typu *Order*.

Kromě zmíněných typů je dále možné využít typ *singleton*. Uvádí se elementem *Singleton*. Tento typ byl definován v protokolu verze 4. Singleton je prvek, který je jediný svého druhu pro celou službu. Prvek nelze vytvářet, ani odstranit. Lze k němu pouze přistupovat. Deklarace v CSDL souboru by mohla mít podobu

```
<Singleton Name="Contoso" Type="Self.Supplier">
    <NavigationPropertyBinding Path="Products" Target="Products" />
</Singleton>
```

Jak je z ukázky patrné, může obsahovat jak strukturní vlastnosti, tak navigační vlastnosti [18].

2.2 Využívání služby

Komunikace se službou probíhá skrze HTTP protokol. Jednotlivé funkcionality služby se využívají kombinací jedinečné URL adresy a příslušné HTTP metody.

2.2.1 Vytváření dat

Vytváření entit se provádí pomocí metody POST. Při úspěšné operaci služba vrátí kód 201 nebo 204. Při vytváření entity se v těle požadavku odešle struktura entity dle metadat služby. Požadavek pro vytvoření entity typu *ContactType* by například mohl mít tvar

```
POST [adr]/odata/ContactTypes
```

s tělem

```
{"@odata.type":"#Server.Edm.ContactType","Id":0,"Name":"Email"}
```

Při vytváření entity je možné uvést i navigační vlastnosti na existující entity. Příkladem může být URL ve tvaru

```
POST [adr]/Categories
```

s tělem

```
{ "@odata.type":"#Server.Edm.Category",
  "Id":0,
  "Name":"Misc",
  "ParentCategoryId":null,
  "StoreItems@odata.bind":["http://localhost:50921/odata/StoreItems('knf')",
                           "http://localhost:50921/odata/StoreItems('tshrt')"]
}
```

Dotaz žádá službu o vytvoření entity, a pomocí *@odata.bind* rovněž o nastavení navigační vlastnosti (Category) na hodnotu nově vytvořené entity u entit StoreItem(‘knf’) a StoreItem(‘tshrt’). V ukázkové databázi to znamená nastavit sloupec *StoreItem.CategoryId* na hodnotu *Category.Id* nově vytvořené entity.

V případě, že je potřeba vytvořit záznam v navigační vlastnosti entity, je nutné v URL uvést název navigační vlastnosti a klíčové slovo *$ref*. Pokud se reference vkládá do vlastnosti typu kolekce entit, odesílá se požadavek metodou *POST*. Pokud se reference vkládá do vlastnosti typu entita, odesílá se požadavek metodou *PUT*. Příkladem by mohl být dotaz

```
PUT [adr]/odata/Categories(3)/ParentCategory/$ref
```

s tělem

```
{"@odata.id":"http://localhost:50921/odata/Categories(1)"}
```

Pro entitu *Category s Id 3* je nastavena navigační vlastnost (typu entita) *ParentCategory* na entitu *Category s Id 1*. V ukázkové databázi by se dotaz projevil nastavením sloupce *Category.ParentCategoryId* na hodnotu 1.

Dotaz pro vložení reference entity do kolekce navigačních vlastností by mohl mít tvar adresy

```
POST [adr]/Customers(3)/Orders/$ref
```

s tělem

```
{"@odata.id":"http://localhost:50921/odata/Orders(2)"} .
```

2.2.2 Editace dat

Editace entit probíhá pomocí metody PATCH. Při úspěšné editaci služba vrátí kód 200 nebo 204. Při editaci entit se odesílají pouze editované položky. Příkladem by mohl být dotaz s adresou

```
PATCH [adr]/Customers(1)
```

a tělem

```
{"@odata.type":"#Server.Edm.Customer","Firstname":"Walter"}
```

Jak je z ukázky patrné, klient změnil pouze vlastnost *Firstname*.

Služba může podporovat rovněž editaci pomocí metody PUT. V takovém případě se změní všechny strukturní vlastnosti dle těla požadavku. Pokud v těle není uvedena vlastnost, která má implicitní hodnotu, služba použije hodnotu implicitní. V případě, že chybějící vlastnost implicitní hodnotu nemá, je službou vrácena chyba 400.

Editaci navigačních vlastností je možné provést způsobem popsaným v kapitole 2.2.1.

2.2.3 Odstraňování dat

Odstranění entity je provedeno pomocí požadavku s metodou DELETE na URL dané entity. Příkladem by mohl být požadavek na adresu

```
DELETE [adr]/ContactTypes(1)
```

Tělo může být prázdné. Při úspěšně provedené operaci služba vrátí kód 204. Služba poté odstraní příslušné reference na danou entitu v navigačních vlastnostech svázaných entit. Pokud je při vykonávání požadavku porušena referenční integrita, operace selže a entita

není odstraněna. Služba může rovněž podporovat případnou modifikaci nebo odstranění příbuzných entit svázaných navigační vlastností.

Pro odstranění reference u navigační vlastnosti entity je nutné použít operátor *$ref*. Při odstraňování reference z kolekce navigačních vlastností příbuzné entity je dále potřeba poskytnout parametr *$id* s adresou odstraňované entity. Příkladem by mohl být dotaz

```
DELETE [adr]/Customers(3)/Orders/$ref?$id=[adr]/Orders(2) ,
```

který odstraní entitu *Order s Id 2* z navigační kolekce *Orders* entity *Customer s Id 3*. V podkladové databázi to znamená odstranění hodnoty ze sloupce *CustomerId* u entity *Order s Id 2*. Pokud je odstraňovaná navigační vlastnost typu entita, parametr $id není potřebný, jelikož je odstraňovaná reference jednoznačně dána. Například dotaz

```
DELETE [adr]/Categories(3)/ParentCategory/$ref
```

odstraní referenci u vlastnosti *ParentCategory* entity *Category s Id 3*. V podkladové databázi se operace projeví odstraněním hodnoty ve sloupci *ParentCategoryId* entity *Category s Id 3*.

2.2.4 Čtení dat

Čtení dat se provádí pomocí metody GET. Při úspěšném provedení dotazu služba vrátí kód 200. Přistupovat je možné k různým prvkům služby:

- Kolekce entit se může nacházet například na adrese

```
GET [adr]/Customers
```

- Jednotlivé entity se adresují dle primárního klíče, např.

```
GET [adr]/Customers(5)
```

- Vlastnosti entit se adresují pomocí názvu vlastnosti dané entity, např.

```
GET [adr]/Customers(5)/Lastname
```

- Pokud by vlastnost byla komplexním typem (např. typ *Address*, který obsahuje primitivní typ *City*), adresa by měla tvar

```
GET [adr]/Customers(5)/Address/City
```

- Navigační vlastnosti se adresují dle názvu navigační vlastnosti dané entity, např.

```
GET [adr]/Customers(5)/Orders
GET [adr]/Orders(3)/Customer
```

Při dotazování kolekcí entit je možné, že server použije tzv. *Server-Driven Paging*. Dotaz v takovém případě nevrátí všechny prvky kolekce, nýbrž serverem specifikovaný počet prvků. Výstup po aplikaci omezení počtu výsledků na dva záznamy by mohl mít tvar

```
"@odata.context": "http://localhost:50921/odata/$metadata#Orders",
"value": [
    {
        "Created": "2014-11-11T00:00:00+01:00",
        "Id": 1,
        "CustomerId": 1
    },
    {
        "Created": "2014-11-13T00:00:00+01:00",
        "Id": 2,
        "CustomerId": 1
    }
],
"@odata.nextLink": http://localhost:50921/odata/Orders?$skip=2
```

Pomocí informace *@odata.nextLink* je možné získat odkaz na další stránku záznamů.

V dotazech je dále možné využít tzv. *System Query Options*, které slouží k podrobnější specifikaci klientského dotazu. Uvádí se znakem *„$“*, a Jednotlivé operátory se oddělují znakem *„&“*. V následujících odstavcích budou uvedeny nejpoužívanější operátory a jejich význam.

Operátor *$select* slouží k výběru vlastností, které se mají ve výsledku dotazu zobrazit. Příklad použití by mohl mít podobu

```
GET [adr]/Customers?$select=Firstname,Lastname
```

Operátor *$expand* slouží k připojení obsahu navigační vlastnosti k výsledku dotazu. Příkladem může být dotaz

```
GET [adr]/Orders?$expand=OrderItems,Customer
```

jenž vrátí data ve tvaru

```
{
  "Created":"2014-11-11T00:00:00+01:00","Id":1,"CustomerId":1,"OrderItems":[
    {
      "OrderId":1,"StoreItemId":"knf","Count":1
    },{
      "OrderId":1,"StoreItemId":"mcht","Count":2
    }
  ],"Customer":{
    "Id":1,"Firstname":"Milan","Lastname":"Gaty\u00e1s","Note":"Test"
  }
}
```

Operátor *$orderby* definuje pořadí entit ve výsledku dotazu. Přiklad dotazu:

```
GET [adr]/Orders?$orderby=Id desc,CustomerId asc
```

Pokud není směr řazení definovaný, je implicitně předpokládaný směr *asc* (vzestupný).

Operátorem *$top* je možné z klientské strany omezit počet vrácených záznamů. Operátorem *$skip* je možné daný počet záznamů přeskočit. Příkladem může být dotaz

```
GET [adr]/Orders?$top=1&$skip=2
```

Službou je vrácen maximálně jeden záznam, který je na 3. Pozici výsledku dotazu.

Operátor *$count* slouží pro zobrazení počtu prvků ve výsledku dotazu. Užívá se dvojím způsobem:

```
GET [adr]/Orders?$count=true
GET [adr]/Orders/$count
```

V prvním případě se do těla odpovědi vloží informace

```
"@odata.count":3 ,
```

v případě druhém je výsledkem dotazu pouze číslo udávající počet záznamů ve výsledku dotazu.

Operátor *$format* rozhoduje o formátu dat vrácených službou. Možné argumenty jsou *json* a *atom*. Informaci o požadavku formátu je možné předat i v hlavičce požadavku vyplněním hlavičky *Accept*.

Operátor *$filter* slouží k omezení záznamů dle určitého kritéria. Příkladem by mohl být dotaz

```
GET [adr]/Orders?$filter=Id gt 2 or CustomerId eq 1,
```

který vrátí prky, jež mají *Id vetší než 2* nebo je *CustomerId rovno 1*.

Při filtraci dat je možné používat *filtrační operátory*. Přehled nejdůležitějších filtračních operátorů je uveden v následující tabulce.

Operátor	Význam
eq, ne	Ekvivalence, nonekvivalence
gt, ge, lt, le	Větší, větší rovno, menší, menší rovno
and or, not	Logické operátory součinu, součtu a negace
add, sub, mul, div, mod	Aritmetické operace součtu, rozdílu, násobení, dělení, a dělení modulo

Tabulka 5 Vybrané operátory pro filtraci dat [20]

Kromě filtračních operátorů je možné při filtraci dat použít i *vestavěné dotazovací funkce* (Build-in Query Functions). Výběr funkcí viz následující tabulka.

Funkce	Význam, příklad použití
contains, startswith, endswith	Filtruje entity dle existence podřetězce/začátku řetězce/konce řetězce ve vlastnosti entity. [adr]/Cutomers?$filter=contains(Firstname,‘lan’)
trim, tolower, toupper	U vlastnosti entity odstraní „bílé znaky“/převede řetězec na velká písmena/převede řetězec na malá písmena. [adr]/StoreItems?$filter=tolowert(Name) ne ‘t-shirt’
year, month, day	Získá rok/měsíc/den z data. [adr]/Orders?$filter=year(Created) eq 2015
round, ceiling, floor	Zaokrouhlí číslo dle desetinné čárky/nahoru/dolů. [adr]/StoreItems?$filter=round(Price) gt 1000

Tabulka 6 Vybrané funkce pro filtraci dat [20]

Při dotazování je samozřejmě možné dotazovací operátory kombinovat a vnořovat, např.

```
GET [adr]/Orders?$top=1&$expand=OrderItems($expand=StoreItem($select=Id))
```

Dotaz zobrazí 1. entitu *Order*. Zobrazí rovněž seznam navigačních vlastností *OrderItems*, a pro každou z nich zobrazí navigační vlastnost *StoreItem*, pro kterou zobrazí jen vlastnost *Id*. Obsah těla odpovědi služby může mít tvar

```
{
"@odata.context":"http://localhost:50921/odata/$metadata#Orders(OrderItems(St
oreItem(Id)))","value":[
```

```
    {
      "Created":"2014-11-11T00:00:00+01:00",
      "Id":1,"CustomerId":1,"OrderItems":[
        {
          "OrderId":1,"StoreItemId":"knf","Count":1,
          "StoreItem":{ "Id":"knf" }
        },{
          "OrderId":1,"StoreItemId":"mcht","Count":2,
          "StoreItem":{ "Id":"mcht" }
        }
      ]
    }
  ]
}
```

2.2.5 Volání akcí

K akcím se přistupuje skrze dotazy s metodou POST. V kapitole 2.1.2 byla uvedena ukázková akce ve tvaru

```
<Schema xmlns="http://docs.oasis-open.org/odata/ns/edm" Namespace="Default">
  <Action Name="AddOrderItem" IsBound="true">
    <Parameter Name="bindingParameter" Type="Collection(Server.Edm.Order)"/>
    <Parameter Name="addOrderItemModel" Type="Server.Edm.AddOrderItemModel"/>
  </Action>
</Schema>
```

Akce je *vázána na kolekci typu Order* a nachází se ve *jmenném prostoru* s názvem *Default*. Požadavek bude mít v takovém případě tvar adresy

```
POST http://localhost:50921/odata/Orders/Default.AddOrderItem
```

s tělem

```
{"addOrderItemModel":
   {"@odata.type":"#Server.Edm.AddOrderItemModel",
    "orderId":2,"orderItemId":1}} .
```

Po úspěšném provedení akce služba vrátí kód 201 (pokud akce vytváří nové entity), případně kód 200 (akce vrací výsledek), nebo kód 204 (akce nemá návratovou hodnotu).

2.2.6 Volání funkcí

Funkce datové služby se volají pomocí metody GET. V kapitole 2.1.2 byl uveden příklad funkce ve tvaru

```
<Schema xmlns="http://docs.oasis-open.org/odata/ns/edm" Namespace="Default">
  <Function Name="GetTotalCost" IsBound="true">
    <Parameter Name="bindingParameter" Type="Collection(Server.Edm.Order)"/>
    <Parameter Name="orderId" Type="Edm.Int32" Nullable="false"/>
    <ReturnType Type="Edm.Double" Nullable="false"/>
  </Function>
</Schema>
```

Tato funkce je (stejně jako demonstrovaná akce) vázaná na *kolekci typu Order* a nachází se ve *jmenném prostoru* s názvem *Default*. Volání funkce má potom tvar

```
GET [adr]/Orders/Default.GetTotalCost(orderId=1)
```

a služba může vrátit například řetězec

```
{"@odata.context":"http://localhost:50921/odata/$metadata#Edm.Decimal",
 "value":8500.5000}
```

Pokud by byla funkce vázaná přímo na entitu, dotaz by měl tvar

```
GET [adr]/Orders(1)/Default.GetTotalCost [19][20].
```

3 MOŽNOSTI POSKYTOVÁNÍ ODATA SLUŽEB

OData služby je možné díky jednoznačnosti předpisu chování služby a využití HTTP protokolu provozovat nad různorodou infrastrukturou [21].

V prostředí jazyka *Java* je možné využít knihovnu *Apache Olingo*. Knihovna podporuje tvorbu OData služeb ve verzi č. 2. Připravuje se rovněž podpora pro verzi č. 4, která je nyní ve fázi beta [22].

V prostředí *PHP* je možné využít knihovnu *OData Producer Library for PHP*. Pomocí této knihovny je možné vytvářet služby ve verzi protokolu č. 2. Knihovna podporuje pouze poskytování dat pro čtení. Zbylé operace služba v současné době neposkytuje [23].

V prostředí *.NET* je možné využít dvou způsobů implementace OData služeb. Prvním je použití technologie *WCF Data Services*. Pomocí WCF Data Services je možné vytvářet OData služby maximálně ve verzi protokolu č. 3 [24]. Druhou možností je využití technologie *ASP.NET Web Api OData*. Pomocí této technologie je možné vytvářet služby jak ve verzi 3, tak ve verzi 4 [25].

WCF Data Services slouží jako jednoduchá cesta pro implementování OData služeb nad datovými zdroji. Pomocí vhodného nástroje typu Entity Framework knihovna ukryje veškeré detaily ohledně HTTP přenosu a implementace operací předepsaných OData protokolem. Takovéto chování je vhodné pro datový zdroj, který poskytuje bohaté dotazovací možnosti. Některé datové zdroje ovšem popsané vlastnosti postrádají (data vrácená z volání funkcí, nebo data přijata z jiných datových služeb). Zároveň je pro uživatele téměř nemožné přidávat do služby nové funkcionality, jelikož se pro něj knihovna WCF Data Services tváří víceméně jako černá skříňka.

Na základě těchto omezení se OData tým společnosti Microsoft rozhodl více podporovat knihovnu ASP.NET Web Api OData, která umožňuje širší kontrolu nad samotnou službou. Sám vývojář může implementovat funkcionality protokolu, které ještě vlastní knihovna neobsahuje [26].

3.1 ASP.NET Web Api

ASP.NET Web Api je knihovna umožňující poskytování webových služeb typu REST. Komunikace mezi klientem a serverem probíhá pomocí HTTP protokolu a jeho metod. Podobně jako knihovna ASP.NET MVC[3] využívá *kontroléry* (controllers) a *akce* (actions). Kontroléry slouží jako vstupní brány pro jednotlivé typy poskytovaných entit, akce potom jako jednotlivé operace, jež jsou definovány nad danými entitami. Knihovna nese název *Microsoft.AspNet.WebApi* a je možné ji stáhnout a nainstalovat jako NuGet balíček. Na této knihovně je přímo závislá knihovna, jež umožňuje tvorbu OData webových služeb.

