Sem vložte zadání Vaší práce.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Technologie pro tvorbu RESTful API

Bc. Radim Janda

Katedra Softwarového Inženýrství
Vedoucí práce: Ing. Jiří Hunka

3. května 2018

Poděkování

Chtěl bych poděkovat Ing. Jiřímu Hunkovi za vedení mé diplomové práce, cenné rady a odborný dohled. Děkuji také Ing. Jiřímu Chludilovi za oponenturu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 3. května 2018 .....

Odkaz na tuto práci

Janda, Radim. *Technologie pro tvorbu RESTful API*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Tato diplomová práce se zabývá rozborem technologií pro tvorbu RESTful API aplikací.

V práci jsem se zaměřil jak na rozbor samotné architektury REST, tak na důkladnou rešerši vybraných frameworků pro tvorbu RESTful API aplikací, včetně dalších podpůrných nástrojů, které mohou tento vývojový proces ulehčit. Z mnou prozkoumaných technologií je vytvořena sestava technologií a nástrojů pro maximální podporu tvorby RESTful API aplikací. Tato sestava je následně otestována v praktické části při vývoji takové aplikace. Na závěr diskutuji možnosti sjednocení těchto technologií do univerzálního nástroje.

V rešeršní části často pracuji i s praktickými prvky, včetně ukázek kódu. V praktické části byly použity klasické postupy podle běžného životního cyklu vývoje software - tedy analýzy, návrhu, implementace a nasazení.

Hlavním přínosem této práce je porovnání dostupných technologií a rozbor toho, jaké technologie vybrat, aby bylo dosaženo nejvyšší efektivnosti procesu vývoje RESTful API aplikací. Dalším přínosem je také vytvoření konkretního RESTful API, které aktuálně funguje na dobročinném projektu. V poslední řadě práce přináší teoretické základy pro možné sjednocení těchto technologií, které je možné v budoucnu rozvíjet.

Klíčová slova REST, RESTful API, REST frameworky, nástroje, technologie

Abstract

This diploma thesis consists of analysis of technologies used to create RESTful API aplications.

Thesis is focused on analysis of REST architecture itself and thorough research of selected RESTful API frameworks, including other tools which could be used to improve proces of developing these aplications. From research, there is also created stack of these tools and technologies, which should give maximum support for creating RESTful API aplications. This stack is also tested in practical part, which focus on development of this kind of application. At the end there are discussed options of merging these technologies into universal tool.

The research part is highly practically oriented and also includes simple code samples. Practical part is driven by classic software developement life cycle including analysis, design, implementation and deployment.

The main contribution of this thesis is overview of available technologies and analysis of which selection of these technologies could achieve highest effiency process of developing RESTful API aplication. Another contribution is developed RESTful API, which is currently running for charitable project. Lastly this thesis gives theoretical foundations for possible merging of these technologies, which could be envolved in future.

Keywords REST, RESTful API, REST frameworks, tools, technologies

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V dnešní době je REST již poměrně známou a často používanou architekturou. K dispozici je však stále více technologií a nástrojů zabývajících se buď samotným vývojem RESTful API aplikací, nebo alespoň částečnou podporou tohoto vývoje. Pro vývojáře pak nemusí být snadné se v tak velkém množství dostupných technologií orientovat.

Řešením může být tato práce, která shrnuje populární dostupné technologie a rovněž se zaměřuje na to, jaké technologie a nástroje vybírat, aby byla dosažena maximální podpora při tvorbě RESTful API aplikací.

Tímto tématem jsem se rozhodl zabývat, protože i mně samotnému chyběl přehled mezi těmito technologiemi. Tvorba RESTful API je pak pro mě zajímavá oblast, protože jako Android programátor z klientského pohledu toto rozhraní často využívám, avšak postrádal jsem znalosti o tom, jak přesně aplikace na serverech fungují.

Práce je rozdělena do několika částí, kdy je nejprve probrána samotná architektura REST (1. kapitola) a následují podrobnější rešerše dostupných technologií (2. - 3. kapitola). Vlastní tvorba pak začíná stanovením sestavy technologií a nástrojů pro maximální podporu tvorby RESTful API (4. kapitola). Práce pokračuje praktickou částí vývoje RESTful API aplikace za použití této sestavy a zhodnocení její skutečné efektivity (5. kapitola). Vše je zakončeno diskuzí ohledně sjednocení použitých technologií a nástrojů, například do univerzálního jednotného nástroje (6. kapitola).

Kapitola 1

Architektura REST - principy a požadavky

Dříve než se začneme podrobněji zabývat technologiemi spojenými s architekturou REST, bude nezbytné stanovit si, co vlastně tato architektura představuje a jaké jsou její základní principy či požadavky. Tato kapitola tedy poslouží jako úvod do této tématiky a bude se na ni navazovat ve všech následujících kapitolách. Bude zde objasněno, k čemu vlastně architektura slouží, jak se zhruba používá, jaké pravidla je třeba dodržovat, jakým chybám se při návrhu vyhnout a také se podíváme na možnosti zabezpečení.

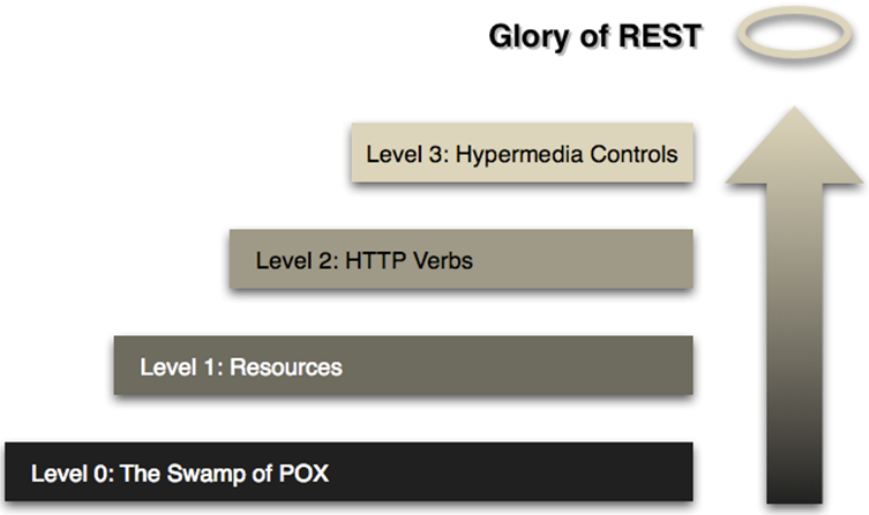
1.1 REST obecně

[1] **REST** (*Representational State Transfer*) poprvé představil Roy Fielding ve své dizertační práci v roce 2000. Jedná se o architektonický styl, navržený pro tvorbu rozhraní v distribuovaném prostředí, řídící se několika přísnými zásadami. Použití této architektury je v praxi nejčastější v situacích, kdy k datům aplikace A potřebujeme přistupovat z různých prostředí (např mobilní aplikace B, webová aplikace C). V takovém případě definujeme nějaké veřejné rozhraní (API), které aplikace A bude poskytovat a aplikace B, C s ním budou schopny komunikovat. REST toto rozhraní definuje za nás.

Na rozdíl od webových služeb řešících obdobné situace jako jsou **SOAP**, nebo **XML-RPC** je **REST orientován datově**. Webové služby definují vzdálené procedury a protokol pro jejich volání, REST určuje, jak se přistupuje k datům. Zaměřuje se na jednotný a snadný přístup ke **zdrojům (resources)**. Zdroj může být prakticky cokoliv, koncept nemá žádná omezení. Může to být například nějaký konkrétní objekt v databázi, dokument, výsledek nějakého výpočtu nebo webová stránka.

1.2 Základní principy

[49] V této podkapitole se podíváme na základní principy RESTu. V literatuře se pro lepší pochopení často rozdělují do 4 úrovní, viz následující obrázek:



Obrázek 1.1: Graf znázorňuje úrovně architektury REST. Zdroj: `https://www.itnetwork.cz/images/92/rest/levels_of_rest.png`

1.2.1 Úroveň 0: The Swamp of POX

Pohled na samotný přenos dat mezi klienty a serverem. Typicky je používán **protokol HTTP** pro jeho rozšířenost a jednoduchost. REST však nevyžaduje použití HTTP, takže teoreticky je možné použít i jinou metodu přenosu.

1.2.2 Úroveň 1: Zdroje - Resources

Pohled na zdroje a jejich jednoznačnou identifikaci. Jak již bylo zmíněno, zdrojem může být prakticky cokoliv, ale pravidlem je, že každý zdroj musí mít své **jednoznačné umístění (URL)**. V případě HTTP tedy nebudou požadavky posílány na jeden centrální bod, ale podle toho, k jakým zdrojům chceme přistupovat.

Nejčastěji budeme chtít, aby naše RESTful API pracovalo se strukturovanými daty. URL se pak typicky skládá z nějakého typu zdroje (pod tím si můžeme představit například název tabulky): ***/<zdroj>***

U daného zdroje můžeme přistupovat i k jeho konkrétním instancím podle jejich ID (v případě tabulky bychom vybírali její konkrétní řádek podle ID): ***/<zdroj>/<id>***

Mimo toho můžeme mezi zdroji tvořit i různé relace: ***/<zdroj>/<id>/<relace>/<id relace>/<další relace>...***

Výše zmíněné si můžeme představit na příkladě s jednoduchým blogem, kde data jsou články (articles), ke kterým můžeme přidávat komentáře (comments). Tato data budou ukládána v nějaké strukturované databázi a REST API bude umožňovat přístup k nim:

/articles - přistupujeme ke všem článkům

/articles/1 - přistupujeme ke článku s ID 1

/articles/1/comments - přistupujeme ke všem komentářům u článku s ID 1

/articles/1/comments/5 - přistupujeme ke komentáři s ID 5 u článku s ID 1

1.2.3 Úroveň 2: HTTP Verbs - metody rozhraní

Pohled na dostupné metody. REST pro práci se zdroji implementuje čtyři základní metody, které jsou známé pod označením **CRUD**- tedy vytvoření dat (**Create**), získání požadovaných dat (**Retrieve**), změnu (**Update**) a smazání (**Delete**). V případě použití HTTP jsou tyto metody implementovány pomocí odpovídajících metod tohoto protokolu.

Významy jednotlivých HTTP Verbs jsou následující:

1.2.3.1 Metody

GET - získání dat

POST - vytvoření dat

PUT - úpravy dat (upraví celý zdroj - chová se jako SET)

DELETE - smazání dat

[3] Často se můžeme setkat i s metodou **PATCH**, která se podobá metodě PUT, ale používá se pouze k částečné úpravě zdroje, kdežto u PUT vždy přenášíme zdroj celý. Tuto metodu však v originální specifikaci HTTP nenajdeme, jedná se o její rozšíření.