3.1.1 Podpora datového typu DateTime

Jak už bylo uvedeno v kapitole 2.1.2, OData protokol verze 4 nespecifikuje takový typ *DateTime* mezi své základní typy. Nicméně v ASP.NET Web Api OData 5.4 je tento datový typ *podporován*. Podkladový datový zdroj služby může tento datový typ u svých entit obsahovat. V takovém případě se tato vlastnost entity automaticky přeloží do datového typu *DateTimeOffset* a naopak [27].

3.1.2 Možnosti hostování ASP.NET Web Api služeb

Nejnižší vrstva Web Api architektury je odpovědná za hostování služby. Slouží jako rozhraní mezi Web Api a HTTP protokolem. Tato vrstva je odpovědná za vytváření HTTP požadavků, jež předává vyšším vrstvám, a vytváření HTTP odpovědí, které předává podkladovému protokolu. Jedná se o třídy *HttpRequestMessage* a *HttpResponseMessage*. Díky této vrstvě je možné, aby zmíněné zprávy mohli být vytvářeny z různých hostovacích zdrojů [29].

Jednou z možností hostování je tzv. *self hosting*. Self hosting umožňuje hostovat Web Api např. v konzolové aplikaci, nebo windows službě. Tento způsob hostování je vhodný například při ladění a testování služby. Pro hostování tohoto typu se používají třídy *HttpSelfHostServer* a *HttpSelfHostConfiguration*. Více informací o možnostech self hostování viz [28].

[3] http://www.asp.net/mvc

Další možností hostování Web Api služby je tzv. *web hosting*. Tento způsob hostování využívá rozhraní *ASP.NET* nad *IIS* (Internet Information Service). Hlavním prvkem je zde třída *HttpControllerHandler*, která propojuje ASP.NET architekturu s architekturou Web Api. Činí tak překladem instancí typu *HttpRequest* (nebo *HttpResponse*) do instancí typu *HttpReqestMessage* (*HttpResponseMessage*), a naopak. Tento způsob hostování je již možné použít pro ostrý provoz služby.

Jako poslední bude v tomto textu zmíněna možnost hostování na serveru splňující specifikaci *OWIN* (Open Web Interface for .NET). OWIN je standard, který definuje způsob komunikace mezi serverem a aplikačními komponentami. Pro hostování aplikace je dostačující, aby server splňoval daný standard požadovaný aplikačními komponentami.

OWIN rovněž umožňuje větší modularitu prostředí ASP.NET jako takového. Cílem rozhraní je odstranit závislost webové aplikace na knihovně *System.Web*, a možnost vybrat si pouze moduly, které jsou pro aplikaci skutečně potřeba. V současné době je možné skrze OWIN vytvářet Web Api a SignalR[4] aplikace. Podpora pro MVC aplikace se očekává s vydáním ASP.NET vNext[5], kde se MVC a Web Api frameworky sjednotí do frameworku MVC 6 [29].

3.1.3 Životní cyklus zpracování požadavku

Pro vytvoření Web Api služby je vhodné mít přehled o procesech, které probíhají jak při zpracování požadavků klienta, tak odesílání odpovědí ze strany serveru. Následující text lze dokumentovat přílohou PII.

Proces zpracování klientského požadavku začíná převedením požadavku na třídu *HttpRequestMessage*. Za tuto činnost je odpovědný hostující subsystém.

Zprávu je poté možné zachytit pomocí tzv. *Message handlers*. Message handlery mohou zprávu editovat nebo validovat, případně mohou rovnou vytvořit odpověď pro klienta, přičemž zcela obejdou zbytek procesu zpracování zprávy.

---

[4] http://www.asp.net/signalr

[5] http://www.asp.net/vnext

Dalším krokem je třída *HttpRoutingDispatcher*, jejímž úkolem je nalézt pro požadavek vhodný *kontrolér*. Kontrolér je nalezený z *tvaru URL* adresy požadavku a *cest* (routes), které jsou pro danou službu definované. Následně se vytvoří instance kontroléru.

Poté se nalezne vhodná *akce*. Akce je nalezena dle *URL požadavku* a *HTTP metody* požadavku.

Dále je možné použít *autentizační a autorizační filtry*. Jak název napovídá, tyto filtry mohou být využity pro autentizaci a autorizaci požadavků.

Z požadavku jsou poté namapovány argumenty pro akci (pokud je akce požaduje). K tomuto účelu se využívá tzv. *Model binding*. Informace lze získat z *URL* požadavku, *hlaviček* požadavku, nebo *těla* požadavku.

Dále je možné do procesu zasáhnout pomocí *filtrů akcí* (action filters). Filtr je volaný *před* provedením akce, a následně *po* jejím provedení. Tímto způsobem je tedy možné provést například určitý preprocessing a postprocessing, případně logování, apod.

Po vrácení řízení z akce je na základě jejího výsledku vytvořena zpráva typu *HttpResponseMessage*.

Tato zpráva opět projde filtry akcí, autentizačními filtry (pro případnou výzvu k autentizaci), třídami HttpControllerDispatcher, HttpRoutingDispatcher a message handlery. Nakonec je odpověď předána hostovací vrstvě, která ji předá podkladovému HTTP spojení [29].

3.2 Entity Framework

Entity Framework je nadstavba nad technologií ADO.NET (Microsoft ActiveX Data Object.NET). Hlavní úlohou této knihovny je abstrakce konkrétního datového úložiště a jeho struktury do formy modelů.

Pojem entita specifikuje data, jež jsou spojena s určitým objektem určité aplikace. Může to být například entita s názvem Zákazník, kterou charakterizují informace Jméno, Příjmení, Adresa, apod. Na entity se lze dívat z různých úhlů pohledů:

- *Fyzický* – Data jsou specifikována např. pomocí tabulek, klíčů, omezení a indexů. Mohou být rozdělena i do více tabulek. Vývojář musí rozumět konkrétnímu databázovému mechanismu.

- *Logický* – Jednotlivé fyzické celky spojené do celku logického. V relační databázi se jedná o pohledy. Vývojář musí opět rozumět konkrétnímu databázovému mechanismu.
- *Konceptuální* – Data jsou konceptuálním obrazem skutečného světa. Informace jsou reprezentovány entitami a jejich atributy. Důraz je kladen na tyto informace, nikoliv na strukturu podkladové databáze. Zde je již vývojář oproštěn od nutnosti znalosti databáze.

3.2.1 Modelové soubory

Entity Framework je nástroj, jenž je vhodný pro vytváření konceptuálních modelů z modelů fyzických a logických. K realizaci této činnosti využívá XML souborů pro definici *konceptuá*lního modelu (conceptual model), *datového* modelu (storage model), a *mapování* (mapping) mezi modely a podkladovou databází.

Konceptuální model definuje podobu databáze z pohledu aplikace. Zároveň obsahuje třídy, které umožňuji vývojáři komunikovat s databází. Tento model je specifikovaný pomocí CSDL (Conceptual Schema Definition Language) souboru. Ukázkový soubor by mohl mít podobu

```
<edmx:ConceptualModels>
  <Schema xmlns="http://schemas.microsoft.com/ado/2009/11/edm"...>
    <EntityContainer Name="Model1Container"
    annotation:LazyLoadingEnabled="true">
      <EntitySet Name="Customers" EntityType="Model1.Customer" />
    </EntityContainer>
    <EntityType Name="Customer">
      <Key> <PropertyRef Name="CustomerID" /> </Key>
      <Property Type="Int32" Name="CustomerID" Nullable="false"
        annotation:StoreGeneratedPattern="Identity" />
      <Property Type="String" Name="FirstName" Nullable="false" />
      <Property Type="String" Name="LastName" Nullable="false" />
      <Property Type="String" Name="AddressLine1" Nullable="false" />
      <Property Type="String" Name="AddressLine2" Nullable="false" />
      <Property Type="String" Name="City" Nullable="false" />
    </EntityType>
  </Schema>
</edmx:ConceptualModels>
```

Jak je na první pohled patrné, model je až na několik detailů totožný s CSDL (Common Schema Definition Language) souborem, jež definuje entity a ostatní objekty poskytované OData službou. Tato skutečnost může být využita pro zjednodušení implementace služby, jak je uvedeno v kapitole 6.2.2.

Datový model definuje podobu databáze z pohledu podkladové databáze. Podobně jako CSDL model, sestává tento model z entit a jejich definicí. Avšak zde jsou entity logickým obrazem skutečné databáze. Jsou zde tedy využity datové typy konkrétní databáze, databázové příkazy, apod.

Mapovací model umožňuje přecházet mezi konceptuálním a datovým modelem. Model definuje, jak překládat entity, jejich vlastnosti a vazby mezi těmito modely. Mapování umožňuje provádět činnosti jako připojení k databázi, modifikace databáze, apod. s minimálním množstvím manuálně napsaného kódu.

3.2.2 Techniky využívání Entity Framework

Entity Framework je možné využívat pomocí tří způsobů:

- *Database First* – Nejdříve je vytvořena podkladová databáze, a na jejím základě je vygenerován konceptuální model.
- *Design First* – Pomocí uživatelského rozhraní je vytvořen model databáze. Poté je vytvořena struktura databáze a konceptuální model s objektovými třídami.
- *Code First* – Nejdříve jsou vytvořeny třídy, jež reprezentují konceptuální entity. Poté je vytvořena databáze na základě těchto tříd. Konceptuální model fyzicky neexistuje. Je automaticky generován a uchován v paměti.

Techniky je možné kombinovat za účelem maximální kontroly nad procesem. Např. je možné simultánně vytvářet strukturu databáze, a zároveň psát konceptuální třídy. Rovněž je možné vygenerovat Code First třídy z již existující databáze [30].

4 MOŽNOSTI KONZUMOVÁNÍ ODATA SLUŽEB

Konzumace OData služeb je možná ze všech prostředí, jež mohou komunikovat skrze HTTP protokol. Práci se službou usnadňují klientské knihovny, pomocí kterých klient se službou komunikuje.

V prostředí jazyka *Java* je možné využít knihovnu *Apache Olingo*. Stejně jako při popisu knihovny v kapitole 3 je možné konzumovat služby protokolu verze 2, nebo 4 [22].

V *JavaScript* prostředí lze využít knihovnu *ODataJS* od *Apache Olingo*. Tato knihovna je momentálně ve fázi beta a umožňuje konzumovat služby ve verzi 4. Knihovna je založena na knihovně *datajs*, která je jejím předchůdcem. Pomocí datajs je možné konzumovat služby verze 1 až 3 [22][31].

V prostředí jazyka *C++* lze využít knihovnu *ODataCpp*. Tato knihovna slouží pro konzumaci OData služeb verze 4. V současné době se pracuje rovněž na možnosti poskytování OData služeb [32].

V prostředí jazyka *Python* je možné využít knihovnu *ODataPy*. Knihovna podporuje protokol verze 4. Je postavena nad knihovnou ODataCpp. V současné době je knihovna stále ve vývoji, a podporuje pouze některé klientské funkcionality [33].

V *.NET* prostředí lze využít šablonu OData *v4 Client Code Generator*. Tato šablona je zdarma dostupný doplněk do vývojového prostředí Visual Studio. Knihovna podporuje protokol verze 4. Pro konzumaci služeb nižší verze protokolu je možné použít knihovnu *WCF Data Services Client for OData v1-3* [34][35].

V textu uvedené výčty knihoven umožňujících poskytování a konzumaci OData služeb nejsou kompletní. Uvedeny jsou zejména knihovny, jež jsou doporučeny organizací OASIS. Pro podrobnější seznam možných alternativ viz [21].

5 MOŽNOSTI ZABEZPEČENÍ ODATA SLUŽEB

Služba, která svůj obsah potřebuje chránit před neoprávněným přístupem, musí být chráněna pomocí *autentizace*, případně pomocí *autorizace* pro dosažení větší granularity. Na základě informací získaných pomocí autentizace a autorizace je možné rozhodnout se o přidělení požadovaných prostředků.

5.1 Autentizace

Autentizace je proces, při kterém se klient služby *identifikuje*. Autentizace se u datových služeb provádí nejčastěji pomocí *jména a hesla*, případně autentizačního *tokenu*. Jelikož je OData protokol provozován nad HTTP protokolem, bude další text směrován na možnosti autentizace v HTTP prostředí. Doporučeným způsobem zajištění autentizace je autentizace typu *Basic nad HTTPS protokolem*. Tento způsob zajistí maximální šanci na kompatibilitu s generickým klientem služby. V textu budou uvedeny i alternativní možnosti autentizace [36].

5.1.1 Basic

Basic autentizace je založena na předložení *uživatelského jména a hesla pro určitou oblast* (realm). Oblastí v rámci serveru může být *více*, rovněž ale nemusí být definována *žádná*.

Pokud se neautentizovaný klient pokusí získat přístup ke zdroji, který je chráněn autentizací v rámci oblasti „Finance“, server vrátí odpověď s chybou 401 (Unauthorized) a přiloží header s *výzvou k autentizaci* (challenge) ve tvaru

```
WWW-Authenticate: Basic realm="Finance"
```

Pro získání autentizace je potřeba při odesílání žádosti odeslat i *autentizační header*. Součástí obsahu headeru jsou přihlašovací údaje klienta ve tvaru jméno:heslo, které jsou zakódovány pomocí base64[6] kódování. Pokud by se tedy klient přihlašoval pomocí uživatelského jména Customer a hesla Secret, autentizační header by měl tvar

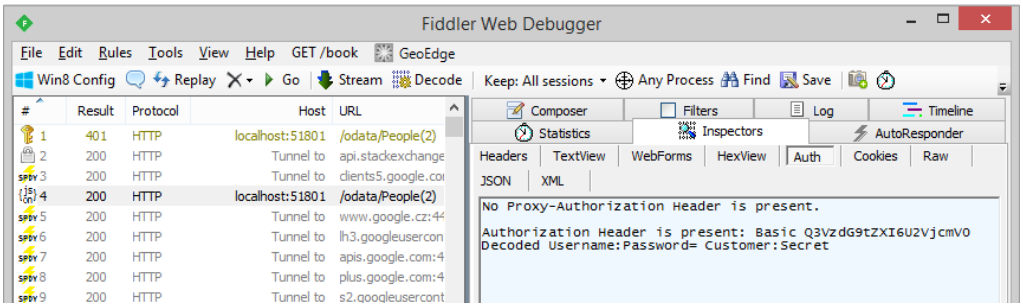
```
Authorization: Basic Q3VzdG9tZXI6U2VjcmV0
```

Tento header je platný pro celou oblast „Finance”.

[6] https://tools.ietf.org/html/rfc4648

Výhodou této autentizační metody je *jednoduchost*. Nevýhodou je *slabá zabezpečenost*. Jméno a heslo je přenášeno ve formě, která je lehce dekódovatelná, a tudíž je možné přihlašovací údaje získat sniffovacím programem a následným dekódováním je převést do čitelné podoby (viz obrázek 2). Schéma Basic by bez dalšího vylepšení nemělo být používáno pro přenos citlivých informací.

Toto schéma je možné v základní verzi použít např. pro statistické účely.



Obrázek 2 Dekódovaný Basic autentizační header

5.1.2 Digest

Schéma Digest funguje na podobném principu jako Basic. Oproti schématu Basic je ale uživatelovo heslo chráněno pomocí *hashovací funkce*.

V případě, že neautentizovaný klient požaduje zabezpečený obsah, vrátí server *challenge* ve tvaru

```
WWW-Authenticate: Digest challenge
```

kde challenge se skládá přinejmenším z

```
realm | nonce
```

kde *realm* značí oblast, pro kterou platí přihlášení a *nonce* (number used once) značí řetězec, který je vygenerovaný serverem. Nonce by měl být jiný po každém neúspěšném požadavku ze strany klienta. Navíc se může nastavit životnost (ve formě počtu requestů nebo stáří), po které se nonce změní. Nonce by měl mít tvar hexadecimálního, nebo base64 zakódovaného řetězce.

Dále je možné přidat položku *algorithm*, kterou je možné určit použitou hashovací funkci. Pokud není algorithm určen, implicitně je zvolený algoritmus MD5[7].

Rovněž je možné přidat položku *qop* (quality of protection), která určuje vlastní způsob implementace zabezpečení. Tato položka je dle standardu volitelná, doporučuje se ji však explicitně uvádět ve všech hlavičkách. Při absenci se předpokládá implementace dle standardu RFC 2069.

Header odeslaný serverem by tedy mohl mít tvar:

```
WWW-Authenticate: Digest realm="Finance", qop="auth", nonce="1e58g9u1458g5"
```

Klient potom opakuje žádost o obsah s autentizačním headrem:

```
Authorization: Digest, realm="Finance", qop="auth", username="Customer"
      nonce="1e58g9u1458g5", uri="odata/Customers(2)", nc=000000002,
      cnonce="5s4g35fs468", response="e5f1nb8j8..."
```

V případě, že je qop je nastaven na hodnotu „auth", klient vygeneruje hodnoty pro položky *cnonce* (client nonce) a *nc* (client nonce counter). Cnonce je náhodný řetězec generovaný klientem (opět hexadecimální nebo base64 zakódovaný). Nc je počítadlo, které se inkrementuje při každém odeslání požadavku na server. Z těchto údajů se vygeneruje hodnota *response* dle vztahu

$$response = MD5(H1 : nonce : nc : cnonce : qop : H2) \tag{1}$$

kde

$$H1 = MD5(username : realm : password) \tag{2}$$

$$H2 = MD5(method : uri) \tag{3}$$

Method je metoda dotazu (GET, POST, apod.), a *uri* je relativní cesta k požadovanému prostředku služby.

Poté, co server obdrží daný header, zkontroluje, zda nevypršela životnost řetězce nonce. Pokud ano, vrátí opět challenge pro zadání údajů s novým nonce řetězcem. Jinak server přečte hodnotu nc a porovná ji s poslední obdrženou u daného klienta (server si tedy musí pamatovat hodnoty pro dané klienty). Pokud je hodnota nc menší nebo rovna než poslední

[7] http://tools.ietf.org/html/rfc1321

pamatovaná, server vrátí chybu 401 a celý proces je třeba opakovat. Jinak najde heslo patřící danému klientovi dle položky headeru s názvem username (z databáze nebo jiného zdroje). Nakonec server provede výpočet dle vztahu (1) a porovná výsledky. V případě, že jsou totožné, je klient považován za autentizovaného a je mu poskytnut přístup ke chráněnému obsahu.