Některé z výše uvedených HTTP metod mají také vlastnosti idempotent nebo safe. Idempotent ve zkratce znamená, že pokud bude metoda zavolána několikrát, výsledek bude vždy stejný. Safe znamená, že metoda nijak nemodifikuje konkrétní resource.

- Idempotentní metody: GET, PUT, DELETE
- Safe (bezpečné) metody: GET

Vidíme, že použití HTTP metod ve výše zmíněném návrhu REST tyto vlastnosti splňuje.

1.2.3.2 Stavové kódy

HTTP rovněž definuje i stavové kódy odpovědí. V REST API musí stavový kód odpovědi odpovídat dané situaci. Nejčastěji se setkáváme s následujícími:

200 OK - Požadavek proběhl v pořádku (např při GET na nějaký zdroj)

201 Created - Při POST, pokud byl vytvořen nový obsah

204 No Content - Když požadavek na server proběhne v pořádku, ale server nic nevrátí

400 Bad Request - Požadavek na server je nějakým způsoben nečitelný (třeba špatný JSON apod.)

401 Unauthorized - Klient není ověřen

403 Forbidden - Klient nemá přístup k danému obsahu

404 Nod Found - Zdroj není nalezen

405 Method Not Allowed - Zdroj není dostupný pro tuto metodu. Například je možné použít GET /articles a POST /articles, ale už třeba nemůžeme článek smazat. Nelze tedy zavolat DELETE /articles - je vhodné uživatelům našeho API v tomto momentu poskytnout seznam podporovaných metod pro danou URL

410 Gone - Zdroj není už na téhle adrese dostupný - použití při verzování

429 Too-Many Requests - Pokud klient překročil maximální počet požadavků (např. za den)

Kompletní seznam lze najít v dokumentaci [4].

1.2.4 Úroveň 3. - Hypermedia Controls

Poslední pohled je znám pod akronymem **HATEOAS** (*Hypertext as the Engine of Application State*). Princip spočívá v komunikaci klientské aplikace s aplikačním serverem skrz dynamicky poskytované hypermédium. Jedná se koncový bod, který vrátí spolu s daty také odkazy na další zdroje a informuje klienta o tom, jakou akci může momentálně nad konkrétními zdroji provést.

Typickým příkladem může být bankovní účet (pokud na něm nemáme peníze, nemůžeme je vybrat). **HATEOAS** by tak klientovi dynamicky poskytl odkazy k dostupným zdrojům podle aktuálního zůstatku na účtu. V současné době ale není příliš populární a používá se jen výjimečně.

1.3 Pravidla a omezení architektury REST

Kromě základních principů zmíněných v předchozí podkapitola má REST samozřejmě svá pravidla a dost často i nepsané standardy, které je ale vhodné dodržovat. V literatuře se uvádí 6 následujících pravidel, které je potřeba dodržet v každé RESTful architektuře.

1.3.1 Client-server

V tomto omezení se setkáváme s poměrně známým pojmem Separation of Concerns, který rozděluje systém na komponenty, kdy každá z nich má jednoznačné úkoly a minimálně zasahuje do ostatních komponent.

V případě client-server architektury se bude jednat o dvě hlavní komponenty, kterými budou klient a server. Serverová část se u RESTu chová v podstatě jako rozhraní. Nabízí definovanou množinu služeb a naslouchá požadavkům klienta na tyto služby. Klient k těmto službám přistupuje zasíláním požadavků a server je může (ale nemusí) vykonat, následně zašle odpověď s informacemi o výsledku.

1.3.2 Stateless - bezstavovost

V architektuře client-server zmíněné v předchozím bodě bezstavovost znamená, že serverová část neuchovává žádná data o aktuální komunikaci (přesněji stavu komunikace) s klientem. Veškeré tyto stavy drží klientská aplikace, server si pouze může stav dočasně zrekonstruovat pomocí dat zaslaných klientem.

Toto omezení přináší řadu výhod i nevýhod. Mezi hlavní výhody patří zjednodušená implementace serverové části a vyšší spolehlivost, jelikož se server snadněji zotaví z případné chyby. Naopak hlavní nevýhodou je samozřejmě opakované zasílání některých nepotřebných dat.

1.3.3 Cache

Toto omezení má za účel zvýšit efektivitu sítě. Vyžaduje, aby server explicitně označoval vrácená data jako cacheable nebo non-cacheable. Data označená jako cacheable jsou uložena do vyrovnávací paměti a v případě opakování stejného požadavku na server jsou znovu použita. Cacheable se může používat například pro nějaké statické dokumenty. Ačkoliv tím zřejmě dosáhneme snížení interakcí mezi klientem a serverem, v jistých případech můžeme mluvit i o snížení spolehlivosti (pokud by se nečekaně změnila odpověď serveru na již cachovaný požadavek klienta).

1.3.4 Uniform interface

Jedná se o nejdůležitější požadavek, zahrnuje především základní principy popsané v předchozí podkapitole úrovní RESTu. Definuje konečnou množinu operací, které je možné nad zdroji provádět. Podrobněji lze rozdělit do 4 částí:

Identification of resources - Každý zdroj je jednoznačně identifikován pomocí standardu URI (Unique Resource Identifier) a podle toho se řídí požadavky klienta. Podrobněji viz 1. úroveň RESTu probraná na začátku této kapitoly.

Manipulation of resources through these representations - V prvé řadě je dobré uvědomit si, že reprezentace zdroje může být různá. Například strukturovaná data mohou být reprezentována pomocí formátů JSON, XML apod. Toto omezení nám však říká, že ať už je zdroj reprezentován jakkoliv, klient musí být po jeho získání schopen jej libovolně modifikovat. Se zdroji se manipuluje pomocí metod popsaných v 2. úrovni RESTu DOPLNIT ODKAZ.

Self-descriptive messages - U všech zpráv jsou používány tzv MIME standardy [5], popisující obsah zprávy. Díky tomu lze snadno identifikovat typ obsahu a zprávu tak snadno zpracovat.

HATEOAS - Vysvětlenu v 3. úrovni RESTu: 1.1.4 - Hypermedia Controls

1.3.5 Layered System

Vrstvený systém znamená rozdělení systému do několika vrstev s tím, že každá vrstva může využívat pouze služby sousední vrstvy a na další vrstvy vůbec nevidí. Toto omezení dává již dříve zmíněné client-server architektuře možnost přidávání mezivrstev, které se mohou starat například o load-balancing. Nevýhodou je samozřejmě vzniklá režie na mezivrstvách, což má za následek nižší výkon.

1.3.6 Code-On-Demand

Poslední a jediné volitelné omezení, které říká, že klient může jako odpověď na své požadavky dostat i samotný kód, který bude následně vykonán přímo u klienta. Může se jednat například o Java applety. Hlavní výhodou je samozřejmě zjednodušení rozhraní a snížení zátěže na serverové části.

1.4 RESTful Web API - Možnosti zabezpečení

[6] V předchozích podkapitolách jsme probrali základní principy a pravidla architektur. Jako další nás čeká otázka zabezpečení. Často budeme potřebovat, aby naše RESTful API podporovalo nějaké uživatelské účty a bylo co

nejlépe zabezpečené proti potencionálním škůdcům. Při návrhu zabezpečení samozřejmě musíme dbát na pravidla stanovené v předchozí kapitole. Zejména omezení stateless může být při implementaci účtů snadno porušeno, jelikož na straně serveru REST není podporován sessions managment (stavové relace mezi klientem a serverem). Možnosti zabezpečení jsou také vázány na protokol, použitý pro přenos dat v REST implementaci. Pro názorný příklad se podíváme na nejčastěji používaný protokol HTTP.

Následující sekce probírá základní možnosti implementace uživatelských účtů a zabezpečení REST architektur. Hlavní otázkou tedy bude, jak spolehlivě a bezpečně identifikovat klienta na základě jeho požadavků na server.

1.4.1 Protokol HTTP/HTTPS

Ačkoliv REST se přímo neváže na protokol HTTP, v drtivé většině případů je to pro něj základ. V případě uživatelských účtů a uživatelů se často nevyhneme případu, kdy budeme muset v HTTP požadavcích posílat citlivé údaje (jako třeba jméno + heslo). Zde se nám ale objevuje problém odposlechnutí komunikace. Potencionální útočník by tyto údaje mohl zjistit a poté snadno zneužít. Proto bude v následujících metodách, kdy nechceme, aby nějaké části komunikace unikli, důležité použít právě zabezpečený HTTPS.

HTTPS vylepšuje standardní protokol HTTP šifrováním zajištěným nejčastěji protokolem SSL nebo TLS. V dnešní době je tento protokol již poměrně rozšířený a poskytuje tak zabezpečenou variantu klasické HTTP komunikace.

1.4.2 Basic HTTP Authentication

Jedná se o velmi jednoduchý a celkem spolehlivý způsob zabezpečení, který k HTTP požadavku připojuje hlavičku Authorization. Hlavička nese jednoduše sestavený token typu Basic a je zpravidla ve formátu: Basic STRINGBASE64, kde "STRINGBASE64" je právě onen token vytvořený z přihlašovacího jména a hesla. Jednoduchý postup by tedy byl:

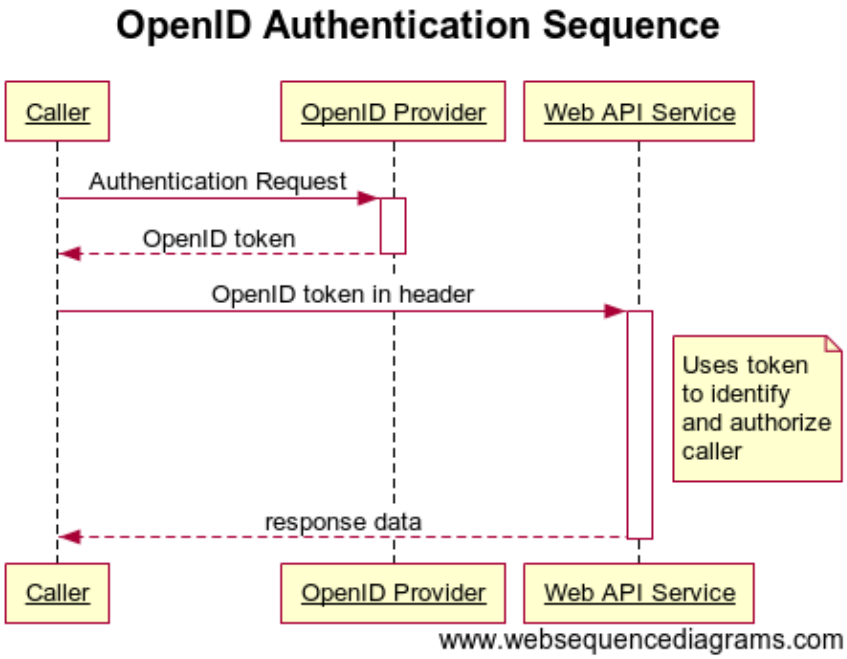
1. Zadám **jméno** a **heslo**
2. **"Jméno.heslo"** je překonvertováno do **base64**, čímž získáme výsledný token
3. Před token z minulého bodu se přidá **"Basic"** a tento řetězec se přiřadí do hlavičky **Authorization**
4. Serverová část ověří token například podle nějaké uživatelské databáze.

Tato hlavička se musí posílat při každém požadavku na server a podle ní bude ověřována identita klienta, jakékoliv session tokeny by mohli porušit omezení stateless. Je vhodné použít, pokud máme vlastní databázi účtů, vůči které chceme identitu uživatele ověřovat.

1.4.3 OpenID a OpenID-Connect

Samotný protokol **OpenID** se v RESTful architekturách **nepoužívá**, jelikož obecně nesplňuje požadavek **stateless**. Zajišťuje přihlášení uživatele a jeho principem je nechat přihlašovací údaje ověřit nějakou třetí stranou (důvěryhodnou autoritou). Klient by tedy při přihlašování na serverovou aplikaci byl přesměrován, aby se přihlásil v aplikaci důvěryhodné autority, ta by následně sdělila serverové aplikaci, že se skutečně jedná o daného klienta. V RESTu však tato přihlášení a sessiony zakazuje právě zmíněné omezení stateless. Lze však využít novou variantu tohoto protokolu, která využívá tokeny potvrzující identitu - OpenID-Connect.

V případě **OpenID-Connect**, jsme při přihlášení opět přesměrováni do aplikací ověřovací autority a po úspěšném přihlášení získáváme OpenID-Token. Takové tokeny bývají nejčastěji již typu **Bearer**, což znamená, že byl vygenerován a odeslán zpátky klientovi. Teprve až s tímto tokenem, který se opět přidá do hlavičky, se bude klientská aplikace připojovat na serverovou část REST architektury a přistupovat ke chráněným zdrojům. Token tedy slouží k potvrzení identity důvěryhodnou autoritou, kterou si klient nárokuje. Hlavní výhodou je, že můžeme používat stejné přihlašovací údaje pro více různých služeb.



Obrázek 1.2: Obrázek znázorňuje sekvenční diagram při ověření protokolem OpenID. Zdroj: `http://www.jamiekurtz.com/2013/01/14/asp-net-web-api-security-basics/`

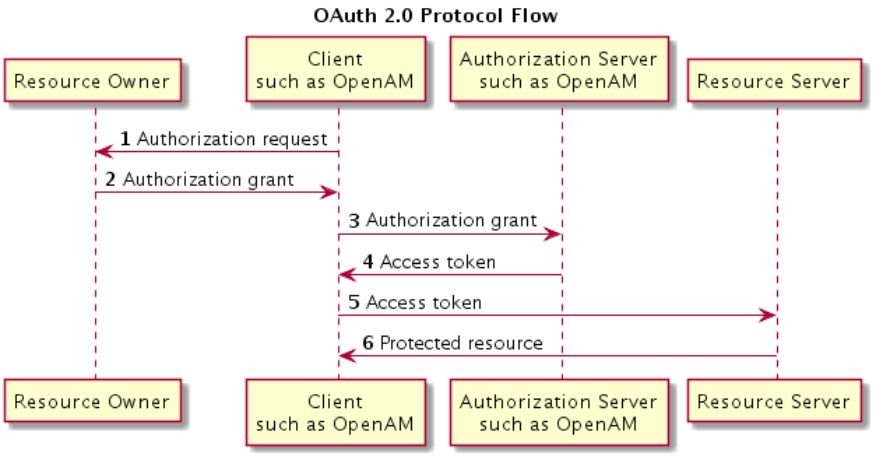
Je vhodné použít, pokud potřebujeme ověřit identitu klienta nějakou třetí důvěryhodnou stranou.

1.4.4 OAuth a OAuth 2.0

Princip je skoro stejný jako OpenID-Connect, ale hlavním rozdílem je to, že OpenID-Connect slouží k potvrzení identity uživatele, zatímco **OAuth slouží k autorizaci aplikace pro přístup k datům** autority poskytující **OAuth token**. Zde si pro snadnější pochopení pod ověřovací autoritou představíme Google, který poskytuje řadu různých služeb.

Z klientské aplikace jsme přesměrováni na Google přihlášení a následně navráceni zpět s OAuth tokenem. Klientská aplikace nikdy nezískala přímo naše přihlašovací údaje do Google účtu. Získaný OAuth token přímo neidentifikuje uživatele, ale funguje spíše jako přístupový klíč k určitým datům právě na straně ověřovací autority (v tomto případě nějakých dat na našem Google účtu). V případě REST architektury můžeme token zaslat v hlavičce na serverovou část, která poté už za našimi zády přistupuje ke Google serverům a následně k našim datům, na která jsme dali povolení. Samozřejmě ho ale lze využít také k ověření identity, zda jsme právě my vlastníci konkrétního Google účtu.

[7] OAuth 2.0 je nové vylepšení tohoto protokolu, avšak princip zůstává stejný. Jedná se o kompletní přepsání, takže jej lze brát jako zcela nový protokol. Přidává nové možnosti a specifikace, jako například definování 4 hlavních rolí - klient, autorizační server, resource server, resource owner.



Obrázek 1.3: Obrázek znázorňuje sekvenční diagram při ověření protokolem OAuth 2.0. Zdroj: `https://backstage.forgerock.com/docs/am/5/oauth2-guide/`

Je vhodné použít, pokud chceme, aby serverová část REST API spolupracovala se servery třetích stran jménem klienta.

1.5 RESTful Web API - Dokumentace

REST sám o sobě nedefinuje žádné veřejné rozhraní, které by přesně popisovalo používání daného api, jako tomu je například u SOAP protokolů, kde máme WSDL dokument. Proto je vhodné RESTful API popsat například pomocí nějakého veřejně dostupného textového nebo html dokumentu. V případě použití moderních technologií typicky použijeme nějaký framework, který většinu této dokumentace vygeneruje za nás. Každá smysluplná dokumentace by však měla obsahovat:

- URI: struktury umístění veřejně dostupných zdrojů
- Metody, které je možné nad zdroji provádět
- URL parametry, které je možné zadat(specifikovat povinné, nepovinné parametry)
- Data požadavku: například v případě POST/PUT požadavků - popsat, jak má vypadat tělo požadavku.
- Možné stavové kódy odpovědí a jejich význam
- Popis dat obsažených v odpovědi v případě konkrétního stavového kódu
- Nějaká typická ukázka požadavku a odpovědi

Konkrétní frameworky zabývající se automatickou tvorbou dokumentace budou probrány v dalších kapitolách.

1.6 Časté chyby, kterým se vyhnout

[8] REST je dnes velice populární, avšak jeho zásady jsou často porušovány. Řada aplikací, které se označují za RESTful, porušují některé z hlavních požadavků REST architektury, proto bych ještě rád shrnul nejčastější problémy, kterých se při návrhu vyvarovat.

Příliš časté využívání metody GET: Metoda HTTP GET má svá pravidla, měla by být idempotentní a bezpečná, tzn. nezáleží na tom, kolikrát ji zavoláme, výsledek vždy bude stejný (nic se po ní nezmění). Tato metoda je však často používána k zapisování nějakých dat, když URI neidentifikuje konkrétní zdroj, ale nějakou operaci, kterou bychom chtěli nad zdrojem provést. To by byla ale chyba, v takovém případě by bylo vhodné zavolat nad konkrétním zdrojem např metodu PUT.

Ignorování stavových kódu: HTTP poskytuje řadu stavových kódů, které je vhodné využívat. Chyba je, pokud API vrací například jen HTTP 200 OK. Některé API dokonce i v případě chybových hlášek použijí tento stavový kód a chybu vrátí např. v těle odpovědi. Je nutné používat korektní chybové hlášky (např HTTP-400 Bad Request) a další stavové hlášky (např HTTP-201 Created po vytvoření resource).

Ignorování cachování: HTTP má propracované cachovací mechanismy, které je vhodné využít (např If-Modified-Since header, nebo stavový kód HTTP-304-Not Modified). Cachováním lze zvýšit škálovatelnost aplikace, řada API ho však zcela vynechává.

Ignorování hypermedia: Volání do API by měla vracet alespoň nějaké linky. Například při vytvoření zdroje je vhodné v odpovědi API vrátit link právě k tomuto vytvořenému zdroji. V opačném případě by se link vytvářel na straně klienta, který by k tomu potřeboval nějaké další znalosti o API - chyba.

Ignorování MIME typů: Často se stává, že API při vracení dat zdroje používá jen jednu reprezentaci a nespecifikuje její typ. To je chyba, specifikování typu u každého zdroje je velice důležité k tomu, aby danému API mohlo porozumět co nejvíce klientů.

1.7 Shrnutí

Tato kapitola objasňuje základní vlastnosti architektury REST, tedy jak se zhruba používá, jaká pravidla je třeba dodržovat, jakým chybám se při návrhu vyhnout a dále shrnuje dokumentování REST API a možnosti zabezpečení. Jedná se spíše jen o stanovení základních znalostí ohledně architektury, které je nutné znát pro další kapitoly této práce.

Kapitola 2

Technologie a nástroje pro tvorbu RESTful API

Tato kapitola obsahuje rešerši **technologií a nástrojů, které se používají pro tvorbu RESTful API**. Jak již bylo zmíněno v předchozí kapitole, budeme se zaměřovat na RESTful API fungující standardně přes **HTTP/HTTPS Protokol**. Z hlediska klientů pak nebude problém k takovému API přistupovat, jelikož HTTP požadavky budou fungovat prakticky na všech podstatných platformách (Windows, Linux, Android, iOS, ..). Klientská část architektury pro nás tedy nebude příliš podstatná a v této rešerši se budeme zabývat technologiemi pro zprovoznění samotného API, tedy serverové části, jak je cílem mé práce.

Rešerše poslouží pro čtenáře jako **úvod do technologií pro tvorbu RESTful API aplikace** a ulehčí mu volbu technologie v případě, že by sám měl v plánu se do vývoje takové aplikace pustit. V dalších částech práce také budeme s některými těmito technologiemi pracovat prakticky, proto je vhodné se s nimi nejprve seznámit teoreticky.

Na začátku kapitoly před vybráním konkrétních technologií pro rešerši je nejprve nutné zvolit **kritéria**, podle kterých budou technologie **vybírány a hodnoceny**.

2.1 Stanovení nutných požadavků na technologie pro tvorbu REST API

Po stanovení základních požadavků této architektury v předchozí kapitole již můžeme určit prvky, které by každé RESTové API, pracující na protokolu HTTP/S, mělo nezbytně umět. Do rešerše tedy budeme vybírat technologie, se kterými není problém pracovat tak, aby splňovaly následující požadavky:

Zpracovávat HTTP požadavky Potřebujeme server, který nepřetržitě běží a dokáže zpracovávat HTTP požadavky od klientů, minimálně POST,PUT,GET

a DELETE. Server má obsloužit data z těchto požadavků a formátovat odpověď včetně stavového kódu.