Pokud by útočník získal přístup k autentizačním headeru, nemůže opakovat pokus o přístup k prostředku služby, aniž by musel inkrementovat hodnotu nc. Tím pádem by ale musel vytvořit novou hodnotu položky response, k čemuž mu chybí znalost uživatelského hesla. Tímto způsobem je zajištěna ochrana vůči tzv. *Replay attack*. V RFC verzi 2069 položky cnonce a nc nejsou používány, a tudíž může útočník použít odchycený header pro přístup k prostředku na dané URL.

Jak je z popisu patrné, schéma Digest poskytuje pouze ochranu přihlašovacích údajů. Vlastní data jsou stejně jako ve schématu Basic přenášeny v nezabezpečené formě [37].

5.1.3 OAuth

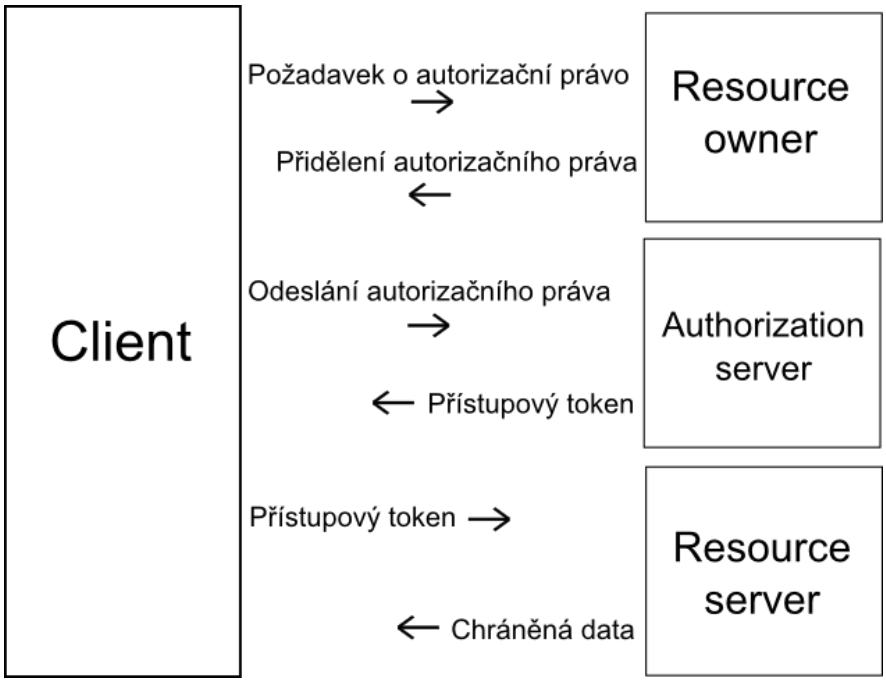
Schéma OAuth na rozdíl od předchozích schémat rozlišuje mezi vlastníkem přístupu k zabezpečeným datům a klientem, který tyto data požaduje. Schéma zavádí následující termíny:

- *Resource owner* je prvek schopen přiřazovat přístup ke chráněným zdrojům na základě znalosti uživatelského jména a hesla.
- *Resource server* uchovává chráněná data, která zpřístupňuje na základě přístupového tokenu.
- *Client* přistupuje k chráněným zdrojům resource serveru na základě přístupového tokenu.
- *Authorization server* vydává klientovi přístupový token na základě úspěšné autentizace resource owner-a.

Resource owner tedy může klientovi poskytovat přístup k chráněným zdrojům, aniž by mu musel sdělovat své přihlašovací údaje. Token, který klient obdrží, může být časově omezený a může poskytovat přístup pouze k podmnožině dat vlastněných resource owner-em.

Resource server a authorization server mohou být různé, tzn., uživatel se může autentizovat např. skrze facebook (authorization server), a resouce server může využít vygenerova-

ný token k získání informací o resource owner-ovi. Abstraktní průběh OAuth autentizace viz obrázek 3.



Obrázek 3 Abstraktní OAuth schéma [38]

Proces požádání o *autorizační právo* (authorization grant) a jeho získání může být realizován:

- *Autorizačním kódem* (authorization code), kde klient přesměruje resource owner-a na autorizační server, který ho po ověření údajů (autentizaci) přesměruje zpět na klienta spolu s autorizačním kódem. Klient tento autorizační kód odešle autorizačnímu serveru, a na jeho základě je mu poskytnut přístupový token. Výhodou tohoto přístupu je, že klient nemá žádný přístup k údajům resource owner-a, pracuje pouze s autorizačním kódem.
- *Implicitně*, kde klient přesměruje resource owner-a na autorizační server, který ověří jeho údaje. Na jejich základě resource owner-ovi přidělí přístupový token, a přesměruje ho zpět na klienta. Proces je tedy zjednodušenou formou realizace autorizačním kódem. Z tohoto schématu také vyplívá, že klient není nijak autentizován. Toto schéma je šetrnější k objemu přenesených dat a zvyšuje rychlost odezvy aplikace. Na druhou stranu má resource owner přístup k tokenu, který je určen pouze pro klienta.

- *Přihlašovacími údaji resource owner-a*, na základě kterých je přidělen přístupový token. Tento přístup se doporučuje implementovat pouze tehdy, když je možné klientovi důvěřovat a sdělit mu údaje resource owner-a.
- *Přihlašovacími údaji klienta*. Toto schéma lze uplatnit, pokud je klientem sám resource owner, anebo klient přistupuje k chráněným zdrojům, které sám spravuje.

Jak přihlašovací údaje a tokeny, tak chráněná data, jsou dle specifikace přenášeny v nechráněné formě. Je tedy doporučeno použít dalších prostředků pro zabezpečení přenášených dat [38].

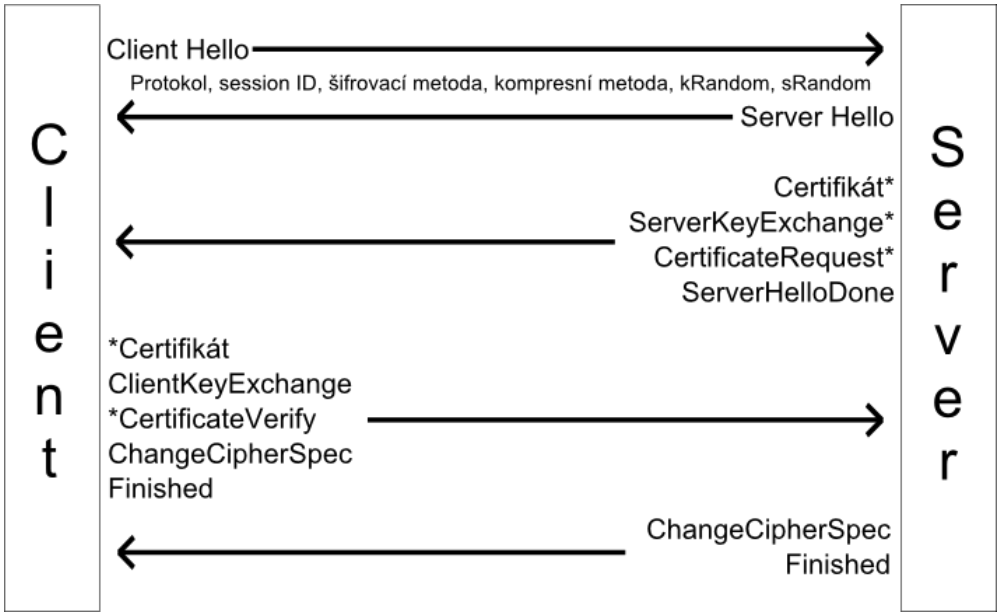
5.1.4 HTTPS

Výše zmíněná schémata sdílí jednu společnou vlastnost. Obsah přenášených dat není nijak zabezpečen. Mimo schéma Digest nejsou zabezpečeny ani přihlašovací údaje uživatele. Použitím sniffovácích nástrojů je potom možné probíhající komunikaci zachytit, a data si přečíst.

Této nežádoucí vlastnosti popsaných schémat lze předejít zapouzdřením celého HTTP protokolu do „ochranného obalu“, zvaného *HTTPS* (HTTP Secure). Protokol umožnuje celou komunikaci zabezpečit pomocí sady *šifrování*, které jsou implementovány pomocí *SSL* (Secure Socket Layer), nebo jeho nástupcem, *TLS* (Transport Layer Security).

Mezi protokoly SSL (v3.0) a TLS (v1.2) existují odlišnosti, v zásadě jsou však principiálně podobné. Protokol SSL v současné době již není považovaný za bezpečný. V další části textu bude popsán protokol TLS v1.2.

Protokol TLS se skládá ze dvou částí: Z *TLS Handshake* protokolu a *TLS Record* protokolu. Schéma Handshake protokolu viz obrázek 4.



Obrázek 4 TLS Handshake [40]

Handshake protokol slouží k navázání spojení se serverem, dohodnutí se na použitých algoritmech, volitelné autentizaci a inicializaci proměnných pro dané algoritmy.

V prvním kroku odešle klient zprávu typu *Hello*, ve které sdělí serveru podporované verze TLS protokolu, volitelně stavové identifikační číslo sezení (pro pokračování v předchozím sezení), požadované kombinace metod šifrování a komprese seřazeny sestupně dle preference klienta a náhodně vygenerované číslo (kRandom).

Server odpoví rovněž zprávou typu *Hello*, ve které uvede nejvyšší podporovanou verzi TLS protokolu ze seznamu nabízených, číslo sezení (vygenerované klientem, anebo nově přidělené), první podporovanou kombinaci šifrovacích a kompresních metod ze seznamu nabízených a náhodně vygenerované číslo (sRandom).

Další fází je *výměna klíčů* (key exchange). Úkolem výměny klíčů je vygenerování tzv. *pre_master_secret* řetězce. Tato fáze se liší v závislosti na zvolených šifrovacích a kompresních metodách.

- Při *Anonymní výměně klíčů* se nekontroluje totožnost serveru ani klienta. Certifikáty serveru ani klienta se tedy neposílají. Místo nich se používají zprávy *ServerKeyExchange* a *ClientKeyExchange*. Může se zde uplatnit např. algoritmus

Diffie-Hellman[8]. Tento algoritmus umožňuje provést dohodu mezi klientem a serverem na společném klíči, aniž by bylo možné získat tento klíč třetí stranou např. sniffováním komunikace. Výměna ale není imunní vůči *MITM* (Man In The Middle) útoku, jelikož díky absenci autentizace je možné, aby se třetí strana pro server tvářila jako klient, a pro klienta jako server.

- V případě *Autentizovaného serveru* se provádí autentizace serveru na základě jeho certifikátu. Může se zkombinovat např. s *RSA*[9] algoritmem, který pracuje na bázi asymetrického šifrování. Klient obdrží certifikát, který prověří. Certifikát musí být platný a musí být ověřený důvěryhodnou certifikační autoritou. Spolu s certifikátem server odešle i veřejnou část klíče asymetrické šifry. Klient vytvoří pre_master_secret, který zašifruje pomocí dodaného veřejného klíče. Ten potom pošle zprávou ClientKeyExchange. Server poté řetězec dešifruje pomocí svého tajného klíče. Zde již není možné, aby se třetí strana tvářila jako server, jelikož je server jednoznačně určen svým digitálně podepsaným certifikátem.
- Při výměně klíčů typu *Autentizovaný server a klient* je průběh výměny klíče podobný, jako u autentizovaného serveru. Navíc se zde provádí výzva zprávou *CertificateRequest* ze strany serveru, a autentizace klienta pomocí jeho certifikátu.

Další kroky jsou pro všechny typy výměny klíčů stejné. Klient odešle zprávu *ChangeCipherSpec*, která informuje server, že další komunikace z jeho strany už bude v šifrované formě dle dohodnutých pravidel. Poté odešle zprávu *Finished*. Server odpoví rovněž sekvencí *ChangeCipherSpec, Finished.* Tímto je vlastní proces handshake u konce.

Po vzájemné výměně se pre_master_secret použije pro výpočet *master_secret*. Tento řetězec si vypočítá server i klient, a bude sloužit k vlastní ochraně přenesených dat. Master secret je 48 bytů dlouhý klíč, který se vypočte ze vztahu

$$master\_secret = PRF(pre\_master\_secret, "master\_secret", kRandom + sRandom) \quad (4)$$

[8] https://www.ietf.org/rfc/rfc2631.txt

[9] https://www.ietf.org/rfc/rfc2437.txt

PRF (Pseudo Random Function) je algoritmus, který generuje hash sekvenci v závislosti na argumentech pre_master_secret, identifikačního popisu a násady. Podoba PRF je určena na základě dohodnutých metod v handshake fázi.

Record protokol slouží k šifrování komunikace. Master_secret se rozdělí na část pro šifrování dat pomocí algoritmu symetrického šifrování (dohodnutého v handshake fázi), a *MAC* (Message Authentication Code), který slouží ke generování hashe zprávy. Tento hash vypočítá i druhá strana. Pokud je hash stejný, znamená to, že zprávu po trase nikdo needitoval [39][40].

5.2 Autorizace

Autorizace je proces, při kterém se rozhoduje, zda má (obvykle autentizovaný) uživatel právo (autorizaci) pro přístup k požadované službě. Toto rozhodování je založeno na dodatečných informacích o uživatelovi. Můžou to být např. skupiny nebo role, do kterých uživatel patří.

II. PRAKTICKÁ ČÁST

6 IMPLEMENTACE SLUŽBY

Jako datový zdroj služby bude využita databáze prezentovaná v kapitole 2. ER diagram databáze viz obrázek 1. V následujícím textu bude popsána implementace datové služby v podobě případových studií pro vybrané funkcionality definované OData protokolem.

Jak již bylo uvedeno v kapitole 3, možností implementace OData služby v prostředí .NET je více. Rozbor v tomto textu se soustředí na knihovnu ASP.NET Web Api OData, jelikož podporuje nejnovější verzi protokolu, a nabízí vyšší flexibilitu než knihovna WCF Data Services.

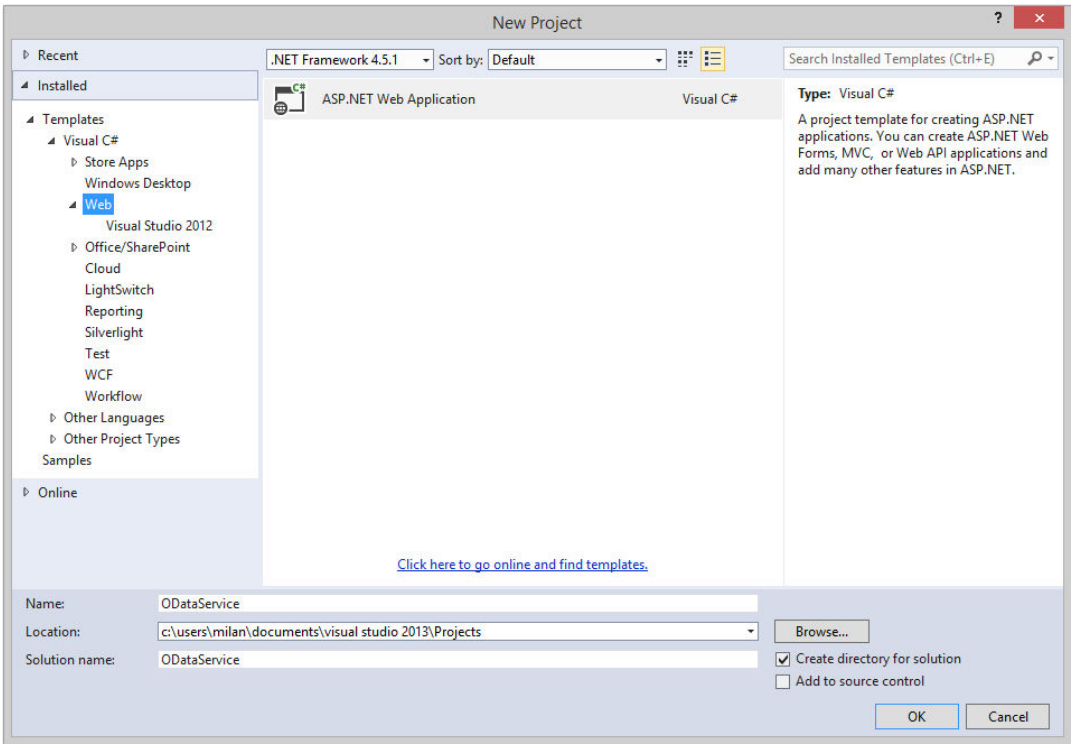
6.1 Vytvoření projektu a příprava k registraci služby

Prvním krokem k vytvoření datové služby je založení projektu. Při vytváření projektu je nutné vybrat jako typ projektu položku ASP.NET Web Application. Dále je možné specifikovat, jaká kostra předlohy ASP.NET projektu se má vytvořit, případně jaké knihovny se mají předinstalovat. V případě, že se zaškrtne políčko Web API, stáhnou se automaticky knihovny potřebné pro implementaci Web Api služby. Rovněž se stáhne knihovna Microsoft ASP.NET Web Api 2.1 Web Host, díky které je možné Web Api hostovat pomocí IIS. Obrázková dokumentace postupu viz obrázek 5 a obrázek 6.