Zpracovávat požadavky na různé URL Jak víme, každý zdroj musí mít své jednoznačné umístění (tedy URL). Potřebujeme tedy server, který dokáže zpracovávat požadavky na různé lokátory v rámci jeho subdomény a podle toho požadavek obsloužit.

Prace se strukturovanými daty Jak již bylo zmíněno, data budou typicky posílána ve strukturovaných formátech jako je **JSON** nebo **XML**. Vhodná technologie pro REST by ideálně měla mít alespoň základní podporu pro jednodušší práci s těmito formáty.

Persistence dat Zdrojem v restu může být cokoliv - obrázek, položka v databázi, nebo jiný soubor na disku. Server musí umět nejen tato data přijímat/odesílat, ale také je spolehlivě uchovat. Technologie by tedy měly poskytovat minimálně snadný přístup k databázi, ale ideálně také přístup k práci s daty na disku.

Výše zmíněné prvky dnes již podporuje nespočetně technologií. Pro tuto rešerši budou vybírány technologie hlavně podle jejich současné popularity použití v architektuře REST. Zaměříme se tedy na hlavní programovací jazyky, používající se při tvorbě webových aplikací a ke každému jazyku podrobněji probereme jednu populární REST API technologii (většinou se bude jednat o nějaký framework), ostatní technologie pak uvedeme jen okrajově. Mezi populární jazyky při tvorbě webových aplikací lze zcela jistě zařadit: **C#, Java, Ruby, Python, Node.JS**.

2.2 Stanovení kriterií pro rešerše technologií

Rešerše bude probíhat tak, že každá vybraná technologie bude prozkoumána podle informací dostupných na webu a ve většině případech i odzkoušena na jednoduchém projektu - REST API pro blog s články a komentáři. Dále budou popsány její základní principy, tedy s čím se setkáme, či jak budeme postupovat v případě, že tuto technologii použijeme, hlavní výhody, nevýhody, a také proběhne hodnocení podle následujících kritérií:

Jednoduchost a intuitivnost Jak obtížné a intuitivní je za pomocí této technologie vytvoření funkčního RESTful API. *(1 - lehké a intuitivní, 5 - velmi obtížné)*

Dokumentace Jak dobře je daná technologie zdokumentovaná, tedy pokud nebylo obtížné ji za pomoci dostupné dokumentace pochopit a správně použít. *(1 - výborná dokumentace, 5 - špatná nepřehledná dokumentace)*

Spolehlivost Zda má technologie nějaká známá bezpečnostní rizika, nebo je náchylná k výpadkům. *(1 - spolehlivá bezpečná technologie, 5 - technologie má známé problémy)*

Popularita Jak často je tato technologie používána v tématice RESTful API. Zde budeme brát v úvahu, jak často je technologie zmiňována na diskuzních serverech typu `stackoverflow.com` *(1 - rostoucí vysoká popularita, 5 - nízká popularita, nebo v posledních letech značně klesá)*

Jak již je z popisků zřejmé, tato kritéria pak u každé technologie budou zhodnocena číslem od **1 (velmi dobré)** do **5 (velmi špatné)**, . Každé uložené hodnocení bude také odůvodněno.

2.3 Technologie: C# - ASP.NET / ASP.NET Core

Obrázek 2.1: Logo ASP.NET Core. Zdroj: `https://docs.microsoft.com/en-us/dotnet/`

[11] Technologie **ASP.NET** či její novější verze nyní představovaná jako **ASP.NET Core** byla vybrána k prozkoumání, jelikož se jedná o nejtypičtějšího a nejčastějšího zástupce, pokud chceme vyvíjet různé webové API na základech programovacího jazyka **C#** a frameworku **.NET**.

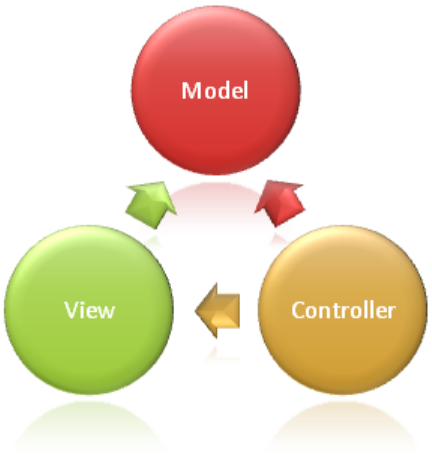
ASP.NET Core je tedy open-source webový framework vyvíjený společností Microsoft a jeho komunitou. Jeho typické použití zahrnuje jak webové rozhraní (tedy včetně RESTful API), tak i další moderní webové či cloudové aplikace.

2.3.1 Principy technologie

První podstatnou pomůckou pro práci s tímto frameworkem je vývojové prostředí, kterým je **Visual Studio**[12]. To je vyvíjeno rovněž firmou Microsoft, která tak poskytuje co nejsnadnější práci s frameworkem a jako bonus i snadný přístup k jeho dalším službám poskytnutým pomocí cloudové výpočetní platformy **Microsoft Azure**[13]. Tu si podrobněji probereme také v další kapitole

podpůrných technologií pro tvorbu RESTful API. Dále je zde snadný import dalších knihoven a frameworků pro .NET.

Samotná práce s frameworkem ASP.NET Core se pak zakládá na použití **Model-View-Controller**[14] architektury, tedy rozdělením aplikace do tří hlavních součástí:



Obrázek 2.2: Obrázek přibližuje princip architektury Model-View-Controller. Zdroj: `https://docs.microsoft.com/cs-cz/aspnet/core/mvc/overview/_static/mvc.png`

Model Představuje stav aplikace, tedy data, se kterými REST API pracuje.

View Zajišťuje prezentaci současného stavu aplikace. Měl by mít minimální logiku, může se jednat o prosté zobrazení HTML kódu. V případě REST API pro nás není tato část tolik podstatná, protože v něm nepracujeme s grafickou stránkou, ale s daty v přenosových formátech (např JSON)

Controller Komponenty, které zpracovávají interakci s uživatelem, která se pak promítne na změně modelu. V případě REST API sem řadíme veškeré zpracování HTTP požadavků.

Jednoduché REST API vytvořené pomocí frameworku ASP.NET Core by tedy vypadalo tak, že nadefinujeme několik základních controllerů obstarávající HTTP požadavky na klíčové zdroje *(např: PUT:/articles, nebo GET:/comments/[id])* a ke každému z těchto zdrojů, který se má chovat jako kolekce, vytvoříme nějaký model.

Model pak bude standardní třída reprezentující strukturu daného zdroje. Mezi jednu z hlavních výhod ASP.NET Core patří také **EntityFramework**[15], který nám umožní snadná ukládání a načítání těchto objektů z databáze. Další výhodou těchto objektů je to, že není nutné je při vracení konvertovat do přenosových formátů (JSON,XML), ASP.NET Core toto provádí automaticky.

Stejně tak je ale připraveno i na drtivou většinu všech dalších záležitostí, které při tvorbě REST API řešíme, např. připravené třídy pro vracení stavů apod.

Kromě toho lze při práci s tímto frameworkem pracovat s řadou dalších balíčků o základu C#, .NET. Ve VisualStudiu je to více než snadné a to s použitím **NuGet Package Manageru**[16]. ASP.NET Core má také dobrou podporu pro asynchronní práci s metodami async/await.

2.3.2 Ukázky kódu

Model pro zdroj typu article (článek):

```
namespace BlogApi.Models
{
    public class ArticleModel
    {
        public long Id { get; set; }
        public string Title { get; set; }
        public string Content { get; set; }
    }
}
```

Metoda GET articles v controlleru pro články:

```
[Route("api/[controller]")]
public class ArticlesController : Controller
{
        // implementuje GET /api/articles
        [HttpGet]
        public IEnumerable<ArticleModel> GetAll()
        {
                // nacitani z databaze bylo pro ukazku
                // trochu zjednoduseno
                return DB.ArticleItems.ToList();
        }
}
```

2.3.3 Hlavní výhody

Microsoft Azure Snadné napojení na cloudové služby Microsoft Azure přináší řadu výhod a funkcí, které můžeme využít. Za nejpodstatnější výhodu považuji snadné provozování aplikace přímo na cloudu, tedy serverech k tomu přímo určených a vysoce spolehlivých.

Rozsáhlá dokumentace i v češtině Microsoft a jeho komunita poskytuje pro ASP.NET Core opravdu rozsáhlou dokumentaci a velká část z ní je dostupná i v češtině.

Řada dostupných komponentů Můžeme využívat řadu dalších komponentů vytvořených v jazyce C# nebo frameworku .NET.

Asynchronní orientace Jednoduchá práce s asynchronně volanými funkcemi.

2.3.4 Hlavní nevýhody

Operační systém Jelikož je vyvíjen společností Microsoft, můžeme logicky očekávat nižší podporu jiných operačních systému než je Windows.

Kompatibilita mezi verzemi frameworku Nižší kompatibilita mezi různými verzemi frameworku, aktualizování projektů může být složité, hlavně pokud se uživatel rozhodne aktualizovat z ASP.NET na ASP.NET Core

2.3.5 Zhodnocení podle kritérií

Jednoduchost a intuitivnost - 2 Jednoduchá architektura MVC, množství ulehčujících prvků, se kterými je ale nutné se naučit pracovat.

Dokumentace - 1 Výborná dokumentace v několika jazycích od Microsoftu a jeho komunity.

Spolehlivost - 2 [18] Vysoké zabezpečení obsahuje funkce pro správu ověřování, autorizace, ochrany dat, SSL vynucení, tajné klíče aplikace, žádost proti padělání, ochrana a správa CORS. Je možné za příplatek snadno hostovat na spolehlivých serverech cloudu Microsoft Azure. ASP.NET Core je nová technologii, tudíž může obsahovat ještě nevyladěné chyby.

Popularita - 2 [42][17] Co se týče ASP.NET Core, jedná se sice o novou, ale na popularitě rychle rostoucí technologii. Starší ASP.NET kdysi bývalo dominantním nástrojem na poli webových aplikacích, avšak jeho sláva se s klesající podporou chýlí ke konci.

Mezi další populární technologie pro jazyk C# lze řadit:

- Nancy - `http://nancyfx.org/`
- Manos - `https://github.com/jacksonh/manos`
- a další..

2.4 Technologie: Java - Spring

Obrázek 2.3: Logo Java Spring. Zdroj: `https://spring.io/`

[19] Jako hlavní zástupce technologií pro jazyk **Java** byl vybrán framework **Spring**. Jedná se o open-source framework pro vývoj J2EE aplikací, které zahrnují podnikové aplikace, informační systémy, a pro tuto práci také poskytuje vysokou podporu pro RESTful služby.

2.4.1 Principy technologie

Framework jako takový je organizovaný do několika modulů. Hlavní skupiny těchto modulů, které budou pro tuto práci podstatné, jsou:

Core Container Základní části frameworku

Data Access / Integration Poskytuje přístup k datům (např JDBC - Java Database Connectivity)

Web Poskytuje veškeré funkce pro webové aplikace, včetně podpory pro RESTful webové služby. Základem těchto služeb je zde **Model-View-Controller** architektura podobně jako v předchozí sekci u technologie **ASP.NET Core**

Celý princip práce s tímto frameworkem je tedy velice podobný jako u předchozího ASP.NET Core. Na začátku vytvoříme controller, obstarávající HTTP požadavky na nějaký zdroj (např: PUT:/articles, nebo GET:/comments/[id]), a opět ke každému zdroji vytvoříme nějaký model, což bude tentokrát Java třída.

Vrstvu **View** zde také nemusíme nijak výrazně řešit, protože pracujeme např. s JSON objekty a o potřebné se postará ve Springu zabudovaná knihovna **Jackson-JSON**[20], která převádí instance Java objektů do jejich JSON reprezentace.

Perzistance dat pak zajišťuje datová vrstva frameworku, která umožňuje snadný přístup k několika druhům databází, včetně např. **MongoDB**[21], která pracuje přímo s JSON objektu a pro RESTové API s těmito objekty pracující, pak může být více než zajímavá.

Rozdíl oproti ASP.NET Core je také možnost volby různých IDE (vývojových prostředí). Zatímco ASP.NET Core se zaměřuje na Visual Studio, které pro něj má nejvyšší podporu, v případě Springu lze sáhnout hned po několika různých IDE, avšak ty pro něj nebudou mít tak velkou podporu jako např. výše zmiňované Visual Studio pro ASP.NET.

Práci nám zde usnadní také velká řada balíčků vytvořených pro jazyk Java, k těm se snadno dostaneme pomocí buildovacích nástroju **Gradle**[22] nebo **Maven**[23]

2.4.2 Ukázky kódu

Model pro zdroj typu article (článek):

```
public class Article {
        public final long id;
        public final String title;
        public final String content;
        // pripadne pouzijeme private a getter + setter
}
```

Metoda GET article podle id v controlleru pro články:

```
@RestController
public class ArticlesController {
        @RequestMapping("/articles")
        public Article getArticle(@RequestParam(value="id",
         defaultValue="0") int id)      {
                // nacteme podle ID, ale opet v ukazce
                // nebudeme rozebirat nacteni z databaze
                return new Article();
        }
}
```

2.4.3 Hlavní výhody

JSON Databáze Možnost snadno napojit na JSON databáze typu MongoDB.

Řada dostupných komponentů Můžeme využívat řadu dalších komponentů vytvořených v jazyce Java.

Rozsáhlost Rozsáhlý a rozšířený framework.

2.4.4 Hlavní nevýhody

Asynchronní operace Nemá tak dobrou podporu pro asynchronní funkce jako např. ASP.NET Core.

Orientace ve frameworku celkově Jelikož se jedná o velice rozsáhlý framework, může zabrat nějakou dobu se v něm zorientovat jako v celku.

2.4.5 Zhodnocení podle kritérií

Jednoduchost a intuitivnost - 3 Jednoduchá architektura MVC, ale např. oproti ASP.NET Core chybí snadné napojení na cloud a horší práce s asynchronními funkcemi.

Dokumentace - 2 Dobrá dokumentace na stránkách projektu (spring.io), avšak není tak rozsáhlá jako např. u ASP.NET Core.

Spolehlivost - 1 [24] Poskytuje vysoké možnosti zabezpečení, ve firmách je Spring i Java EE vysoce uznávána a brána za spolehlivou technologii používající se i u bankovních systémů.

Popularita - 1 [42]Celkově je Java EE brán jako jeden z nejpopulárnějších jazyků pro tvorbu webových aplikací, výjimkou ve vysoké popularitě není ani Spring framework, jehož popularita v posledních letech stále roste.

Mezi další populární technologie pro jazyk Java lze řadit:

- Play Framework - `https://www.playframework.com/`
- RESTEasy - `http://resteasy.jboss.org/`
- Restlet - `https://restlet.com/`
- a další..

2.5 Technologie: Ruby - Rails

Obrázek 2.4: Logo Rails. Zdroj: `http://rubyonrails.org/`

Rails[33] je framework založený na jazyce **Ruby**.Je určený pro vývoj webových aplikací napojených na databázi, používající architekturu **model-view-controller**. Je znám hlavně pro jednoduché a rychlé generování kódu, celé

webové aplikace můžeme vygenerovat pouze s použitím několika příkazů. Pro RESTful API se doporučuje použít jeho novější verze, které umožňuje snadné ignorování vrstvy **view**, která je zde zbytečná, a vytvořit tak webovou aplikaci zpracovávající pouze HTTP requesty.

2.5.1 Principy technologie

Jak již bylo zmíněno, jedná se o model-view-controller architekturu, tím pádem se podobá oběma předchozím frameworkům (ASP.NET Core a Java Spring). Při práci s Rails a obecně i v Ruby se doporučuje pracovat v **Linuxovém** prostředí, jelikož je zde daleko jednodušší instalace všech potřebných komponentů, než je tomu například v případě Windows. Pakliže máme Rails zprovozněné, můžeme začít stavět základy našeho RESTful API za pomocí příkazové řádky.

Nejprve je třeba vytvořit samotný projekt. Následující příkaz vytvoří základ pro naše ukázkové API pro jednoduchý blog. Parametrem *–api* udáváme, že se bude jednat pouze o webové rozhraní zpracovávající HTTP požadavky, bez vrstvy **view**.

```
rails new blog-api --api
```

Následně přichází na řadu vrstva **model**, tedy vygenerujeme pro náš projekt základní třídy modelů. Podíváme se i na jednoduché reference mezi modely. Tedy model komentářů **Comment** bude vždy přiřazen k nějakému článku **Article** - k tomu poslouží typ attributu **references**.

```
rails g model Article title:string content:string
rails g model Comment author:string text:string
 article:references
```

Předchozí příkazy nám také vytvořili migrace, které pak jednoduše použijeme pro aktualizování struktury databázi. Vytvořené třídy v jazyce Ruby pak vypadají následovně:

```
# app/models/comment.rb
class Comment < ApplicationRecord
  # model association
  belongs_to :article

  # validation
  validates_presence_of :author, :text
end
```

Controllery, neboli třídy zpracovávající samotné HTTP požadavky k našim již vytvořeným modelům pak už vytvoříme jednoduše:

```
rails g controller Articles
rails g controller Comments
```

Controllery v Ruby pak vypadají následovně, můžeme si také všimnout velice snadného přístupu do databáze.

```
# GET /articles
def index
  @articles = Article.all
  json_response(@articles)
end
```

Celý vygenerovaný projekt obsahuje však spoustu dalších souborů než jen těchto několik tříd, avšak tyto třídy jsou pro nás do začátku nejpodstatnější. Naše vygenerované RESTful API pak již můžeme snadno spustit a dolaďovat podle našich představ.

V Rails také jistě využijeme testovací framework **RSpec**[34]. Práci nám usnadní i množství dalších snadno dostupných balíčků pro jazyk Ruby (tzv. **gemů**).

2.5.2 Hlavní výhody

Generování kódu Rychlé a snadné generování kódu nám umožní vytvořit jednoduché RESTful API už během několika minut.

Řada dostupných komponentů Můžeme využívat řadu dalších komponentů vytvořených v jazyce Ruby, tzv gemů.

Tutoriály Zajímavé interaktivní výukové tutoriály Rails for Zombies, dostupné zde: `http://railsforzombies.org/`.

Přístup ke kolekcím Snadný přístup ke kolekcím do databáze.

2.5.3 Hlavní nevýhody

Windows O něco horší podpora pro windows, nutnost instalace spousty dodatečného software.

Přehlednost Vygenerované kódy v Ruby jsou dle mého názoru méně přehledné než tomu bylo např u ASP.NET Core nebo Java Spring.

Výkon Jedná se o velice pomalý framework ve srovnání s ostatními technologiemi.

2.5.4 Zhodnocení podle kritérií

Jednoduchost a intuitivnost - 1 Jednoduchá architektura MVC, mnoho zlehčujících prvků, se kterými je ale nutné se naučit pracovat.

Dokumentace - 1 Velice rozsáhlá dokumentace, k dispozici jsou také zajímavé výukové interaktivní tutoriály.

Spolehlivost - 3 [35][36] Jedná se o dlouholetě testovanou a bezpečnou technologii, nicméně v posledních letech od ní velké firmy upouští. Nižší spolehlivost tohoto frameworku způsobuje hlavně nižší výkon jazyka Ruby.

Popularita - 3 [42][35][36]Před lety se jednalo o nejpopulárnější framework pro tvorbu obdobných webových aplikací, avšak dnes je jeho popularita již na sestupu. Velké společnosti raději volí jiné alternativy, na vině je výkon jazyka Ruby, který zaostává oproti jeho alternativám.

Mezi další populární technologie pro jazyk Ruby lze řadit:

- Grape - `https://github.com/ruby-grape/grape#what-is-grape`
- Awesome Ruby API - `https://github.com/edymerchk/awesome-ruby-api`
- a další..

2.6 Technologie: Python - Django REST framework

Obrázek 2.5: Logo Django REST framework. Zdroj: `http://www.django-rest-framework.org/`

[37]**Django** je open source webový aplikační framework napsaný v Pythonu, který se volně drží architektury **model-view-controller**. Django byl navržen tak, aby umožňoval vývojářům co nejrychlejší dokončení aplikace z jejího návrhu. Také je vysoce zaměřen na bezpečnost a pomáhá vývojářům vyhnout se spoustě známých bezpečnostních chyb. V případě tvorby RESTful API aplikací se podíváme na jeho konkrétní specializovaný modul - **Django REST framework**[38].

2.6.1 Principy technologie

Ani v tomto frameworku nebude výjimkou použitá model-view-controller architektura. K zprovoznění frameworku je potřeba mít nainstalovaný Python a základní Django framework, jejich zprovoznění je opět snazší v **Linuxových** prostředích. Můžeme pracovat i ve Windows, bude však nutné instalovat další dodatečný software.