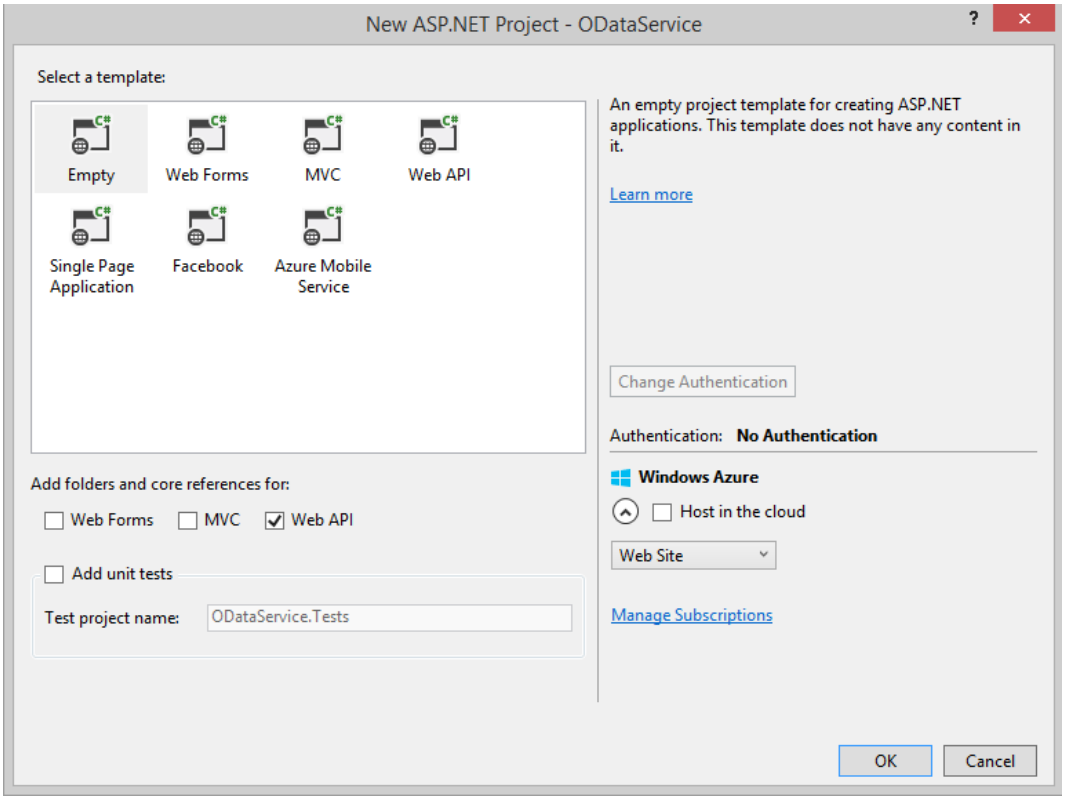
Dalším krokem je příprava k registraci datové služby. V případě, že bude služba hostována pomocí IIS, je možné využit metodu *Application_Start* třídy *HttpApplication*. Tato metoda se volá při startu webové aplikace, a je tedy vhodným místem pro provedení registrace. Konfigurace webové aplikace se provádí pomocí statické metody *Configure* třídy *GlobalConfiguration*. Tato metoda vyžaduje argument s názvem *configurationCallback*, který je typu *Action<HttpConfiguration>*. Pro přístup k metodě Application_Start je potřebné vytvořit v kořenovém adresáři projektu soubor s názvem *Global.asax*. Obsah souboru může mít podobu

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        GlobalConfiguration.Configure(WebApiConfig.Register);
    }
}
```

Argumentem metody je statická metoda, ve které se dále pracuje s instancí třídy HttpConfiguration. V této metodě se provede vlastní registrace služby.



Obrázek 5 Vytvoření ASP.NET Web aplikace



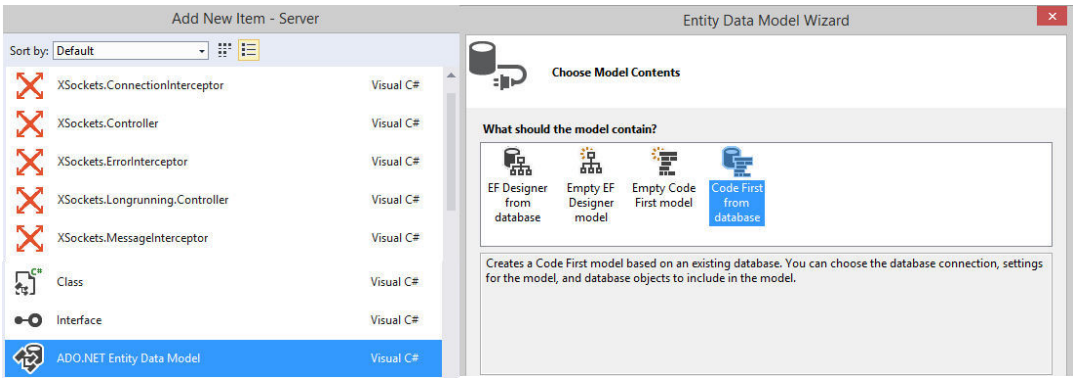
Obrázek 6 Specifikace šablony a knihoven nové ASP.NET Web aplikace

6.2 Registrace služby

Registrace služby se provádí pomocí metody *MapODataServiceRoute*, která patří do třídy *System.Web.OData.Extensions.HttpConfigurationExtensions*. Nejdříve je však potřebné specifikovat datový zdroj a model služby, který je pro registraci požadován.

6.2.1 Datový zdroj

Data, jež budou OData službou poskytována, jsou uchována v existující relační databázi. Obecně je možné použít jakýkoliv datový zdroj. Pro usnadnění komunikace mezi databází a datovou službou bude využita knihovna Entity Framework pomocí techniky Code First. Pro vytvoření modelu datového zdroje je potřebné přidat do projektu položku *ADO.NET Entity Data Model*. Následně se vybere možnost *Code First from database*. Poté se nastaví fyzické připojení na server, a vygeneruje se *connection string* řetězec. Dalším krokem je okno s volbami, které součásti databáze se mají namodelovat. Volba *Code First* principu je volena především pro *usnadnění vytváření modelu* entit, o kterém pojednává následující kapitola. Toto usnadnění v současné době *není realizovatelné* z technik *Design First* a *Database First*. Po odsouhlasení dialogového okna se vygeneruje potomek třídy *DbContext*, který specifikuje kolekce modelovaných entit, a jejich vlastnosti a vazby (pokud nejsou vyjádřeny atributově u daných entit). Vizualizace popsaného postupu viz obrázek 7 [41].



Obrázek 7 Vytvoření datového modelu pomocí Entity Framework

6.2.2 Model entit

Model entit slouží k definici entit služby a jejich vazeb, akcí, metod, apod. Z popisu je zřejmé, že tento model slouží jako zdroj při generování CSDL dokumentu služby. Model je reprezentovaný pomocí rozhraní *IEdmModel*.

K vytvoření modelu slouží třída *ODataModelBuilder*. Obsahuje mimo jiné metody *AddEntityType*, *AddEntitySet*, *AddProcedure*, apod. Každá entita se konfiguruje manuálně. U složitých modelů je tento proces poměrně pracný. Nabízí však maximální kontrolu nad výslednou podobou CSDL dokumentu.

Model služby může být vytvořen i jednodušším způsobem, a to pomocí třídy *ODataConventionModelBuilder*, která od třídy ODataModelBuilder dědí. Tato třída dokáže vygenerovat model na základě tříd konceptuálního modelu Entity Framework Code First techniky. Pokud tedy pro komunikaci s podkladovým datovým zdrojem používáme Entity Framework, nabízí tato možnost značné ulehčení práce. Použití třídy by mohlo mít podobu

```
ODataConventionModelBuilder builder = new ODataConventionModelBuilder();
builder.EntitySet<Category>("Categories");
builder.EntitySet<Contact>("Contacts");
builder.EntitySet<ContactType>("ContactTypes");
builder.EntitySet<Customer>("Customers");
var os = builder.EntitySet<Order>("Orders");
var o = builder.EntityType<Order>();
o.Collection.Function("GetTotalCost").Returns<double>()
    .Parameter<int>("orderId");
o.Collection.Action("AddOrderItem")
    .Parameter<AddOrderItemModel>("addOrderItemModel");
builder.EntitySet<OrderItem>("OrderItems");
builder.EntitySet<StoreItem>("StoreItems");
```

K zahrnutí entity do modelu je nutné pouze nadefinovat název kolekce entity daného typu, a poskytnout datový typ, který splňuje konvence pro sestavení modelu. Entity, jež jsou poskytnuty jako generické argumenty metody, jsou třídy vygenerované pomocí nástroje Entity Framework. Dále jsou demonstrovány příkazy pro deklaraci akce a funkce. Jejich vysvětlení bude uvedeno v kapitole 6.5. Jedná se o procedury demonstrované v kapitolách 2.2.5 a 2.2.6 [41].

6.2.3 Provedení registrace služby

Po vytvoření modelu je již možné zavolat metodu MapODataServiceRoute. Tato metoda má mnoho přetížených forem. Nejjednodušší je využít formu

```
public static ODataRoute MapODataServiceRoute(this HttpConfiguration
    configuration, string routeName, string routePrefix, IEdmModel model);
```

kde *routeName* je jméno routy, která se použije při rozhodování o směrování požadavku na daný kontrolér a akci, *routePrefix* je volitelný řetězec, který je v případě vyplnění nutné

připsat do URL požadavku, a *model* je již zmiňovaný model služby, ze kterého se generují metadata. Příkladem volání by mohl být kód

```
config.MapODataServiceRoute("odata", "odata", builder.GetEdmModel());
```

V takovém případě musí být všechny požadavky zasílány na adresu začínající řetězcem *[adr]/odata/*, kde *[adr]* je adresa kořene aplikace. Jak je z popisu signatury metody patrné, při registraci služby se rovněž definují pravidla směrování požadavků. Implicitně jsou použity výchozí směrovací konvence, které zahrnují základní tvary URL pro různé typy požadavků. Bohužel tyto implicitní konvence nepokrývají všechny scénáře reálných případů v praxi. Například nejsou definované směrovací pravidla pro entity s kompozitním klíčem. Více o implicitních konvencích viz zdroj [42].

6.3 Kontroléry

Kontrolér je třída, která zpracovává vlastní HTTP požadavky. Každá službou poskytovaná entita má svůj vlastní kontrolér. Konkrétní kontrolér pro zpracování konkrétního požadavku je nalezen dle směrovacích pravidel služby. Kontrolér entity Customer by mohl mít podobu

```
namespace Server.Controllers
{
    public class CustomersController : ODataController
    {
        private WebApiContext db = new WebApiContext();
    }
}
```

Název kontroléru musí končit povinnou konvencí, a to řetězcem *Controller*. Prefix názvu kontroléru má dle povinné konvence tvar názvu kolekce *EntitySet,* která byla pro entitu Customer definovaná v kapitole 6.2.2, tedy *Customers.* Dále je zvykem, že jsou kontroléry definovány ve jmenném prostoru *Controllers*.

Tento kontrolér bude zpracovávat požadavky, jež začínají adresou *[adr]/odata/Customers/*. Rovněž je v kontroléru definována proměnná typu WebApiContext, což je potomek třídy DbContext, jež byl vygenerován nástrojem Entity Framework. Tato instance bude sloužit ke komunikaci s podkladovým datovým zdrojem [41].

6.4 Akce kontroléru

Kontroléry jednotlivých entit sami o sobě neprovádějí žádné operace. Slouží pouze jako vstupní brána pro danou entitu. Pro definici operací slouží akce kontroléru. Každá akce

definuje jednu operaci. Akce jsou metody třídy dědící od třídy *ODataController*. Dle názvu akce je pomocí konvence odvozena metoda HTTP požadavku, která ji může aktivovat. To znamená, že např. akci s názvem Put je možné vyvolat pouze požadavkem s metodou PUT. Je však možné povolit i jiné HTTP metody pomocí speciálního atributu uvedeného v další části textu. Konkrétní akce pro obsloužení požadavku je vybrána na základě směrovacích pravidel služby.

Akce je možné psát i jako asynchronní, u implementované služby budou uvedeny jejich synchronní verze.

6.4.1 Akce pro vytváření dat

Pro vytváření entity slouží metoda se signaturou *Post*. Pro entitu typu *Customer* má akce tvar

```
public IHttpActionResult Post(Customer customer)
{
    if (!ModelState.IsValid) return BadRequest(ModelState);
    db.Customers.Add(customer);
    db.SaveChanges();
    return Created(customer);
}
```

Akce vrací instanci třídy *HttpResponseMessage* skrze třídu, jež implementuje rozhraní *IHttpActionResult*. Toto rozhraní předepisuje metodu odpovědnou za asynchronní vytvoření instance HttpResponseMessage.

Argumentem Akce je instance entity Customer, která je do akce vložena pomocí model bindingu, který byl popsán v kapitole 3.1.3.

Prvním krokem akce je validace přijatého modelu. Třída *ApiController* (od které třída ODataController dědí) definuje vlastnost *ModelState*, která obsahuje vlastnost *IsValid*. Tato vlastnost se automaticky inicializuje po procesu model bindingu. V případě, že model není validní (chybějící hodnota vlastnosti entity, apod.), je pomocí metody *BadRequest* třídy ApiController vrácena instance typu HttpResponseMessage s chybou 400.

Jinak je do podkladového datového zdroje skrze instanci typu DbContext vložena nová entita, a změny v databázi jsou uloženy. Následně služba odpoví pomocí metody *Created* třídy ODataController zprávou s kódem 201 a vytvořenou entitou.

Jak bylo uvedeno v kapitole 2.2.1, OData protokol obsahuje definici, jež umožňuje definovat svázané entity již při procesu vytváření entity. Děje se tak definováním položky

@odata.bind v požadavku klienta. Tato *funkcionalita* ovšem *není v současné době* knihovnou *podporována*.

Pro vytváření, případně editaci referencí dané entity slouží akce s názvem *CreateRef*. Pro entitu typu *Customer* může mít akce podobu

```
[AcceptVerbs("POST")]
public IHttpActionResult CreateRef([FromODataUri] int key,
    string navigationProperty, [FromBody] Uri link)
{
    var customer = db.Customers.SingleOrDefault(c => c.Id == key);
    if (customer == null) return NotFound();
    switch (navigationProperty)
    {
        case "Contacts":
            var relatedKey = Helpers.GetKeyFromUri<int>(Request, link);
            var contact = db.Contacts.SingleOrDefault(c =>
                c.Id == relatedKey);
            if (contact == null) return NotFound();
            customer.Contacts.Add(contact);
            break;
        case "Orders":
            // ...
        default:
            return StatusCode(HttpStatusCode.NotImplemented);
    }
    db.SaveChanges();
    return StatusCode(HttpStatusCode.NoContent);
}
```

Akce je označena atributem *AcceptVerbs* s argumentem *POST*. Tento atribut určuje podmínku požadované metody požadavku pro jeho zpracování. Jelikož má entita Customer pouze navigační vlastnosti typu kolekce entit, je postačující přijímat pouze požadavky typu POST.

Názvy argumentů akce *nejsou nahodilé*, nýbrž je jejich tvar povinný dle implicitních směrovacích pravidel. Argument *key* slouží pro model binding identifikátoru entity, argument *navigationProperty* pro model binding názvu vazby mezi entitami, a argument *link* pro model binding identifikátoru navigační entity.

Zároveň je pomocí atributů argumentů specifikováno, z jaké části požadavku se má model binding provést. Atribut *FromODataUri* specifikuje, že model binding se provádí z URL požadavku klienta, zatímco atribut *FromBody* specifikuje, že se má binding provádět z obsahu těla požadavku.

Dle argumentu key je nalezena entita typu Customer. Dále je provedena modifikace konkrétní vazby dle argumentu navigationProperty. Demonstrován je případ pro vazbu

s názvem *Contacts*. Nejdříve je z argumentu link vypreparován klíč svázané entity pomocí metody *GetKeyFromUri*. Pro podobu této metody viz příloha PIII. Poté je nalezená entita typu Contact přidána do navigační kolekce Contacts entity Customer. Při úspěšném vyřízení požadavku vrací akce kód 204, v opačném případě kód 404 (neznámá entita) nebo kód 501 (neznámá vazba).

Kód pro vytvoření, případně editaci vazby typu entita by byl velice podobný. Pro navigační vazbu *ParentCategory* entity *Category* může mít akce tvar

```
[AcceptVerbs("POST", "PUT")]
public IHttpActionResult CreateRef([FromODataUri] int key,
    string navigationProperty, [FromBody] Uri link)
{
    var category = db.Categories.SingleOrDefault(c => c.Id == key);
    if (category == null) return NotFound();
    switch (navigationProperty)
    {
        case "ParentCategory":
            var relatedKey = Helpers.GetKeyFromUri<int>(Request, link);
            var parentCategory = db.Categories.SingleOrDefault(c =>
                c.Id == relatedKey);
            if (parentCategory == null) return NotFound();
            category.ParentCategory = parentCategory;
            break;
        case "ChildCategories":
        // ...
        case "StoreItems":
        // ...
        default:
            return StatusCode(HttpStatusCode.NotImplemented);
    }
    db.SaveChanges();
    return StatusCode(HttpStatusCode.NoContent);
}
```

U této akce je v atributu AcceptVerbs definována i metoda PUT, jelikož vazba ParentCategory je typu entita [41][43].

6.4.2 Akce pro editaci dat

Jak bylo uvedeno v kapitole 2.2.2, editaci entity je možné provést dvojím způsobem. Prvním způsobem je využití částečné úpravy pomocí akce s názvem *Patch*. V takovém případě se odesílají pouze změněné vlastnosti entity. Kód pro entitu typu *Customer* má tvar

```
[AcceptVerbs("PATCH", "MERGE")]
public IHttpActionResult Patch([FromODataUri] int key, Delta<Customer> patch)
{
    if (!ModelState.IsValid) return BadRequest(ModelState);
    Customer customer = db.Customers.Find(key);
    if (customer == null) return NotFound();
    patch.Patch(customer);
```

```
        try
        {
            db.SaveChanges();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!CustomerExists(key)) return NotFound();
            else throw;
        }
        return Updated(customer);
    }
```

Akci je možné aktivovat pomocí HTTP metod *PATCH* a *MERGE*. Je to proto, že v originální specifikaci HTTP 1.1 není definována žádná metoda pro „částečnou editaci“. OData specifikace proto definovala metodu MERGE. V roce 2010 však byla v dokumentu RFC 5789 definována metoda PATCH. Doporučenou metodou je metoda PATCH. Z důvodu zvýšení interoperability jsou však definovány obě metody.

Akce obsahuje argumenty pro klíč editované entity a argument *patch*. Tento argument je typu *Delta*, a obsahuje informace o změněných vlastnostech entity.

V případě, že je model binding úspěšný, a entita je nalezena dle klíče, se provede částeční editace entity pomocí metody *Patch* třídy Delta. Následně jsou data uložena. V akci je zařazena rovněž kontrola vůči souběhu. V případě, že byla entita v průběhu vykonávání akce odstraněna, je vrácena chyba 404. Obsah metody *CustomerExists* je triviální, obsahuje pouze kontrolu databázového kontextu na přítomnost entity:

```
    private bool CustomerExists(int key)
    {
        return db.Customers.Count(e => e.Id == key) > 0;
    }
```

Po úspěšné editaci entity je službou vrácena zpráva s kódem 204.

Druhou možností editace entity je úplná editace. Korespondující akce nese název *Put*, a její tvar má podobu

```
    public IHttpActionResult Put([FromODataUri] int key, Customer customer)
    {
        if (!ModelState.IsValid) return BadRequest(ModelState);
        if (key != customer.Id) return BadRequest();
        db.Entry(customer).State = EntityState.Modified;
        try
        {
            db.SaveChanges();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!CustomerExists(key)) return NotFound();
            else throw;
```

```
        }
        return Updated(customer);
    }
```

Vyvolává se HTTP metodou PUT. Argumenty sestávají z klíče entity a vlastní entity. V případě, že je model validní, a argument key se shoduje s klíčem entity, je provedena vlastní editace. Ta se provede pomocí metody *Entry* třídy DbContext, která vrací instanci typu *DbEntityEntry*, u které se nastaví vnitřní stav na hodnotu *Modified*. V takovém případě se při volání metody db.SaveChanges vygeneruje pro danou entitu příkaz UPDATE s novými hodnotami vlastností entity. V případě, že nedojde k souběhu, je službou vrácen kód 204 [41].

6.4.3 Akce pro odstraňování dat

Odstranění entity je zpřístupněno pomocí akce s názvem *Delete*. Její podoba pro entitu typu *Customer* má tvar

```
public IHttpActionResult Delete([FromODataUri] int key)
{
    Customer customer = db.Customers.Find(key);
    if (customer == null) return NotFound();
    db.Customers.Remove(customer);
    db.SaveChanges();
    return StatusCode(HttpStatusCode.NoContent);
}
```

Akce obsahuje argument *key*, pomocí kterého se nalezne požadovaná entita. V případě, že je nalezena, je provedeno její odstranění z podkladového datového zdroje, a službou je navrácena zpráva kódem 204. V opačném případě je vrácena zpráva s kódem 404.

Odstraňování reference v navigační vlastnosti entity je zprostředkováno pomocí akce s názvem *DeleteRef*. Reference může být odstraněna z navigační vlastnosti typu entita, nebo typu kolekce entit. Každou variantu zpracovává samostatná akce. Pro vlastnost *ParentCategory* entity *Category* má akce tvar

```
public IHttpActionResult DeleteRef([FromODataUri] int key,
    string navigationProperty)
{
    var category = db.Categories.SingleOrDefault(c => c.Id == key);
    if (category == null) return NotFound();
    switch (navigationProperty)
    {
        case "ParentCategory":
            category.ParentCategoryId = null;
            break;
        default:
            return StatusCode(HttpStatusCode.NotImplemented);
    }
```

```
    db.SaveChanges();
    return StatusCode(HttpStatusCode.NoContent);
}
```

Akce obsahuje argumenty *key* a *navigationProperty*. Pokud je entita dle klíče nalezena, a navigační vlastnost je rozpoznána, nastaví se hodnota cizího klíče entity na hodnotu null. Následně jsou změny kontextu uloženy, a službou je navrácena zpráva s kódem 204.

Akce pro odstranění reference z navigační vlastnosti typu kolekce entit pro entitu typu *Category* má tvar

```
public IHttpActionResult DeleteRef([FromODataUri] int key,
    [FromODataUri] string relatedKey, string navigationProperty)
{
    var category = db.Categories.SingleOrDefault(c => c.Id == key);
    if (category == null) return StatusCode(HttpStatusCode.NotFound);
    switch (navigationProperty)
    {
        case "ChildCategories":
            var childCategoryId = Convert.ToInt32(relatedKey);
            var childCategory = db.Categories.SingleOrDefault(c =>
                c.Id == childCategoryId);
            if (childCategory == null) return NotFound();
            childCategory.ParentCategoryId = null;
            break;
        case "StoreItems":
        // ...
        default:
            return StatusCode(HttpStatusCode.NotImplemented);
    }
    db.SaveChanges();
    return StatusCode(HttpStatusCode.NoContent);
}
```

Přibyl zde argument *relatedKey*. Tento argument je klíčem svázané entity, a je pomocí implicitních směrovacích pravidel automaticky vypreparován z URL požadavku klienta.

V akci je kód pro vlastnost *ChildCategories*. V případě, že jsou nalezeny obě vazební entity, je u podřízené kategorie odstraněna reference na její nadřízenou kategorii. Poté je službou vrácena zpráva s kódem 204 [41][43].

6.4.4 Akce pro čtení dat

Ke čtení dat slouží akce s prefixem *Get*. Číst lze kolekci entity daného typu, konkrétní entitu, vlastnost entity a navigační vazbu entity. Akce pro čtení kolekci entity typu *Customer* má tvar

```
public IQueryable<Customer> GetCustomers()
{
    return db.Customers;
```

```
}
```

Akce zde nevrací instanci třídy HttpResponseMessage, nýbrž *IQueriable<Customer>* kolekci. V těle akce je pouze volána vlastnost Customers třídy WebApiContext. Tato vlastnost je typu *Dbset<Customer>* implementujícího rozhraní *IQueriable<Customer>*.

Akce pro čtení konkrétní entity typu *Customer* má tvar

```
public SingleResult<Customer> GetCustomer([FromODataUri] int key)
{
    return SingleResult.Create(db.Customers.Where(
        customer => customer.Id == key));
}
```

Návratovým typem akce je typ *SingleResult<Customer>*. Tento typ reprezentuje kolekci IQueriable s nejvíce jedním prvkem. V těle akce je vytvořena instance typu SingleResult pomocí statické metody *Create* třídy SingleResult. Argumentem metody je potencionálně nalezená entita typu Customer dle argumentu akce s názvem *key*.

Pro čtení vlastností entity je nutné pro každou vlastnosti definovat samostatnou akci. Akce pro vlastnost *Firstname* entity typu *Customer* má tvar

```
public IHttpActionResult GetFirstname(int key)
{
    var customer = db.Customers.Find(key);
    if (customer == null) return NotFound();
    return Ok(customer.Firstname);
}
```

Akce vrací instanci třídy HttpResponseMessage. V případě, že je entita nalezena dle argumentu akce s názvem *key*, je vrácena zpráva s kódem 200, a obsahem zprávy ve tvaru požadované vlastnosti. Jinak je vrácena zpráva s kódem 404.

Pro čtení navigačních vlastností entity je rovněž nutné definovat pro každou navigační vlastnost samostatnou akci. Akce pro navigační vlastnost *Contacts* entity *Customer* má tvar

```
public IQueryable<Contact> GetContacts([FromODataUri] int key)
{
    return db.Customers.Where(m => m.Id == key).SelectMany(m => m.Contacts);
}
```

6.4.5 Povolení Query Options při čtení dat

Povolení pro klientské dotazovací operátory se musí udávat *explicitně*. Pokud je potřebné povolit dotazování nad celou službou, je možné využít metodu *EnableQuerySupport* třídy *HttpConfiguration*. Tato metoda by se volala v metodě WebApiConfig.Register, která byla zmíněna v kapitole 6.1.

V případě, že je potřebné povolit klientské dotazovací operátory pouze pro určité akce, použije se atribut s názvem *EnableQuery*. Povolení dotazů pro kolekci entit typu Customer by mělo tvar

```
[EnableQuery]
public IQueryable<Customer> GetCustomers() { … }
```

Tento atribut je možné přidat pro jakoukoliv akci, jenž má návratovou hodnotu typu *IQueriable*, nebo *SingleResult*.

Atribut obsahuje mnoho vlastností, které umožňují modifikovat možnosti klientských dotazů nad danou akcí. Přehled vybraných vlastností viz tabulka 7.

Vlastnost	Význam, příklad použití
AllowedArithmeticOperators	Povolené aritmetické operátory
AllowedFunctions	Povolené funkce
AllowedLogicalOperators	Povolené logické operátory
AllowedQueryOptions	Povolené dotazovací příkazy
MaxExpansionDepth	Maximální hloubka vnoření skrze navigační vlastnosti
PageSize	Počet záznamů na stránku při Server-Driven Paging

Tabulka 7 Vybrané vlastnosti atributu EnableQuery [45]

Omezení možností klientských dotazů skrze vlastnosti atributu EnableQuery jsou vhodné pro fixní omezení, které nezávisí např. na autentizaci a autorizaci klienta. Alternativní způsob omezení dotazů je uveden v kapitole 6.7.2 [41][44][45].

6.5 Akce a funkce

Akce a funkce se zapisují jako metody kontroléru. Jejich struktura musí být explicitně uvedena při vytváření modelu entit, a názvy/datové typy procedury a jejich argumentů se musí shodovat.

Akci *AddOrderItem*, deklarovanou v kapitole 6.2.2, bohužel nelze v dané formě implementovat. Knihovna ASP.NET Web Api OData totiž v současné době *nepodporuje komplexní typy jako argumenty funkcí a akcí*. Jednotlivé vlastnosti komplexního typu *AddOrderItemModel* tedy budou tvořit argumenty akce. Deklarace akce v modelu entit má nově tvar

```
var os = builder.EntitySet<Order>("Orders");
var o = builder.EntityType<Order>();
var a = o.Collection.Action("AddOrderItem");
a.Parameter<int>("orderId");
a.Parameter<int>("orderItemId");
```

Korespondující metoda kontroléru *OrdersController* má tvar

```
[HttpPost]
public IHttpActionResult AddOrderItem(ODataActionParameters parameters)
{
    var order = db.Orders.Find(parameters["orderId"]);
    var orderItem = db.OrderItems.Find(parameters["orderItemId"]);
    if (order == null || orderItem == null) return NotFound();
    orderItem.OrderId = order.Id;
    db.SaveChanges();
    return StatusCode(HttpStatusCode.Created);
}
```

Akce je označena atributem *HttpPost*. Tento atribut je ekvivalentem atributu *AcceptVerbs("POST")*. Pro pole primitivních argumentů je možné použít třídu *ODataActionParameters*. Je to slovník, který se automaticky inicializuje z těla požadavku klienta. K jednotlivým položkám se poté přistupuje pomocí indexace.

Funkce *GetTotalCost* měla v modelu entit podobu

```
o.Collection.Function("GetTotalCost").Returns<double>()
    .Parameter<int>("orderId");
```

Odpovídající metoda v kontroléru *OrdersController* má tvar

```
[HttpGet]
public IHttpActionResult GetTotalCost([FromODataUri] int orderId)
{
    Order order = db.Orders.Find(orderId);
    if (order == null) return NotFound();
    var items = order.OrderItems.Select(x => x.StoreItem.Price * x.Count);
    if (items.Count() == 0) return Ok(0);
    return Ok(items.Sum());
}
```

Jelikož tato akce reprezentuje funkci, obsahuje atribut *HttpGet*, což je zkrácená forma atributu *AcceptVerbs("GET")* [46].

6.6 Automatické generování kontroléru a akcí

Kontrolér a jeho akce je možné vygenerovat automatickým způsobem. Toto automatické generování se nazývá *Scaffolding*. Většina akcí má generickou podobu, a jejich podoba je u různých kontrolérů víceméně stejná.

Pro přidání automaticky generovaného kontroléru je potřebné otevřít kontextovou nabídku složky *Controllers*, kliknout na tlačítko *New*, a následně na tlačítko *Controller*. V dialogovém okně se poté na *záložce Controller* nachází položka Web Api 2 Odata Controller with actions, using Entity Framework.

Po výběru této položky je v dalším dialogovém okně nutné vybrat modelovací třídu (třída vygenerovaná nástrojem Entity Framework), třídu datového kontextu, a název kontroléru. Rovněž je možné zaškrtnout políčko *Use async controller actions*, které vytvoří akce jako asynchronní.

Po odsouhlasení okna se vytvoří kontrolér i s akcemi. Automaticky jsou vygenerované akce GetEntities, GetEntity, Put, Post, Patch, Delete, a akce GetNavigationProperty pro každou navigační vlastnost entity. Případné další akce je nutné dopsat manuálně.

Vygenerované akce jsou vytvořeny podle konvence, tzn. argument *key* v akcích je vždy velikosti jedna, i když je například primární klíč entity kompozitní, apod. Případné nesrovnalosti je potřebné manuálně upravit [47].

6.7 Omezení přístupu ke službě na základě autentizace a autorizace

Po implementaci infrastruktury, která na základě identifikačních údajů klienta přiřadí kontextu požadavku jeho informace, je možné tyto informace využít k vlastní autentizaci a autorizaci. K informacím o klientovi lze v kontroléru přistupovat pomocí vlastnosti *User*. User je typu *IPrincipal*, a obsahuje vlastnost *Identity.IsAuthenticated* pro informaci o autentizaci. Dále obsahuje metodu *IsInRole*, kterou je možné provést autorizaci.

6.7.1 Omezení přístupu

Omezení přístupu ke kontroléru nebo akci se provede pomocí atributu s názvem *Authorize.* Příklad použití pro kontrolér entity *Customer* může mít tvar

```
[Authorize]
public class CustomersController : ODataController
{
    [AllowAnonymous]
    public SingleResult<Customer> GetCustomer([FromODataUri] int key) { … }

    [Authorize(Roles = "Customers, Admins")]
    public IHttpActionResult Delete([FromODataUri] int key) { … }
}
```

V ukázce je přístup ke kontroléru povolen pouze pro autentizované klienty. Pro přístup k akci *Delete* je navíc nutné být v roli *Customers* nebo *Admins*. Přístup k akci *GetCustomer* je naopak povolený i neautentizovanému uživateli díky atributu *AllowAnonymous*.

6.7.2 Omezení dotazů

Na základě autentizace a autorizace je rovněž možné omezit dotazy a operace, které může klient provést nad poskytovanou službou. Validace se provede pomocí metody *Validate* třídy *ODataQueryOptions*. Tato třída obsahuje informace o dotazech a funkcích použitých v požadavku klienta. V případě, že je instance této třídy požadována jako argument akce, se při model bindingu automaticky inicializuje z klientského dotazu. Příklad využití pro akci *GetCustomers* třídy *CustomersController* může mít tvar

```
[EnableQuery]
public IQueryable<Customer> GetCustomers(ODataQueryOptions options)
{
    var settings = GetValidationSettings();
    options.Validate(settings);
    return db.Customers;
}
```

Metoda *Validate* požaduje instanci třídy *ODataValidationSettings*, která předepisuje povolené dotazovací operátory a funkce pro daný požadavek klienta. Příklad rozhodnutí o možnostech klienta může mít tvar

```
private ODataValidationSettings GetValidationSettings()
{
    if (User.IsInRole("Admins")) return new ODataValidationSettings();
    return new ODataValidationSettings()
    {
        AllowedLogicalOperators = AllowedLogicalOperators.None,
        AllowedQueryOptions = AllowedQueryOptions.None
    };
}
```

V případě, že je klient služby autentizovaný, a zároveň patří do skupiny Admins, má práva na všechny dotazovací operátory a funkce. V opačném případě je mu zakázáno používat dotazovací operátory a logické operátory. Možnosti omezení dotazů korespondují s vlastnostmi atributu EnableQuery prezentované v kapitole 6.4.5.

Metoda Validate třídy ODataQueryOptions vyvolá výjimku typu *ODataException*, pokud klient požaduje funkcionality, na které nemá dostatečné oprávnění [45][48].

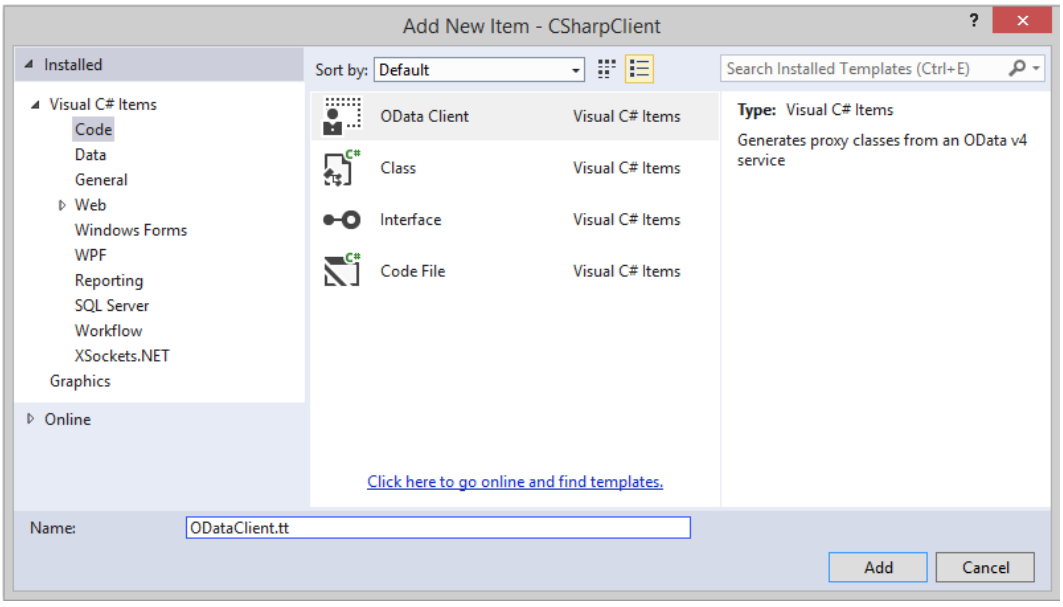
7 IMPLEMENTACE KLIENTA SLUŽBY

Pro konzumaci OData služeb je potřebné mít nainstalované bezplatné rozšíření nástroje Visual Studio s názvem *OData v4 Client Code Generator*[10]. Toto rozšíření slouží ke generování silně typového klienta služby. Poklady pro generování tříd jsou čerpány z CSDL dokumentu služby.

Vlastní knihovna klienta existuje ve jmenném prostoru *Microsoft.OData.Client*. Tato knihovna je založena na knihovně *WCF Data Services Client Library* (kterou je možné využít pro konzumaci OData služeb do verze 3), na kterou navazuje. Na domovských stránkách knihovny (viz [49]) je možné nalézt její zdrojový kód, a odkaz na dokumentaci, který vede na již zmiňovanou knihovnu WCF Data Services Client Library [49].

7.1 Vytvoření a inicializace OData klienta

Přidání OData klienta je možné provést pomocí pravého kliknutí na název projektu v kartě *Solution Explorer*. V kontextové nabídce se zvolí položky *Add*, následně *New Item*. Poté se v kategorii *Code* zvolí možnost *OData Client*. Klient se pojmenuje, a potvrdí tlačítkem *Add*. Vizualizace postupu viz obrázek 8.



Obrázek 8 Vytvoření klienta OData služby

[10] Rozšíření je možné nalézt na adrese https://visualstudiogallery.msdn.microsoft.com/9b786c0e-79d1-4a50-89a5-125e57475937

Následně je potřebné vyplnit kořenovou adresu OData služby. Klient byl pojmenován názvem *ODataClient*. V kartě Solution Explorer přibyl soubor *ODataClient.tt*. Po otevření souboru je třeba vyhledat řetězec