V Djangu můžeme vytvořit více aplikací najednou a následně je sloučit do jedné aplikace. Základní Django projekt můžeme vytvořit jednoduše, příkazem:

```
django-admin startproject djangorest
```

Výše zmíněný příkaz nám vytvoří balíček s několika **Python** soubory. Pro tuto ukázku přeskočíme některé další příkazy pro samotné zprovoznění základů frameworku a podíváme se rovnou na tvorbu **modelu**. Model bude, jak by se dalo očekávat, Python třída *articles/models.py*:

```
class Article(models.Model):
 title = models.TextField()
 content = models.TextField()
```

Předpokládejme, že připojení na databázi už máme plně vyřešené a funkční. Django nám pak k tomuto modelu může snadno vytvořit migrace, které pak aktualizují tyto modely do databáze:

```
python manage.py makemigrations articles
python manage.py migrate
```

V tomto frameworku se však navíc musíme zabývat **serializací**, což je z pohledu architektury v podstatě vrstva **view**. Serializer je tedy Python třída starající se o převod z nějakého serializovaného formátu (např JSON) do Python objektů a opačně. Jejich kód pro tuto ukázku není příliš zajímavý. Uvažujme, že máme klasický JSON serializer pro články, tedy **ArticleSerializer**. S tímto serializerem by se pak pracovalo v **controller** vrstvě následovně:

```
"""
List all articles.
"""
if request.method == 'GET':
 articles = Article.objects.all()
 serializer = ArticleSerializer(articles, many=True)
 return JsonResponse(serializer.data, safe=False)
```

Podmínka (viz výše) by pak umožňovala vrácení všech článků v JSON formátu v případě GET requestu. Vidíme, že je zde opět velice snadný přístup ke všem objektům z databáze, podobně jako tomu bylo u frameworku Rails.

Co je na frameworku trochu zvláštní, je také způsob routování HTTP požadavků, stará se o to speciální Python soubor, v našem případě **articles/urls.py**, za pomocí url patternů (vzorů):

```
urlpatterns = [
    url(r'^articles/$', views.article_list),
    url(r'^articles/(?P<pk>[0-9]+)/$', views.article_detail),
]
```

Celkově se framework nejvíce podobá předchozímu frameworku **Rails**. Práce s ním mi však přijde trochu složitější, avšak Python může nabídnout o něco vyšší spolehlivost (hlavně z rychlostního hlediska) než Ruby. K užitku také bude spoustu dodatečných balíčků pro jazyk Python.

2.6.2 Hlavní výhody

Generování kódu Rychlé a snadné generování projektů, migrací pomáhá k celkovému zrychlení prací na projektu.

Přístup ke kolekcím Snadný přístup ke kolekcím do databáze.

Zabezpečení Vysoké zaměření na bezpečnost, pomáhá vývojářům vyhnout se spoustě známých bezpečnostních chyb.

2.6.3 Hlavní nevýhody

Windows O něco horší podpora pro Windows, nutnost instalace velkého množství dodatečného software.

Přehlednost Opět jistá nepřehlednost vygenerovaných projektů.

Serializace Nutnost explicitně řešit serializace Python objektů. Dle mého názoru je to trochu zbytečná práce navíc, ve všech předchozích frameworcích nebylo potřeba.

2.6.4 Zhodnocení podle kritérií

Jednoduchost a intuitivnost - 3 Sice obsahuje generování kódu, které by práci mělo značně zjednodušit, avšak například oproti Rails frameworku mi přijde značně složitější, jelikož řešíme další požadavky jako je serializace Python objektů.

Dokumentace - 2 Dobrá dokumentace jak samotného Djanga, tak modulu Django REST framework.

Spolehlivost - 1 Poskytuje vysoké možnosti zabezpečení, Python je poslední dobou stále populárnější a je považován za velice spolehlivý jazyk.

Popularita - 2 [42]Popularita frameworku je značně vysoká, v poslední době již překročila dříve dominantí Rails framework, který je teď na sestupu, avšak popularita Java - Spring roste stále rychleji.

Mezi další populární technologie pro jazyk Python lze řadit:

- Bottle - `http://bottlepy.org/`
- Flask - `http://flask.pocoo.org/`
- Falcon - `http://falconframework.org/`
- a další..

2.7 Technologie: NodeJS - ExpressJS

Obrázek 2.6: Loga ExpressJS, NodeJS a MongoDB. Zdroj: `https://medium.com/@sakthivel9393/besics-understanding-of-express-js-node-js-express-js-fast-and-easy-web-framework-for-node-js-8cde82d7aa7e`

[39] **Node.js** je softwarový systém navržený pro psaní vysoce škálovatelných internetových aplikací, především webových serverů. Programy pro Node.js jsou psané v jazyce **JavaScript**, hojně využívající model událostí a asynchronní I/O operace pro minimalizaci režie procesoru a maximalizaci výkonu. [40] **ExpressJS** je pak open source framework pro NodeJS určený pro tvorbu webových aplikací, včetně vysoké podpory pro samotné RESTful API.

2.7.1 Principy technologie

Jednou z hlavních výhod **NodeJS** může být i jeho snadnější instalace na více prostředích. V **Linuxu** ho snadno zprovozníme příkazovou řádkou, přičemž na **Windows** si můžeme z oficiálních stránek stáhnout software, který nám poskytne příkazovou řádku pro NodeJS (**npm** příkazy) a následně můžeme postupovat obdobně snadno jako na Linuxu. Pro ukázku se můžeme podívat na příkaz pro instalaci frameworku express:

```
npm install express --save
```

Když už máme zprovozněný NodeJS včetně frameworku Express, můžeme se podívat na samotnou tvorbu RESTful API, v jejímž případě se opět dostáváme do model-view-controller architektury. Již jsme probírali formát JSON, takže víme, že javaskript nemá nijak striktně předdefinované formáty objektů. Co se týče modelů, použijeme knihovnu **Mongoose**, která umožňuje snadné napojení na **MongoDB** databázi a navíc umožňuje definovat schéma zdrojů:

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var ArticleSchema = new Schema({
  title: {
    type: String,
    required: 'Kindly fill title of Article'
  },
  content: {
    type: String,
    required: 'Kindly fill content of Article'
  }
});

module.exports = mongoose.model('Articles', ArticleSchema);
```

Vidíme, že za pomocí Mongoose lze pak na JSON objekty i definovat různé **integritní omezení**. Co nás může zajímat dále, je samotné routování HTTP requestů - ve frameworku express by routovací funkce vypadaly následovně:

```
var articleList = require('../controllers/articleListController');

app.route('/articles')
 .get(articleList.list_all_tasks)
 .post(articleList.create_a_task);

app.route('/articles/:articleId')
 .get(articleList.read_a_task)
 .put(articleList.update_a_task)
 .delete(articleList.delete_a_task);
```

Vidíme, že použité routování nás již rovnou směruje do vrstvy **controller**. Pro ukázku vezmeme metodu v controlleru sloužící pro vrácení všech článků ve formátu JSON:

```
var mongoose = require('mongoose'),
  Article = mongoose.model('Articles');
```

```
exports.list_all_tasks = function(req, res) {
  Article.find({}, function(err, article) {
    if (err)
      res.send(err);
    res.json(article);
  });
};
```

MongoDB přímo pracuje s formétem JSON, což nám pak umožní snadné dotazování nad kolekcí Articles i v případě dalších parametrů. Vidíme, že serializace do formátu JSON je také velice jednoduchá a nemusíme ji řešit složitě jako v Pythonu.

Toto vše postačí k tomu, aby byl základ našeho RESTful API hotový, následně již zbývá jen nastavit správné parametry (MongoDB připojení, port) a vše můžeme spustit.

ExpressJS pak poskytuje spoustu dalších funkcí, jako například přidávání dodatečného middleware, jehož cílem je zajistit co nejlepší výkon a škálovatelnost.

2.7.2 Hlavní výhody

Windows + Linux Dobrá podpora jak pro Windows, tak pro Linux.

JSON + MongoDB Snadné napojení na MongoDB databázi a snadná práce s JSON objekty, které jsou v RESTful API nejpopulárnější.

Škálovatelnost Vysoká škálovatelnost, množství dalšího dodatečného middleware a funkcí. Dobrá podpora pro asynchronní volání funkcí

NodeJS komunita NodeJS má celkem rozsáhlou komunitu, zajišťující dobrou podporu při tvorbě aplikací.

2.7.3 Hlavní nevýhody

Javascript Dle mého názoru je Javascript trochu nestandardní jazyk a chová se trochu jinak, než by člověk, který zná předchozí prozkoumané jazyky (Java, C#), očekával.

2.7.4 Zhodnocení podle kritérií

Jednoduchost a intuitivnost - 2 Práce s frameworkem se dá rychle naučit, avšak uživatelé, kteří neznají javascript, mohou být překvapeni z jeho nestandardního chování.

Dokumentace - 1 Dobrá dokumentace od vývojářů, včetně rozsáhlé komunity, zajišťující další podporu při tvorbě aplikací.

Spolehlivost - 1 [41] ExpresJS poskytuje vysokou škálovatelnost a další moduly zabraňující známým chybám v protokolu HTTP.

Popularita - 3 [42]NodeJS je technologie, jejíž popularita je na vzestupu, dá se očekávat, že překročí dříve velice populární Ruby - Rails, avšak vývoj frameworku Express je občas nestálý, což mu popularitu trochu snižuje.

Mezi další populární technologie pro jazyk Node.js (Javascript) lze řadit:

- Sails.js - `https://sailsjs.com/`
- Restify - `http://restify.com/`
- Hapi.js - `https://hapijs.com/`
- a další..

2.8 Shrnutí

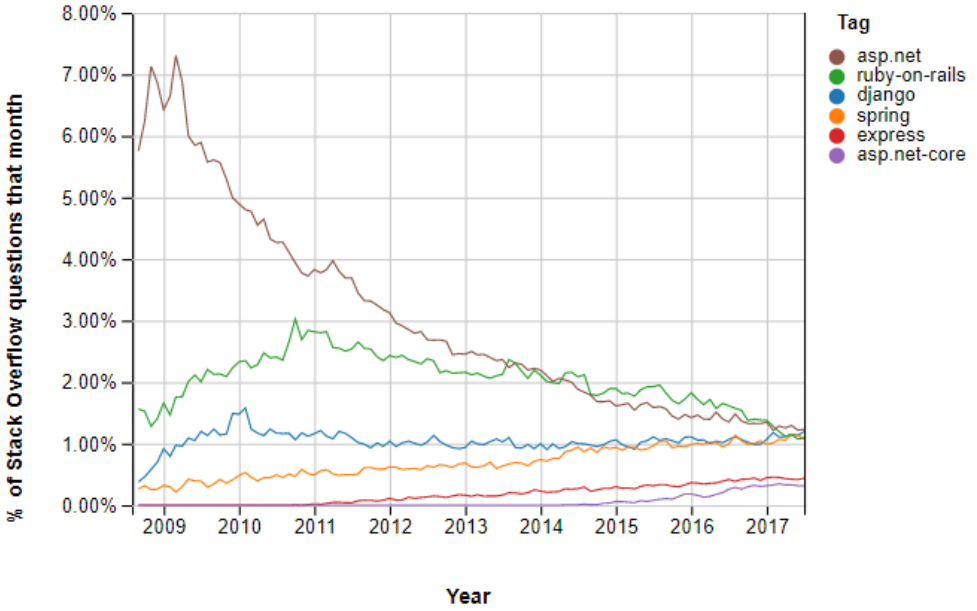
Tato kapitola shrnuje a zhodnocuje populární frameworky pro tvorbu RESTful API aplikací. Čtenář po přečtení získává základní přehled těchto technologií a je mu ulehčen výběr v případě, že by některé z nich sám chtěl v budoucnu použít. V dalších částech práce také budeme na některé tyto technologie navazovat a používat je v praktické části.

Následující tabulky a grafy shrnují technologie probrané v této kapitole, i některé další zajímavé technologie, které však nebyly podrobně zkoumány:

	Jazyk	K1	K2	K3	K4
ASP.NET Core	C# (.NET)	2	1	2	2
Spring	Java	3	2	1	1
Rails	Ruby	1	1	3	3
Django REST	Python	3	2	1	2
ExpresJS	NodeJS (JavaScript)	2	1	1	3

- **K1** - Jednoduchost a intuitivnost
- **K2** - Dokumentace
- **K3** - Spolehlivost
- **K4** - Popularita

Tabulka 2.1: Tato tabulka zhodnocuje technologie podrobně probrané v této kapitole podle předem stanovených kritérií. Důvody udělení daného hodnocení byly v kapitole uvedeny, vychází hlavně z mého subjektivního názoru podle odzkoušení technologie a z informací nalezených na internetu. Hodnocení je na škále od **1 - velmi dobré** do **5 - velmi špatné**. Lze vidět, že žádný framework nedosáhl výrazně špatných hodnocení (4 nebo hůře)



Obrázek 2.7: Popularita frameworků určená z popularity tagů na `stackoverflow.com`. Zdroj: `https://insights.stackoverflow.com/trends`

Jazyk	Zajímavé frameworky
C#	**ASP.NET Core**, Nancy, Manos
Java	**Spring**, Play Framework, RESTEasy, Restlet
Ruby	**Rails**, Grape, Awesome Ruby API
Python	**Django**, Bottle, Flask, Falcon
JavaScript	**ExpressJS**, Sails.JS, Restify, Hapi.JS
php	Laravel, Lumen, Wave Framework, Silex

Tabulka 2.2: Tato tabulka shrnuje další zajímavé frameworky pro tvorbu RESTful API aplikací vzhledem k vybranému programovacímu jazyku. Tučně označené byly podrobně probrány v této kapitole.

Kapitola 3

Další podpůrné technologie

Tato kapitola shrnuje další **technologie a nástroje používané při tvorbě RESTful API**. Rozdílem oproti předchozí kapitole je to, že tyto technologie nejsou použity k samotné implementaci funkční aplikace a jejich využití tím pádem není nutné. Přináší však řadu výhod a v praxi jsou běžně používány, proto je vhodné se jimi zabývat.

Kapitola tedy poslouží pro čtenáře jako **shrnutí nástrojů a jejich výhod**, kterých lze při tvorbě REST API dosáhnout. Z těchto technologií také budeme v další kapitole některé vybírat do sestavy technologií pro nejvyšší míru podpory tvorby RESTful API a následně je použijeme v praktické části práce, tudíž je nejprve vhodné je prozkoumat.

Při zkoumání technologií budeme postupovat tak, že nejprve vybereme oblast, v jaké je tato technologie nápomocná, objasníme hlavní přínosy při implementaci REST API (tedy proč se touto oblastí vůbec zabývat) a na závěr vybereme 1 nebo 2 konkrétní technologie, které podrobněji probereme. Tyto oblasti budeme stanovovat postupně podle průběhu **životního cyklu vývoje aplikací** (tedy co se nám hodí na **začátku při návrhu**, **následně při implementaci** a **údržbě takového software**)

3.1 Mockup pro RESTful API

Před samotnou implementací výsledného RESTful API je nezbytné zabývat se jeho návrhem a zdokumentováním tohoto návrhu. V běžných aplikacích mohou být při navrhování použity různé diagramy tříd a podobně, avšak z předchozích kapitol víme, že REST architektura a frameworky stanovují většinu návrhu za nás. Pokud se nejedná o rozsáhle komplikovanou aplikaci, na které by pracoval široký tým vývojářů, tak různé architektonické nástroje typu Enterprise architect postrádají význam.

Ve většina případů si vystačíme s jednoduchým, srozumitelně popsaným a jednoznačným návrhem RESTful API aplikace, který mohou představovat Mockupovací technologie. [10] Mocking se v IT obecně chápe jako simulování

funkčnosti nějakého komponentu. Mockupovaný objekt se může chovat tak, že nic nepočítá, nemá žádnou logiku, ale vrací zdánlivě správná data. Obecně se mockupování používá často při testování, kdy chceme mít jistotu, že nám objekt, na kterém testování závisí, vrátí data vhodná pro zbytek testování. V rámci REST architektury má mocking spoustu dalších využití než jen testování. V dnešní době existuje mnoho technologií, které poměrně snadno umožňují namockupovat celou serverovou část RESTového API. V této kapitole se budu zabývat účelem a analýzou technologií pro mockupování.

3.1.1 Výhody a použití

V případě namockupovaného REST API získáváme jeho zdánlivě funkční simulaci, s kterou se pracuje stejně tak, jak se bude pracovat s výsledným REST API. Data, která vrací, jsou však jednoduchá, odpovídají alespoň stejnému formátu dat finálních. Každého programátora již určitě napadne několik způsobů, jak toto nasimulované API použít. Ve většině případů bude vhodné mockupovat před konečnou implementací API, nikoliv až po ní. Mezi možnosti použití patří:

Dokumentace API Ve většině mockupovacích technologií alespoň jednoduše popisujeme dostupné metody/volání tohoto nasimulovaného API. Takový výsledek se pak dá použít k dokumentaci, která bude mít i praktické nasimulované ukázky funkčnosti. To zajistí lepší pochopení výsledného API.

Zjednodušený vývoj klientů Klientské části REST architektury implementačně závisí na serverové části. V případě rychlého namockupování serverové části vývoj klientů nemusí čekat na její úplné dokončení, ale můžou se dočasně napojovat na její mockupovou simulaci. To zjednoduší a zrychlí vývoj komponentů na straně klienta.

Testování Mockup má řadu výhod z hlediska testování. V simulovaném REST API můžeme snadno nastavit data, která chceme, aby API simulovalo. Ve výsledném API bychom např. museli data nejprve vkládat do databáze přes různé POST/DELETE operace.

3.1.2 Technologie: Apiary

Obrázek 3.1: Logo Apiary. Zdroj: `https://apiary.io/`

[15] **Apiary.io** je snadno použitelný nástroj pro zjednodušení návrhů webových API. Poskytuje jednak dobrou dokumentaci pro navržené API, ale také **Mockupovaný server**, kde si naše navržené API můžeme snadno odzkoušet.

Celé to funguje v rozhraní dostupném online na stránkách Apiary, kde si napíšeme jednoduchý dokument. Pro ukázku takový dokument bude vypadat zhruba takto:

```
## Questions Collection [/articles]

### List All Articles [GET]

Returns the list of all articles

+ Response 200 (application/json)

  [
   {
    "article": [
     {
      "title": "Article 1",
      "content": "This is first article"
     },
     {
      "title": "Article 2",
      "content": "Second article"
     },
     {
      "title": "Article 3",
      "content": "Last article"
     }
    ]
```

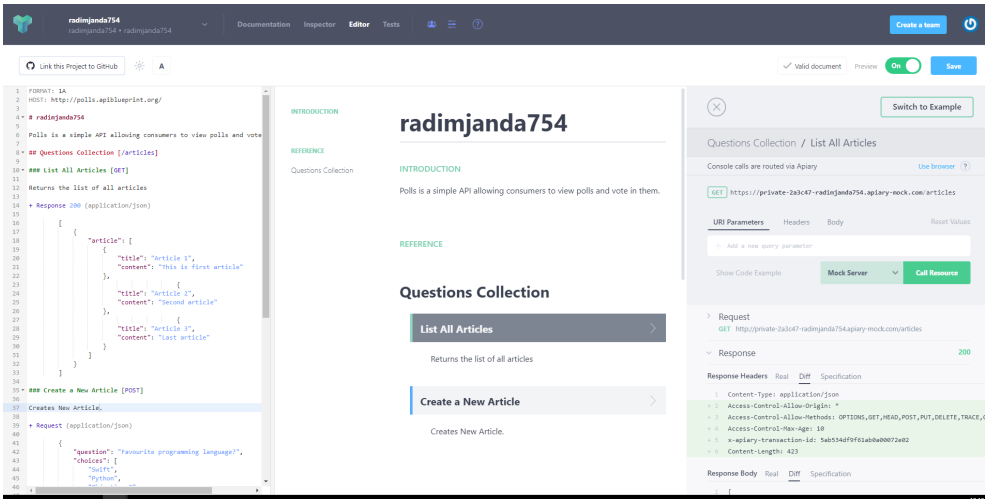
```
    }
  ]

### Post New Article [POST]

... dalsi metody pro zdroj /articles
```

Z dokumentu lze intuitivně vyčíst, že jsme si nadefinovali zdroj **/articles** a k němu přidáváme různé HTTP metody. K těmto reqeustům můžeme vytvářet popisky (tedy již dokumentaci) a nastavit, jak se mají chovat. V případě POST requestů můžeme i nastavovat různé odpovědi podle toho, jak bude vypadat požadavek. Apiary rozhraní nám poté z takového dokumentu vytvoří přehlednou dokumentaci včetně Mockup serverů, kam poté můžeme posílat požadavky i z ostatních klientů a budeme dostávat odpovědi, tak jak jsme si je v dokumentu předpřipravili.

Celé je to velmi přehledné a intuitivní, v rozhraní si pak můžeme také přímo testovat HTTP požadavky a snadno provádět změny v API. Pro podporu týmové práce je pak do tohoto rozhraní také možné sdílet přístup více vývojářům souběžně.



Obrázek 3.2: Rozhraní webové aplikacei Apiary. Zleva můžeme vidět: Zdrojový dokument tvořící dokumentaci a mockup, vygenerovanou dokumentaci, nástroj k vyzkoušení HTTP requestů na mockupované API. Zdroj: Pořízeno z webové aplikace Apiary

Podobné nástroje:

- Mockable - `https://www.mockable.io/` - Velice podobný princip, jen nemusíme pracovat s dokumentem, ale pro definování requestů máme

vytvořené rozhraní.

- Swagger - `https://www.swagger.io/` - Kromě dokumentace a tvorby mockup serverů umožňuje řadu dalších funkcí, např. testování. Na tento nástroj se ještě podíváme, až se budeme zabývat automatickým generováním dokumentace.
- Mocky - `https://www.mocky.io/` - Snadný a rychlý mockup jednoduchých GET requestů, bez nutnosti registrace. Neposkytuje však takové možnosti jako předchozí nástroje.

3.1.3 Technologie: JSON-Server

[26] Pokud chceme hostovat Mockupovaný server na vlastních zařízeních a ne na serverech třetích stran, jak to bylo u předchozích nástrojů, mohl by nás zajímat **JSON-Server**. Jak už název napovídá, jedná se o Node modul běžící na frameworku ExpressJS, díky kterému můžeme nadefinovat **JSON soubor** a ten se pak bude chovat jako datový zdroj.

Samotný JSON-Server nám tedy poběží na localhostu a automaticky vygeneruje **GET, POST, PUT, PATCH a DELETE HTTP** metody pro JSON soubory, které jsme poskytli. Jednoduchý mockup RESTful API tak lze vytvořit během několika minut.

Ukázky kódu:

- 1) Instalace s použitím **npm** (správce balíčků pro Node.js balíčky)