```
public const string MetadataDocumentUri = "";
```

a hodnotu konstanty *MetadataDocumentUri* nahradit danou adresou. Pro službu implementovanou v kapitole 6 hostovanou na lokálním serveru má adresa tvar

```
public const string MetadataDocumentUri = "http://localhost:50921/odata";
```

Při ukládání změn se klient pokusí automaticky vygenerovat silně typované třídy pro komunikaci se serverem. Služba musí být v danou dobu dostupná, aby klient mohl číst její metadata. Následně se do souboru *ODataClient.cs* vygeneruje vlastní klient služby. V tuto chvíli je klient připraven k použití.

7.1.1 Vytvoření kontextu

Komunikačním prostředkem mezi klientskou aplikací a datovou službou je třída dědící od třídy *DataServiceContext*. Tato třída je automaticky vygenerovaná, a její jméno a jmenný prostor závisí na metadatech služby. V případě metadat z přílohy PI bude třída nést název *Container* (dle atributu *Name* elementu *EntityContainer*), a bude se nacházet ve jmenném prostoru *Default* (dle atributu *Namespace* nadřazeného elementu *Schema*).

DataServiceContext je na rozdíl od služby, se kterou komunikuje, stavový. Změny prováděné klientem se v kontextu ukládají, a až po zavolání konkrétní metody jsou provedené modifikace odeslány na server.

Vytvoření instance třídy *Container* má podobu

```
Default.Container container = new Default.Container(
    new Uri("http://localhost:50921/odata/"));
```

Jako argument konstruktoru je požadována instance třídy *Uri* s adresou na kořen OData služby.

7.1.2 Ukládání změn

Ukládání změn provedených klientem se provádí pomocí metody *SaveChanges* třídy *DateServiceContext*. Tato metoda vrací instanci třídy *DataServiceResponse*, jež obsahuje sekvenci typu *OperationResponse*. Tato sekvence obsahuje odpovědi serveru pro každý požadavek, jenž musel být odeslán pro provedení změn uložených v kontextu [50][51].

7.2 Vytváření dat

7.2.1 Vytváření entit

Pro vytváření entit slouží speciálně vygenerované metody ve třídě *Container*. Tyto metody začínají prefixem *AddTo*, následovaným názvem dané entity. Vytvoření nové entity typu Customer může mít tvar

```
Customer customer = new Customer()
    { Firstname = "Ross", Lastname = "Geller" };
client.container.AddToCustomers(customer);
client.container.SaveChanges();
```

V aplikaci byla instance typu *Container* uložena do pomocné třídy, jejíž instance nese název *client*. Nejdříve je vytvořena nová instance typu *Customer*. Následně je vložena do kontextu skrze argument metody *AddToCustomers*. Tato metoda označí danou entitu stavem *Added*. Nakonec jsou změny v kontextu uloženy. Po uložení změn jsou případné serverem generované sloupce entity (např. hodnota identifikátoru) automaticky aktualizovány u entity uložené v kontextu.

7.2.2 Vytváření referencí

Pro vytváření referencí pro navigační vazbu typu kolekce entit slouží metoda *AddLink* třídy *DataServiceContext*. Využití pro vazbu *Orders* entity *Customer* může mít tvar

```
Customer customer; // customer = ...;
Order order; // order = ...;
client.container.AddLink(customer, "Orders", order);
client.container.SaveChanges();
```

Proces získání referencí na instance tříd Customer a Order zde prozatím není uvedený. Metoda AddLink vyžaduje tři argumenty v pořadí entita obsahující navigační vazbu typu kolekce, název navigační vazby a entita vkládaná do vazby. Entita obsahující navigační vazbu nesmí být v době provádění příkazu ve stavu Added.

Vytváření referencí pro navigační vazbu typu entita je realizovatelné pomocí metody *SetLink* třídy *DataServiceContext*. Využití pro vazbu *ParentCategory* entity *Category* může mít tvar

```
Category parentCategory, childCategory;
// parentCategory = ...; childCategory = ...;
client.container.SetLink(childCategory, "ParentCategory", parentCategory);
client.container.SaveChanges();
```

Metoda SetLink opět vyžaduje tři argumenty, a to již existující entitu obsahující navigační vazbu typu entita, název vazby, a entitu vkládanou do vazby.

Metody AddLink a SetLink nenastavují vlastní hodnoty cizích klíčů nebo navigačních vlastností entit v kontextu, a to ani po uložení změn.

Vytváření referencí mezi entitami je možné provést i jiným způsobem. Výsledkem přidání reference v navigační vlastnosti je nastavení cizího klíče závislé entity. Toto natavení hodnoty cizího klíče je možné provést i manuálně pomocí editace entity [50][52].

7.3 Editace dat

7.3.1 Editace entit

Editace entity je realizována pomocí metody *UpdateObject* třídy *DataServiceContext*. Editace vlastnosti *Firstname* entity *Customer* může mít podobu

```
Customer customer; // customer = ...;
customer.Firstname = "Walter";
client.container.UpdateObject(customer);
client.container.SaveChanges();
```

Tato metoda sdělí kontextu, že změny provedené nad danou entitou se mají projevit i v datové službě. Interně je stav dané entity změněn na stav *Modified*. Tato metoda způsobí odeslání požadavku typu PATCH.

7.3.2 Editace referencí

Editaci referencí je možné provést stejným způsobem, který je popsán v kapitole 7.2.2 pro jejich vytváření. Rovněž je možné použít přímé nastavení hodnoty cizího klíče dané entity, např.

```
Category category; // category = ...;
category.ParentCategoryId = 2;
client.container.UpdateObject(category);
client.container.SaveChanges();
```

Tyto příkazy způsobí vytvoření požadavku typu PATCH, který nastaví cizí klíč na požadovanou hodnotu [52].

7.4 Odstraňování dat

7.4.1 Odstraňování entit

Pro odstraňování entit slouží metoda *DeleteObject* třídy *DataServiceContext*. Příklad použití pro entitu typu *Category* může mít tvar

```
Category category; // category = ...;
client.container.DeleteObject(category);
client.container.SaveChanges();
```

Metoda *DeleteObject* interně změní entitu do stavu *Deleted*. Tato metoda neprovádí automatické odstraňování cizích klíčů odkazujících na odstraňovanou entitu.

7.4.2 Odstraňování referencí

Odstraňování referencí v navigační vazbě typu kolekce entit je možné provádět pomocí metody *DeleteLink* třídy *DataServiceContext*. Využití pro navigační vlastnost *Orders* entity *Customer* může mít tvar

```
Customer customer; // customer = ...;
Order order; // order = ...;
client. container.DeleteLink(customer, "Orders", order);
client.container.SaveChanges();
```

Pro odstraňování referencí ve vazbách typu kolekce je možné použít tutéž metodu, kterou lze využít při jejich vytváření. Jediným rozdílem je, že místo cílové entity vkládané do vazby se uvede explicitně hodnota *null*. Využití pro entitu *Category* může mít tvar

```
Category category; // category = ...;
client.container.SetLink(category, "ParentCategory", null);
client.container.SaveChanges();
```

Reference je rovněž možné odstranit manuálním nastavením cizích klíčů na hodnotu *null*, a následným označením entit metodou UpdateObject [52].

7.5 Čtení dat

Čtení dat je prováděno na základě dotazů vytvořených jako instance třídy *DataServiceQuery<T>*. Z této instance je vygenerován HTTP GET dotaz, který je odeslán na server. Z odpovědi jsou poté vytvořeny instance požadovaných entit, které jsou vloženy do kontextu klienta.

Třída DataServiceQuery<T> dědí od rozhraní *IQueryable<T>*, kde *T* je typ entity. To znamená, že pro provádění dotazů je možné použít nástroje *LINQ*.

DataServiceQuery využívá techniku zvanou *Lazy Loading*. To znamená, že vytvořený dotaz není odeslán na server okamžitě, nýbrž je odeslán až v okamžiku potřeby. Odeslání dotazu je provedeno v těchto případech:

- Dotaz je *enumerován* (např. použití v cyklu *foreach*).
- Dotaz je použit pro inicializaci kolekce typu *List*.
- Je explicitně zavolán příkaz *Execute* nebo asynchronní *ExecuteAsync*.
- Je použit LINQ operátor, který požaduje existující kolekci (např. *First*, *Single*).

7.5.1 Čtení kolekce entit

Pro čtení entit byly pomocí nástroje OData v4 Client Code Generator vygenerovány vlastnosti typu DataServiceQuery pro všechny entity poskytované službou. Tyto vlastnosti patří do třídy Container. Příklad čtení z kolekce typu *Customer* může mít tvar

```
IQueryable<Customer> customers = client.container.Customers;
foreach (var customer in customers) { ... }
```

Dotaz na server je odeslán teprve ve druhém řádku kódu.

7.5.2 Čtení entit

Ke čtení konkrétní entity slouží statická metoda *ByKey*. Tato metoda patří do třídy *ExtensionsMethods*, která byla rovněž vytvořena pomocí generátoru klienta. Pro každou entitu je k dispozici samostatná metoda ByKey.

Tato metoda vrací instanci potomka typu *DataServiceQuerySingle<T>*. Ve třídě Container jsou potomci pro každý typ entity (*CustomerSingle*, *OrderSingle*, apod.). Tyto instance jsou vhodné rovněž pro přístup k navigačním vlastnostem dané entity, jak bude uvedeno v další části textu.

Příklad čtení entity typu Customer může mít podobu

```
Customer customer = client.container.Customers.ByKey(1).GetValue();
```

Extension metoda ByKey třídy DataServiceQuery<Customer> vrací instanci typu CustomerSingle. V této době ještě není odeslán žádný požadavek na server. Pro přístup ke konkrétní entitě typu Customer slouží metoda *GetValue*, resp. *GetValueAsync* pro asynchronní čtení.

7.5.3 Čtení vlastnosti entity

Pro čtení konkrétní vlastnosti entity slouží opět metoda ByKey v kombinaci s LINQ metodou *Select*. Použití pro vlastnost *Firstname* entity *Customer* může mít tvar

```
string firstname = client.container.Customers.ByKey(1)
    .Select(x => x.Firstname).GetValue();
```

7.5.4 Čtení navigačních vlastností entity

Čtení navigačních vlastností entity je umožněno pomocí potomků DataServiceQuerySingle daných entit. Čtení navigační vlastnosti *Orders* entity *Customer* může mít podobu

```
IEnumerable<Order> customerOrders = client.container.Customers
    .ByKey(1).Orders.Execute();
```

Jak je z kódu patrné, voláním metody Execute se z kolekce typu IQueryable stává kolekce typu IEnumerable.

7.5.5 Použití operátorů a funkcí

Použití operátorů v dotazech je možno provádět jak pomocí příkazů LINQ, tak pomocí metody *AddQueryOption* třídy DataServiceQuery.

Pro použití OData operátoru *$select* slouží LINQ metoda *Select*. Příkladem může být kód

```
var customerInfos = client.container.Customers.Select(x =>
    new { firstName= x.Firstname, Lastname = x.Lastname }).ToList()
```

Tento kód vygeneruje dotaz

```
GET http://localhost:50921/odata/Customers?$select=Firstname,Lastname
```

Metodu AddQueryOption pro tento operátor nelze použít.

Operátor *$expand* je implementován pomocí metody *Expand* třídy DataServiceQuery. Použití může mít formu

```
IEnumerable<Customer> customersWithOrdersAndContacts = client.container.
    Customers.Expand("Contacts").Expand("Orders").Execute();
```

Tento kód vygeneruje dotaz

```
GET http://localhost:50921/odata/Customers?$expand=Contacts,Orders
```

Proměnná *customersWithOrdersAndContacts* by v případě absence volání metod Expand obsahovala ve svých vlastnostech *Contacts* a *Orders* pouze prázdné kolekce.

Operátor *$orderby* je možné implementovat pomocí LINQ metod *OrderBy* a *OrderByDescending* (resp. *ThenBy* a *ThenByDescending*). Příkladem použití může být dotaz

```
var orders = client.container.Orders.OrderBy(x => x.Created)
    .ThenByDescending(x => x.CustomerId).ToList();
```

Objednávky jsou nejdříve seřazeny vzestupně dle vlastnosti *Created*, a poté sestupně vlastnosti *CustomerId*. Dotaz je možné zapsat i pomocí metody AddQueryOption ve formě

```
var orders = client.container.Orders
    .AddQueryOption("$orderby", "Created,CustomerId desc").ToList();
```

Oba dotazy vyprodukují požadavek ve formě

```
GET http://localhost:50921/odata/Orders?$orderby=Created,CustomerId desc
```

Operátory *$top* a *$skip* jsou implementovány pomocí LINQ metod *Take* a *Skip*. Příkladem použití může být dotaz

```
var customers = client.container.Customers.Skip(5).Take(5).ToList();
```

Ekvivalentem při využití metody AddQueryOption je dotaz

```
var customers = client.container.Customers
    .AddQueryOption("$skip", 5).AddQueryOption("$top", 5).ToList();
```

Oba dotazy vygenerují požadavek

```
GET http://localhost:50921/odata/Customers?$skip=5&$top=5
```

Operátor *$filter* je implementovaný LINQ metodou *Where*. Použití může mít formu

```
var customers = client.container.Customers
    .Where(x => x.Firstname.Contains("lan") && x.Id < 3).ToList();
```

Ekvivalentem je kód

```
var customers = client.container.Customers
    .AddQueryOption("$filter", "contains(Firstname,'lan') and Id lt 3")
    .ToList();
```

Výsledkem je dotaz

```
GET http://localhost:50921/odata/
   Customers?$filter=contains(Firstname,'lan') and Id lt 3
```

Z kódu je patrné, že jak metoda *String.Contains*, tak operátory *&&* a *<* jsou automaticky přeloženy do jejich OData formy. Seznam všech funkcí a operátorů, jež jsou automaticky překládány z .NET kódu do OData dotazů, je možné nalézt v citaci [54]. Jednotlivé operátory a funkce v dotazech je možné kombinovat [53][54].

7.6 Volání akcí a funkcí

Pro volání akcí a funkcí slouží metoda *Execute* třídy *DataServiceContext*. Metoda existuje ve více formách pro volání akcí i funkcí. Volání akce *AddOrderItem* má tvar

```
Uri actionUri = new Uri(client.container.BaseUri,
    "Orders/Default.AddOrderItem");
client.container.Execute(actionUri, "POST",
    new[] { new BodyOperationParameter("orderId", 2),
            new BodyOperationParameter("orderItemId", 1) });
```

Návratovým typem metody Execute je zde instance typu *OperationResponse*. Argumenty jsou v pořadí *Uri* akce, *metoda požadavku* a *pole argumentů* akce. Tvar metody Execute pro volání funkce *GetTotalCost* má tvar

```
Uri functionUri = new Uri(client.container.BaseUri,
    "Orders/Default.GetTotalCost");
decimal ret = client.container.Execute<decimal>(functionUri, "GET",
    true, new UriOperationParameter("orderId", 1)).First();
```

Zde je návratový typ metody určen jejím generickým argumentem. Argumenty jsou v pořadí *Uri* funkce, *metoda požadavku*, zda je *návratová hodnota rozměru jedna* (*true*), nebo rozměru *kolekce* (*false*) a *pole argumentů* funkce [55].

7.7 Využití třídy DataServiceCollection

Pro pohodlné propojení OData klienta s uživatelským rozhraním v prostředí .NET slouží třída *DataServiceCollection<T>*, kde *T* je typ entity. Tato kolekce dědí od třídy *ObservableCollection<T>*, a je ji tudíž možné použít pro *databinding*. Vytvoření kolekce pro entity typu *Customer* může mít tvar

```
DataServiceCollection<Customer> customers =
    new DataServiceCollection<Customer>
    (
        client.container.Customers.Context,
        client.container.Customers.Expand("Orders").Execute(),
        TrackingMode.AutoChangeTracking,
        "Customers",
        EntityChanged,
        EntityCollectionChanged
    );
```

Konstruktor vyžaduje parametry v pořadí instance typu *DataServiceContext* pro synchronizaci změn, *inicializační položky* kolekce, *mód synchronizace*, název *entityset*, *delegát* pro událost *změna entity* a *delegát* pro událost *změna v kolekci* entit.

Z konstruktoru je patrné, že instance typu DataServiceCollection může sloužit jako prostředek pro synchronizaci kontextu a uživatelského rozhraní. V kódu je jako synchronizační mód zvolen mód *AutoChangeTracking*, který provádí automatickou propagaci změn.

Delegáti *EntityChanged* a *EntityCollecionChanged* mají návratovou hodnotu typu *bool*. Změna nebude propagována do kontextu, pokud delegát vrátí hodnotu *true*. V takovém případě je počítáno s tím, že sám delegát provede potřebné synchronizační kroky.

Využití takovéto kolekce potom může mít podobu

```
DataGridView dgCustomer; // …
BindingSource bsCustomer; // …
dgCustomer.DataSource = bsCustomer;
bsCustomer.DataSource = customers;
```

Změny provedené v tabulce *dgCustomer* jsou poté automaticky synchronizovány v kontextu, do kterého patří instance v kolekci *customers*, tzn. přidání entity, odstranění entity a editace entity skrze sloupce tabulky [56].

8 VYHODNOCENÍ

Z kapitol 6 a 7 vyplívá, že OData služby jsou vhodné především pro čtení, vytváření, úpravu a odstraňování datových entit. OData služby splňují požadavky SOA, díky čemuž je možné využít je jako platformě nezávislé a interoperatibilní rozhraní mezi klientem a datovým zdrojem, veřejné datově orientované API pro třetí strany, apod.

OData služby přenášejí data a zprávy ve formátu JSON. Díky tomu mohou být využívány pomocí JavaScritptového, nebo mobilního klienta.

Pro zabezpečení OData služeb je možné využít klasických metod, které se používají k zabezpečení HTTP spojení. OData služby je tedy možné použít v případech, kdy je potřebná vysoká úroveň zabezpečení přenášených dat, a možnost provádění autentizace a autorizace klientů služby.

Díky možnosti využít v dotazech System Query Options má klient široké možnosti upravování dotazů dle vlastních potřeb, přičemž poskytovatel služby nemusí vynaložit žádné dodatečné úsilí. Tyto možnosti však jsou omezené, jelikož klient nemůže využít ekvivalentů příkazů *JOIN* jazyka *SQL*. Jak bylo uvedeno v kapitole 2.2.4, načtení svázaných entit je možné vyžádat operátorem *$expand*. Příkladem by mohl být dotaz

```
[adr]/odata/Customers(1)?$expand=Orders($expand=OrderItems($expand=StoreItem))
```

který zpřístupní položky typu *StoreItem* objednané entitou *Customer* s Id rovno 1. K získání položek bylo nutné použít trojnásobné vnoření příkazu, což může u velkého množství dat způsobit potíže s rychlostí vykonání dotazu.

Zároveň je při využití knihovny Entity Framework nutné brát v potaz řádově pomalejší provádění operací, než je tomu např. při využití instance typu *SqlCommand*. Příkladem může být test ve tvaru

```
var sw = new System.Diagnostics.Stopwatch();
long sum = 0;
for (int i = 0; i < 100; i++)
{
    sw.Restart();
    IEnumerable<string> customerOrderedStoreItemIds =
        client.container.Customers.ByKey(1)
        .Orders.Expand("OrderItems").ToList()
        .SelectMany(x => x.OrderItems).Select(x => x.StoreItemId).Distinct();
    sum += sw.ElapsedMilliseconds;
}
System.Diagnostics.Debug.WriteLine(sum / 100d);
```

Test sto krát opakuje dotaz, jenž vrátí vlastnost *StoreItemId* entit typu *OrderItem* patřících k entitě typu *Customer* s *Id* rovno *1*. Průměrná doba potřebná pro vykonání dotazu a navrácení výsledku činila cca 10 ms. Ekvivalentní SQL dotaz by měl tvar

```
SELECT DISTINCT oi.StoreItemId
FROM Customer c
INNER JOIN [Order] o ON o.CustomerId=c.Id
INNER JOIN OrderItem oi ON oi.OrderId=o.Id
WHERE c.Id = 1
```

Při využití metody *SqlCommand.ExecuteReader* činila průměrná doba méně než 1 ms. Samozřejmě zde byla vynechána část, kdy OData služba vytváří HTTP odpověď a provádí serializaci dat. Tento test byl prováděn nad datovým zdrojem s počtem entit v řádu jednotek. Se zvětšujícím se počtem dat a složitostí dotazu se může doba provádění dotazů při využití knihovny Entity Framework razantně prodloužit. Více informací o výkonnosti Entity Framework viz [57].

V případě, že klient vyžaduje nadstandardní dotaz, který nelze provést pomocí System Query Options, a nebo je doba vykonávání dotazu příliš dlouhá, je nejvhodnější variantou vytvoření Funkce na straně serveru. V těle funkce je potom možné využít dotazování skrze knihovnu Entity Framework, a nebo využít přímého SQL dotazování pomocí metody SqlCommand.ExecuteReader, a poté provést manuální materializaci výsledku do podoby definované kontraktem OData služby.

ZÁVĚR

Cílem práce bylo představit protokol OData a analyzovat možnosti jeho využití na platformě .NET.

V úvodní části textu byly uvedeny klíčové principy SOA a souvislost OData se SOA. Poté byla popsána struktura protokolu OData, následovaná přehledem možností poskytování a konzumování OData služeb na jednotlivých platformách. Rovněž byly uvedeny možnosti zabezpečení OData služeb pomocí autentizace a šifrování přenášených dat.

V praktické části byly uvedeny případové studie pro implementaci OData služby pomocí technologie ASP.NET Web Api OData, a případové studie pro konzumaci implementované služby pomocí .NET knihovny OData v4 Client Code Generator.

Práci je možné využít pro seznámení se s protokolem OData a s vhodnými oblastmi jeho využití. Praktická část práce může být využita jako pomocný text pro implementaci základních funkcionalit OData služby a jejího klienta na platformě .NET.

SEZNAM POUŽITÉ LITERATURY

[1] ERL, Thomas. *SOA: servisně orientovaná architektura: kompletní průvodce*. Vyd. 1. Překlad Ondřej Baše, Lukáš Krejčí. Brno: Computer Press, 2009, 671 s. ISBN 978-80-251-1886-3.

[2] Understanding WSDL. *MSDN - Microsoft Developer Network* [online]. 2003 [cit. 2015-03-18]. Dostupné z: https://msdn.microsoft.com/en-us/library/ms996486.aspx

[3] Understanding SOAP. *MSDN - Microsoft Developer Network* [online]. 2003 [cit. 2015-03-18]. Dostupné z: https://msdn.microsoft.com/en-us/library/ms995800.aspx

[4] Representational State Transfer (REST): CHAPTER 5. *Donald Bren School of Information and Computer Sciences*[online]. 2000 [cit. 2015-03-18]. Dostupné z: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[5] A Guide to Designing and Building RESTful Web Services with WCF 3.5. *MSDN - Microsoft Developer Network* [online]. 2008 [cit. 2015-03-18]. Dostupné z: https://msdn.microsoft.com/en-us/library/dd203052.aspx

[6] SOAP, REST, and More. *MSDN Magazine* [online]. 2009 [cit. 2015-03-18]. Dostupné z: https://msdn.microsoft.com/en-us/magazine/dd942839.aspx

[7] Overview. *OData Version 4.0: Part 1: Protocol Plus Errata 02* [online]. 2014-10-30. 2014 [cit. 2015-02-02]. Dostupné z:http://docs.oasis-open.org/odata/odata/v4.0/errata02/os/complete/part1-protocol/odata-v4.0-errata02-os-part1-protocol-complete.html#_Toc406398200

[8] Data Model. *OData Version 4.0: Part 1: Protocol Plus Errata 02* [online]. 2014-10-30. 2014 [cit. 2015-02-02]. Dostupné z:http://docs.oasis-open.org/odata/odata/v4.0/errata02/os/complete/part1-protocol/odata-v4.0-errata02-os-part1-protocol-complete.html#_Toc406398201

[9] CSDL Namespaces. *OData Version 4.0: Part 3: Common Schema Definition Language (CSDL) Plus Errata 02* [online]. 2014-10-30. 2014 [cit. 2015-02-02]. Dostupné z: http://docs.oasis-open.org/odata/odata/v4.0/errata02/os/complete/part3-csdl/odata-v4.0-errata02-os-part3-csdl-complete.html#_Toc406397921

[10] Entity Model Wrapper. *OData Version 4.0: Part 3: Common Schema Definition Language (CSDL) Plus Errata 02* [online]. 2014-10-30. 2014 [cit. 2015-02-02]. Dostupné z: http://docs.oasis-open.org/odata/odata/v4.0/errata02/os/complete/part3-csdl/odata-v4.0-errata02-os-part3-csdl-complete.html#_Toc406397926

[11] Common Characteristics of Entity Models. *OData Version 4.0: Part 3: Common Schema Definition Language (CSDL) Plus Errata 02* [online]. 2014-10-30. 2014 [cit. 2015-02-02].

Dostupné z: http://docs.oasis-open.org/odata/odata/v4.0/errata02/os/complete/part3-csdl/odata-v4.0-errata02-os-part3-csdl-complete.html#_Toc406397939

[12] Structural Property. *OData Version 4.0: Part 3: Common Schema Definition Language (CSDL) Plus Errata 02* [online]. 2014-10-30. 2014 [cit. 2015-02-02]. Dostupné z: http://docs.oasis-open.org/odata/odata/v4.0/errata02/os/complete/part3-csdl/odata-v4.0-errata02-os-part3-csdl-complete.html#_Toc406397950

[13] Navigation Property. *OData Version 4.0: Part 3: Common Schema Definition Language (CSDL) Plus Errata 02* [online]. 2014-10-30. 2014 [cit. 2015-02-02]. Dostupné z: http://docs.oasis-open.org/odata/odata/v4.0/errata02/os/complete/part3-csdl/odata-v4.0-errata02-os-part3-csdl-complete.html#_Toc406397962

[14] Entity Type. *OData Version 4.0: Part 3: Common Schema Definition Language (CSDL) Plus Errata 02* [online]. 2014-10-30. 2014 [cit. 2015-02-02]. Dostupné z: http://docs.oasis-open.org/odata/odata/v4.0/errata02/os/complete/part3-csdl/odata-v4.0-errata02-os-part3-csdl-complete.html#_Toc406397974

[15] Complex Type. *OData Version 4.0: Part 3: Common Schema Definition Language (CSDL) Plus Errata 02* [online]. 2014-10-30. 2014 [cit. 2015-02-02]. Dostupné z: http://docs.oasis-open.org/odata/odata/v4.0/errata02/os/complete/part3-csdl/odata-v4.0-errata02-os-part3-csdl-complete.html#_Toc406397985

[16] Enumeration Type. *OData Version 4.0: Part 3: Common Schema Definition Language (CSDL) Plus Errata 02* [online]. 2014-10-30. 2014 [cit. 2015-02-02]. Dostupné z: http://docs.oasis-open.org/odata/odata/v4.0/errata02/os/complete/part3-csdl/odata-v4.0-errata02-os-part3-csdl-complete.html#_Toc406397991

[17] Action and Function. *OData Version 4.0: Part 3: Common Schema Definition Language (CSDL) Plus Errata 02* [online]. 2014-10-30. 2014 [cit. 2015-02-02]. Dostupné z: http://docs.oasis-open.org/odata/odata/v4.0/errata02/os/complete/part3-csdl/odata-v4.0-errata02-os-part3-csdl-complete.html#_Toc406398004

[18] Entity Container. *OData Version 4.0: Part 3: Common Schema Definition Language (CSDL) Plus Errata 02* [online]. 2014-10-30. 2014 [cit. 2015-02-02]. Dostupné z: http://docs.oasis-open.org/odata/odata/v4.0/errata02/os/complete/part3-csdl/odata-v4.0-errata02-os-part3-csdl-complete.html#_Toc406398024

[19] Common Response Status Codes. *OData Version 4.0: Part 1: Protocol Plus Errata 02* [online]. 2014-10-30. 2014 [cit. 2015-02-02]. Dostupné z: http://docs.oasis-open.org/odata/odata/v4.0/errata02/os/complete/part1-protocol/odata-v4.0-errata02-os-part1-protocol-complete.html#_Toc406398249

[20] Data Service Requests. *OData Version 4.0: Part 1: Protocol Plus Errata 02* [online]. 2014-10-30. 2014 [cit. 2015-02-02]. Dostupné z: http://docs.oasis-

open.org/odata/odata/v4.0/errata02/os/complete/part1-protocol/odata-v4.0-errata02-os-part1-protocol-complete.html#_Toc406398286

[21] Libraries. *OData: The Protocol for REST APIs* [online]. 2015 [cit. 2015-02-07]. Dostupné z: http://www.odata.org/libraries/

[22] Documentation OData 4.0 Java Library. *Apache Olingo Library* [online]. 2014 [cit. 2015-02-10]. Dostupné z:http://olingo.apache.org/

[23] MSOpenTech/odataphpprod. *GitHub* [online]. 2012 [cit. 2015-02-11]. Dostupné z: https://github.com/MSOpenTech/odataphpprod

[24] WCF Data Services. *Visual Studio | MSDN* [online]. 2015 [cit. 2015-02-11]. Dostupné z: https://msdn.microsoft.com/en-us/library/vstudio/cc668792(v=vs.103).aspx

[25] OData in ASP.NET Web API. *ASP.NET: The ASP.NET Site* [online]. 2015 [cit. 2015-02-11]. Dostupné z:http://www.asp.net/web-api/overview/odata-support-in-aspnet-web-api

[26] [Discussion] Future Direction of WCF Data Services. *MSDN Blogs* [online]. 2014 [cit. 2015-02-11]. Dostupné z:http://blogs.msdn.com/b/odatateam/archive/2014/03/27/future-direction-of-wcf-data-services.aspx

[27] DateTime Support in Web API OData V4. *GitHub* [online]. 2015 [cit. 2015-03-10]. Dostupné z: http://odata.github.io/WebApi/datetime-support/

[28] Self-Host ASP.NET Web API 1. *ASP.NET: The ASP.NET Site* [online]. 2012 [cit. 2015-02-16]. Dostupné z:http://www.asp.net/web-api/overview/older-versions/self-host-a-web-api

[29] GLENN BLOCK, Pablo Cibraro. *Designing Evolvable Web APIs With ASP.NET*. 1., neue Ausg. S.l.: O'Reilly Media, 2013. ISBN 978-144-9337-711.

[30] MUELLER, John Paul. *Microsoft ADO.NET entity framework step by step*. Sebastopol (Calif.): O'Reilly Media, 2013. ISBN 07-356-6416-1

[31] Datajs: JavaScript Library for data-centric web applications. *CodePlex* [online]. 2014 [cit. 2015-02-14]. Dostupné z:http://datajs.codeplex.com/

[32] ODataCpp. *GitHub* [online]. 2015 [cit. 2015-02-14]. Dostupné z: https://github.com/OData/odatacpp-client

[33] ODataPy. *GitHub* [online]. 2014 [cit. 2015-02-14]. Dostupné z: https://github.com/odata/odatapy-client

[34] OData v4 Client Code Generator. *Visual Studio* [online]. 2014 [cit. 2015-02-14]. Dostupné z: https://visualstudiogallery.msdn.microsoft.com/9b786c0e-79d1-4a50-89a5-125e57475937

[35] WCF Data Services Client for OData v1-3. *NuGet Gallery* [online]. 2014 [cit. 2015-02-14]. Dostupné z:http://www.nuget.org/packages/Microsoft.Data.Services.Client/

[36] Security Considerations. *OData Version 4.0: Part 1: Protocol Plus Errata 02* [online]. 2014-10-30. 2014 [cit. 2015-02-07]. Dostupné z: http://docs.oasis-open.org/odata/odata/v4.0/errata02/os/complete/part1-protocol/odata-v4.0-errata02-os-part1-protocol-complete.html#_Toc406398366

[37] HTTP Authentication: Basic and Digest Access Authentication. *IETF Documents* [online]. 1999 [cit. 2015-02-07]. Dostupné z: http://tools.ietf.org/html/rfc2617

[38] The OAuth 2.0 Authorization Framework. *IETF Documents* [online]. 2012 [cit. 2015-02-07]. Dostupné z: https://tools.ietf.org/html/rfc6749

[39] The Secure Sockets Layer (SSL) Protocol Version 3.0. *IETF Documents* [online]. 2011 [cit. 2015-02-07]. Dostupné z: https://tools.ietf.org/html/rfc6101

[40] The Transport Layer Security (TLS) Protocol: Version 1.2. *IETF Documents* [online]. 2008 [cit. 2015-02-07]. Dostupné z:http://tools.ietf.org/html/rfc5246

[41] Create an OData v4 Endpoint Using ASP.NET Web API 2.2. *ASP.NET* [online]. 2014 [cit. 2015-03-08]. Dostupné z: http://www.asp.net/web-api/overview/odata-support-in-aspnet-web-api/odata-v4/create-an-odata-v4-endpoint

[42] Routing Conventions in ASP.NET Web API 2 Odata. *ASP.NET* [online]. 2013 [cit. 2015-03-08]. Dostupné z: http://www.asp.net/web-api/overview/odata-support-in-aspnet-web-api/odata-routing-conventions

[43] Entity Relations in OData v4 Using ASP.NET Web Api 2.2. *ASP.NET: The ASP.NET Site* [online]. 2014 [cit. 2015-03-25]. Dostupné z: http://www.asp.net/web-api/overview/odata-support-in-aspnet-web-api/odata-v4/entity-relations-in-odata-v4

[44] Using $select, $expand, and $value in ASP.NET Web API 2 ODataect,-$expand,-and-$value. *ASP.NET: The ASP.NET Site* [online]. 2013 [cit. 2015-03-31]. Dostupné z: http://www.asp.net/web-api/overview/odata-support-in-aspnet-web-api/using-$select,-$expand,-and-$value

[45] Supporting OData Query Options in ASP.NET Web API 2. *ASP.NET: The ASP.NET Site* [online]. 2013 [cit. 2015-03-31]. Dostupné z: http://www.asp.net/web-api/overview/odata-support-in-aspnet-web-api/supporting-odata-query-options

[46] Actions and Functions in OData v4 Using ASP.NET Web API 2.2. *ASP.NET: The ASP.NET Site* [online]. 2014 [cit. 2015-03-31]. Dostupné z: http://www.asp.net/web-api/overview/odata-support-in-aspnet-web-api/odata-v4/odata-actions-and-functions

[47] OData Scaffolding. *.NET Web Development And Tools Blog* [online]. 2013 [cit. 2015-04-03]. Dostupné z:http://blogs.msdn.com/b/webdev/archive/2013/11/08/odata-scaffolding.aspx

[48] Authentication and Authorization in ASP.NET Web API. *ASP.NET: The ASP.NET Site* [online]. 2012 [cit. 2015-04-01]. Dostupné z: http://www.asp.net/web-api/overview/security/authentication-and-authorization-in-aspnet-web-api

[49] Open Data Protocol. *CodePlex* [online]. 2012 [cit. 2015-04-02]. Dostupné z: https://odata.codeplex.com/

[50] Create an OData v4 Client App. *ASP.NET: The ASP.NET Site* [online]. 2014 [cit. 2015-04-02]. Dostupné z:http://www.asp.net/web-api/overview/odata-support-in-aspnet-web-api/odata-v4/create-an-odata-v4-client-app

[51] Managing the Data Service Context. *MSDN - Microsoft Developer Network* [online]. 2015 [cit. 2015-04-02]. Dostupné z: https://msdn.microsoft.com/en-us/library/gg602811(v=vs.110).aspx

[52] Updating the Data Service. *MSDN - Microsoft Developer Network* [online]. 2015 [cit. 2015-04-02]. Dostupné z: https://msdn.microsoft.com/en-us/library/dd756361(v=vs.103).aspx

[53] Querying the Data Service. *MSDN - Microsoft Developer Network* [online]. 2015 [cit. 2015-04-04]. Dostupné z: https://msdn.microsoft.com/en-us/library/dd673933(v=vs.103).aspx

[54] LINQ Considerations. *MSDN - Microsoft Developer Network* [online]. 2015 [cit. 2015-04-04]. Dostupné z: https://msdn.microsoft.com/en-us/library/ee622463(v=vs.103).aspx

[55] Calling Service Operations and Actions. *MSDN - Microsoft Developer Network* [online]. 2015 [cit. 2015-04-05]. Dostupné z: https://msdn.microsoft.com/en-us/library/hh230677(v=vs.103).aspx

[56] Binding Data to Controls. *MSDN - Microsoft Developer Network* [online]. 2015 [cit. 2015-04-05]. Dostupné z: https://msdn.microsoft.com/en-us/library/ee373844(v=vs.103).aspx

[57] Performance Considerations (Entity Framework). *Visual Studio | MSDN* [online]. 2015 [cit. 2015-04-14]. Dostupné z: https://msdn.microsoft.com/en-us/library/vstudio/cc853327(v=vs.100).aspx

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

ADO.NET	Microsoft ActiveX Data Object.NET. Soubor nástrojů sloužících k manipulaci s datovými zdroji a jejich daty.
ASP	Active Server Pages. Framework sloužící k vytváření webových aplikací.
CRUD	Create, Read, Update, Delete. Sada operací vytvoř, čti, edituj, odstraň.
CSDL	Common Schema Definition Language. Dokument popisující kontrakt OData služby.
CSDL	Conceptual Schema Definition Language. Dokument popisující entity, vazby entit, a funkce v konceptuální vrstvě Entity Frameworku.
EDM	Entity Data Model. Abstraktní model definující datové entity, jejich vazby a vlastnosti.
ER Diagram	Entity Relationship Diagram. Diagram charakterizující datové entity, jejich vazby a vlastnosti.
HTTP	HyperText Transfer Protocol. Protokol aplikační vrstvy určený pro výměnu hypertextových dokumentů.
HTTPS	HTTP Secure. Nadstavba HTTP zajišťující bezpečnost přenosu dat.
IIS	Internet Information Service. Webový server firmy Microsoft.
JSON	JavaScript Object Notation. Formát pro přenos dat využívající strukturu klíč-hodnota.
MAC	Message Authenticaton Code. Hěšovací funkce určená pro autentizace integrity dat.
MD5	Message Digest Algorithm 5. Kryptografická hešovací funkce produkující 128 bitů dlouhý řetězec.
MITM	Man In The Middle. Sowtfarový útok, kde útočník účinkuje jako prostředník komunikace mezi komunikujícími stranami. Ani jedna ze stran neregistruje, že komunikuje skrze útočníka.
MVC	Model View Controller. Architektonický vzor rozdělující aplikaci na modely, pohledy, a kontroléry.

OData	Open Data Protocol. Protokol určený pro poskytování REST webových služeb.
OWIN	Open Web Interface for .NET. Standard definující rozhraní mezi webovým serverem a .NET webovou aplikací.
PHP	PHP: Hypertext Preprocessor. Skriptovací jazyk určený pro tvorbu webových aplikací.
PRF	Pseudo Random Function. Alrogitmus generující hash sekvenci s pomocí pseudonáhodných funkcí a násadových argumentů.
REST	Representational State Transfer. Architektonický styl pro tvorbu aplikací typu klient-server.
RFC	Request for Comments. Sada doporučení týkající se internetového prostředí.
RPC	Remote Procedure Call. Typ komunikace kdy klient využívá proceduru z jiného adresního prostroru.
RSA Algorithm	Rivest, Shamir, Adleman Algorithm. Algoritmus pracující na bázi asymetrického šifrování využívaný v protokolu TLS.
SMTP	Simple Mail Transfer Protocol. Protokol pro přenos emailové komunikace.
SOA	Servisně orientovaná architektura. Architektura definující požadavky na komunikaci mezi logickými jednotkami softwaru.
SOAP	Simple Object Access Protocol. Definice interoperatibilního komunikačního rozhraní pro přenášení zpráv.
SQL	Structured Query Language. Programovací jazyk specificky určený pro komunikaci s relační databází.
SSL	Secure Socket Layer. Šifrovací protokol využívaný protokolem HTTPS. V současné době již není považován za bezpečný.
TCP	Transmission Control Protocol. Protokol 4. vrstvy referenčního síťového modelu. Poskytuje spolehlivé stavové připojení.
TLS	Transport Layer Security. Šifrovací protokol využívaný protokolem

	HTTPS.
URL	Uniform Resource Locator. Jednoznačný identifikátor určující adresu zdroje.
WCF	Windows Communication Foundation. Framework určený pro tvorbu servisně orientovaných aplikací.
WSDL	Web Services Description Language. Jazyk popisující konktrakt webové služby.
XML	EXtensible Markup Language. Značkovací jazyk určený pro vytváření dokumentů. Využívá elementů a atributů pro interpretaci uložených informací.

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

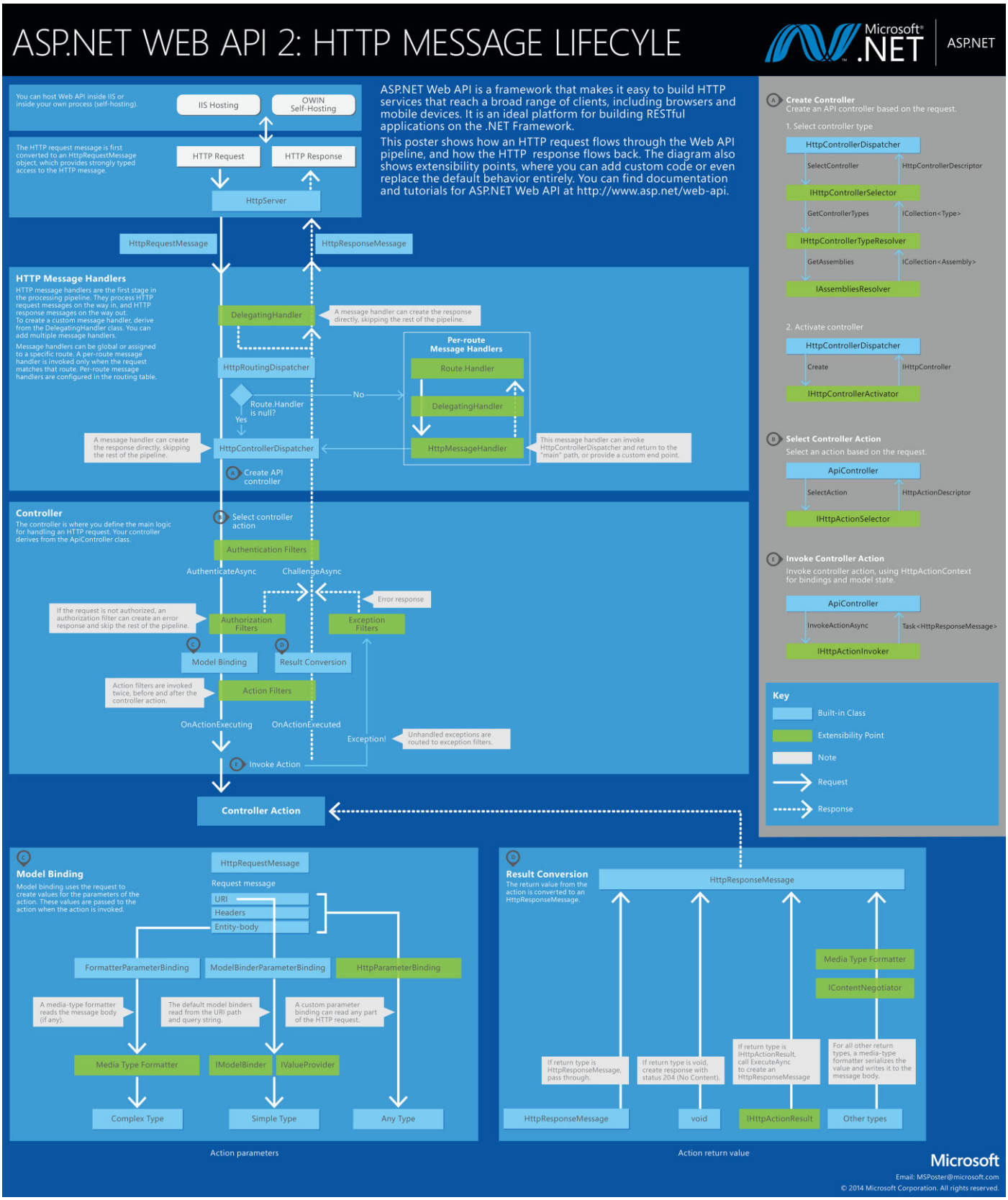
PŘÍLOHA P I: CSDL DOKUMENT ODATA SLUŽBY

```
<?xml version="1.0" encoding="utf-8"?>
 <edmx:Edmx Version="4.0"
     xmlns:edmx="http://docs.oasis-open.org/odata/ns/edmx">
   <edmx:DataServices>
     <Schema Namespace="Server.Edm"
         xmlns="http://docs.oasis-open.org/odata/ns/edm">
       <EntityType Name="Category">
         <Key>
           <PropertyRef Name="Id" />
         </Key>
         <Property Name="Id" Type="Edm.Int32" Nullable="false" />
         <Property Name="ParentCategoryId" Type="Edm.Int32" />
         <Property Name="Name" Type="Edm.String" Nullable="false" />
         <NavigationProperty Name="ChildCategories"
           Type="Collection(Server.Edm.Category)" />
         <NavigationProperty Name="ParentCategory"
           Type="Server.Edm.Category" />
         <NavigationProperty Name="StoreItems"
           Type="Collection(Server.Edm.StoreItem)" />
       </EntityType>
       <EntityType Name="Contact">
         <Key>
           <PropertyRef Name="Id" />
         </Key>
         <Property Name="Id" Type="Edm.Int32" Nullable="false" />
         <Property Name="CustomerId" Type="Edm.Int32" Nullable="false" />
         <Property Name="ContactTypeId" Type="Edm.Int32" Nullable="false" />
         <Property Name="Value" Type="Edm.String" Nullable="false" />
         <NavigationProperty Name="ContactType"
           Type="Server.Edm.ContactType" />
         <NavigationProperty Name="Customer" Type="Server.Edm.Customer" />
       </EntityType>
       <EntityType Name="ContactType">
         <Key>
           <PropertyRef Name="Id" />
         </Key>
         <Property Name="Id" Type="Edm.Int32" Nullable="false" />
         <Property Name="Name" Type="Edm.String" Nullable="false" />
         <NavigationProperty Name="Contacts"
           Type="Collection(Server.Edm.Contact)" />
       </EntityType>
       <EntityType Name="Customer">
         <Key>
           <PropertyRef Name="Id" />
         </Key>
         <Property Name="Id" Type="Edm.Int32" Nullable="false" />
         <Property Name="Firstname" Type="Edm.String" Nullable="false" />
         <Property Name="Lastname" Type="Edm.String" Nullable="false" />
         <Property Name="Note" Type="Edm.String" />
         <NavigationProperty Name="Contacts"
           Type="Collection(Server.Edm.Contact)" />
         <NavigationProperty Name="Orders"
           Type="Collection(Server.Edm.Order)" />
       </EntityType>
       <EntityType Name="CustomersOrdersCount">
         <Key>
```

```
        <PropertyRef Name="CustomerId" />
      </Key>
      <Property Name="CustomerId" Type="Edm.Int32" Nullable="false" />
      <Property Name="OrderCount" Type="Edm.Int32" />
    </EntityType>
    <EntityType Name="Order">
      <Key>
        <PropertyRef Name="Id" />
      </Key>
      <Property Name="Created" Type="Edm.DateTimeOffset"
        Nullable="false" />
      <Property Name="Id" Type="Edm.Int32" Nullable="false" />
      <Property Name="CustomerId" Type="Edm.Int32" Nullable="false" />
      <NavigationProperty Name="Customer" Type="Server.Edm.Customer" />
      <NavigationProperty Name="OrderItems"
        Type="Collection(Server.Edm.OrderItem)" />
    </EntityType>
    <ComplexType Name="AddOrderItemModel">
      <Property Name="orderId" Type="Edm.Int32" Nullable="false" />
      <Property Name="orderItemId" Type="Edm.Int32" Nullable="false" />
    </ComplexType>
    <EntityType Name="OrderItem">
      <Key>
        <PropertyRef Name="OrderId" />
        <PropertyRef Name="StoreItemId" />
      </Key>
      <Property Name="OrderId" Type="Edm.Int32" Nullable="false" />
      <Property Name="StoreItemId" Type="Edm.String" Nullable="false" />
      <Property Name="Count" Type="Edm.Int32" Nullable="false" />
      <NavigationProperty Name="Order" Type="Server.Edm.Order" />
      <NavigationProperty Name="StoreItem" Type="Server.Edm.StoreItem" />
    </EntityType>
    <EntityType Name="StoreItem">
      <Key>
        <PropertyRef Name="Id" />
      </Key>
      <Property Name="Id" Type="Edm.String" Nullable="false" />
      <Property Name="CategoryId" Type="Edm.Int32" Nullable="false" />
      <Property Name="Name" Type="Edm.String" Nullable="false" />
      <Property Name="Price" Type="Edm.Decimal" Nullable="false" />
      <NavigationProperty Name="Category" Type="Server.Edm.Category" />
      <NavigationProperty Name="OrderItems"
        Type="Collection(Server.Edm.OrderItem)" />
    </EntityType>
  </Schema>
  <Schema Namespace="Default"
      xmlns="http://docs.oasis-open.org/odata/ns/edm">
    <Function Name="GetTotalCost" IsBound="true">
      <Parameter Name="bindingParameter"
        Type="Collection(Server.Edm.Order)" />
      <Parameter Name="orderId" Type="Edm.Int32" Nullable="false" />
      <ReturnType Type="Edm.Double" Nullable="false" />
    </Function>
    <Action Name="AddOrderItem" IsBound="true">
      <Parameter Name="bindingParameter"
        Type="Collection(Server.Edm.Order)" />
      <Parameter Name="addOrderItemModel"
        Type="Server.Edm.AddOrderItemModel" />
    </Action>
```

```
      <EntityContainer Name="Container">
        <EntitySet Name="Categories" EntityType="Server.Edm.Category">
          <NavigationPropertyBinding Path="ChildCategories"
            Target="Categories" />
          <NavigationPropertyBinding Path="ParentCategory"
            Target="Categories" />
          <NavigationPropertyBinding Path="StoreItems"
            Target="StoreItems" />
        </EntitySet>
        <EntitySet Name="Contacts" EntityType="Server.Edm.Contact">
          <NavigationPropertyBinding Path="ContactType"
            Target="ContactTypes" />
          <NavigationPropertyBinding Path="Customer" Target="Customers" />
        </EntitySet>
        <EntitySet Name="ContactTypes" EntityType="Server.Edm.ContactType">
          <NavigationPropertyBinding Path="Contacts" Target="Contacts" />
        </EntitySet>
        <EntitySet Name="Customers" EntityType="Server.Edm.Customer">
          <NavigationPropertyBinding Path="Contacts" Target="Contacts" />
          <NavigationPropertyBinding Path="Orders" Target="Orders" />
        </EntitySet>
        <EntitySet Name="CustomerOrdersCounts"
          EntityType="Server.Edm.CustomersOrdersCount" />
        <EntitySet Name="Orders" EntityType="Server.Edm.Order">
          <NavigationPropertyBinding Path="Customer" Target="Customers" />
          <NavigationPropertyBinding Path="OrderItems"
            Target="OrderItems" />
        </EntitySet>
        <EntitySet Name="OrderItems" EntityType="Server.Edm.OrderItem">
          <NavigationPropertyBinding Path="Order" Target="Orders" />
          <NavigationPropertyBinding Path="StoreItem" Target="StoreItems" />
        </EntitySet>
        <EntitySet Name="StoreItems" EntityType="Server.Edm.StoreItem">
          <NavigationPropertyBinding Path="Category" Target="Categories" />
          <NavigationPropertyBinding Path="OrderItems"
            Target="OrderItems" />
        </EntitySet>
      </EntityContainer>
    </Schema>
  </edmx:DataServices>
</edmx:Edmx>
```

PŘÍLOHA P II: ŽIVOTNÍ CYKLUS ZPRÁVY V ASP.NET WEB API



Dostupné na adrese http://www.asp.net/media/4071077/aspnet-web-api-poster.pdf

PŘÍLOHA P III: METODA GETKEYFROMURI

```
public static TKey GetKeyFromUri<TKey>(HttpRequestMessage request, Uri uri)
{
    if (uri == null) throw new ArgumentNullException("uri");
    var urlHelper = request.GetUrlHelper() ?? new UrlHelper(request);
    string serviceRoot = urlHelper.CreateODataLink
    (
        request.ODataProperties().RouteName,
        request.ODataProperties().PathHandler,
        new List<ODataPathSegment>()
    );
    var odataPath = request.ODataProperties().PathHandler.Parse
    (
        request.ODataProperties().Model,
        serviceRoot,
        uri.LocalPath
    );
    var keySegment =
        odataPath.Segments.OfType<KeyValuePathSegment>().FirstOrDefault();
    if (keySegment == null) throw new InvalidOperationException
        ("The link does not contain a key.");
    var value = ODataUriUtils.ConvertFromUriLiteral(keySegment.Value,
       ODataVersion.V4);
    return (TKey)value;
}
```