```
npm install -g json-server
```

- 2) Vytvoření JSON souboru, např **articles.json**

```
"articles": [
 {
  "title": "Article 1",
  "content": "This is first article"
 },
 {
  "title": "Article 2",
  "content": "Second article"
 }
]
```

- 3) Spuštění JSON Serveru s **/articles**

```
json-server articles.json
```

A je hotovo. Náš namockupovaný server běží na localhostu a podporuje veškeré výše zmíněné požadavky na **/articles**. Je zde také podpora pro propojení s databází.

3.2 Issue tracking a verzování kódu

Když už máme naše API navržené např. nějakým nástrojem zmíněným v předchozí sekci, můžeme se pustit do jeho samotné implementace. V praxi na takové aplikaci ve většině případech bude zcela jistě pracovat více vývojářů, kteří si také budou potřebovat sdílet přístup ke kódu. Týmová práce dnes funguje zpravidla na principu rozdělování drobných úkolů v týmu a k tomu se dnes zpravidla používají **Issue Tracking**[27] systémy. Ty bývají propojené s nějakým verzovacím systémem (většina čtenářů jistě bude znát **Git** nebo **SVN**), které se starají o sdílení kódu a všechny proběhlé úpravy tohoto kódu v historii. Verzovecí systém pak může být napojený na nějaký **AutoDeploy** nebo **AutoTest** systém (probereme v dalších sekcích).

V této sekci probereme pár zástupců populárních **Issue Tracking** systému. Verzovací technologie projdeme jen zrychleně, jelikož se dle mého názoru jedná o naprosto základní věc, kterou by měl každý vývojář již znát,

3.2.1 Výhody a použití

Issue Tracking systémy (také známe jako systémy řízení změn apod.) jsou klíčovou součástí v řízení projektů a celkově nutné pro správnou spolupráci týmu. Pokročilejší systémy přináší spoustu dalších výhod, mezi které se řadí např. evidence pracovní doby. Verzovací systémy jsou pak nezbytné pro správnou kontrolu nad změnami v kódu a neměly by chybět u žádného projektu, kde pracuje víc lidí.

Lepší práce v týmu Issue Tracking systémy umožňují snadno rozdělovat úkoly v týmu a kontrolovat postup práce na těchto úkolech. Verzovací systémy pak umožní snadné slučování kódu od více vývojářů.

Kontrola nad změnami v kódu Verzovací systémy poskytují vysokou kontrolu nad všemi změnami provedenými v kódu, také typicky umožňují snadno vyvíjet několik verzí aplikace najednou - např produkční REST API a vývojové REST API.

Další možnosti (AutoDeploy,..) Verzovací systémy lze snadno napojit na další nástroje zařizující AutoDeploy (tedy automatické nasazení aplikace do provozu). Issue Tracking systémy pak poskytují řadu dalších funkcí (evidence pracovní doby, posílání výplat atd, ..)

3.2.2 Technologie: Bitbucket + Git

Obrázek 3.3: Logo Bitbucket. Zdroj: `https://bitbucket.org/`

Veškeré Issue Tracking systémy jsou velmi podobné, demonstrujeme si je na systému **Bitbucket**[28], což je webová služba pro issue tracking propojená s verzovacím systémem **Git**[29]. Představuje snadné a za jistých podmínek i bezplatné online řešení pro verzování a přerozdělování úkolů (issues). Bitbucket nám nejprve umožňuje vytvořit tým z registrovaných uživatelů, ke kterému lze pak přiřazovat projekty a určité Bitbucket-repozitáře, což jsou vlastně projekty s přidruženým Git-repozitářem. V takovém týmu lze pak nastavovat členům různá práva na daný repozitář a samotný repozitář

Rozhraní služby Bitbucket, nám tedy v první řadě umožňuje vytvářet úkoly (**issues**), tyto úkoly lze různě přiřazovat členům týmu a nastavovat jim následující vlastnosti, podobně jako u většiny dalších Issue Tracking systémů:

Type - O jaký typ problému se jedná, může se jednat například o opravu bugu, nebo o vylepšení. Bitbucket poskytuje následující:

- Bug
- Enchancement
- Proposal
- Task

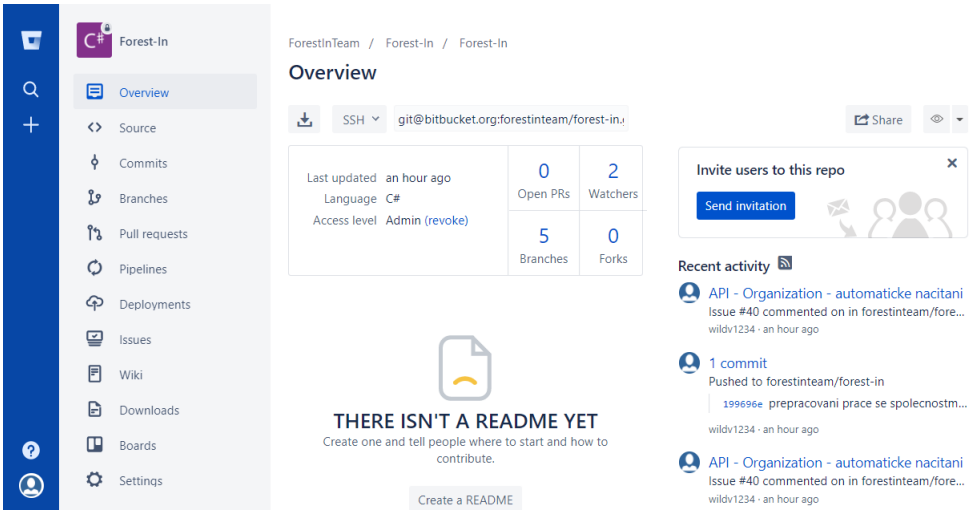
Priority Priorita problému, jak nezbytné je začít se problémem co nejdříve zabývat. Může být:

- Trivial
- Minor
- Major
- Critical
- Blocker

Status Představuje současný stav práce na konkrétním problému, jestli se na něm aktivně pracuje nebo jestli už je vyřešen a podobně. Na Bitbucketu:

- Open
- On hold
- Resolved
- Duplicate
- Invalid
- Wontfix
- Closed

Dále nám rozhraní Bitbucket poskytuje snadné vytvoření **Wiki** pro projekt a celkový přehled nad Git-repozitářem.



Obrázek 3.4: Rozhraní webové aplikacei Bitbucket. Zdroj: Pořízeno z webové aplikace Bitbucket

Samotný **Git** zcela jistě většina čtenářů této práce zná, jedná se o nepopulárnější verzovací systém. Je v něm možné vytvářet **branche** (různé verze projektu), které se skládají z **commitů** (nějaký konkrétní bod, ve kterém jsou zaznamenány změny všech souborů od předchozího commitu). Každý vývojář pak má vlastní lokální kopii Git repozitáře a vytváří vlastní commity, které se pak sloučí při odeslání na sdílený Git repozitář. Co se týče branchů, nejčastější přístup je pro každou změnu vytvořit vlastní branch a při dokončení změny se tento branch merguje (slučuje) s hlavním branchem (**master**). Na master

branch pak může být vázaný **AutoDeploy**, nebo **AutoTest**. V případě nějakých problémů se pak lze snadno vrátit do předchozího commitu a obnovit tak staré soubory.

Další issue tracking systémy:

- Gitlab - `https://about.gitlab.com/` - propojené s Gitem, podobně jako Bitbucket.
- GitHub - `https://github.com/` - propojené s Gitem, podobně jako Bitbucket.
- OpenProject - `https://www.openproject.org/` - není propojený s Gitem, umožňuje však mnoho dalších možností, jako je např. vykazování odpracovaných hodin. Používá se často v menších firmách.

Další verzovací systémy - alternativy pro Git:

- Mercurial - `https://www.mercurial-scm.org/`
- Apache SVN - `https://subversion.apache.org/`

3.3 Cloudové řešení - Hostování, nasazování a další služby

Pokud již máme naimplementovanou alespoň část REST API, zcela jistě bude nutné tuto aplikaci někde provozovat. Můžeme použít vlastní server, či si nějaký pronajmout, avšak dnes se stále častěji používá řešení cloudových služeb, kdy si od poskytovatele těchto služeb pronajímáme část výkonu jeho serveru a na té provozujeme naši aplikaci. V této sekci se oproti běžnému hostování na dedikovaných serverech podíváme na nějaké cloudové služby pro snadné a spolehlivé provozování RESTful API aplikací. Kromě toho také shrneme některé další služby, které nám přinášejí cloudová řešení, jako je automatické nasazování nebo testování.

3.3.1 Výhody a použití

Cloudové servery oproti dedikovaným serverům poskytují následující výhody:

Hardware Není nutné starat se o hardware, vše zařídí poskytovatel cloudových služeb.

Škálování Obrovské možnosti škálování. Při rozšíření našich serverů si stačí jen zakoupit více výkonu.

Placení Není nutné platit za nějaké výkonné servery, které ani nevyužijeme. Zakoupíme si přesně takový výkon, jaký pro naší aplikaci potřebujeme.

Další výhody cloudových řešení Cloudové řešení je široký pojem, který poskytuje nespočetně dalších služeb, které lze u aplikace provozované na cloudu snadno využít.

3.3.2 Technologie: Microsoft Azure

Obrázek 3.5: Logo Azure. Zdroj: `https://blogs.technet.microsoft.com/luishernandez/tag/microsoft-azure-certifications/`

[13] **Azure** je komplexní sada cloudových služeb, které vývojáři a odborníci na IT využívají k sestavování, nasazování a správě aplikací prostřednictvím globální sítě datových center.

Mezi nejčastěji používané funkce samozřejmě patří hostování aplikací. Azure poskytuje různé mohutnější virtální servery, ale pro REST API aplikaci je jednodušší zakoupit si přímo slot pro webovou aplikaci (**Web App**) a případné další doplňky (**SQL server, ...**). Aplikaci publikujeme do našeho Web App slotu, který si ji sám zkompiluje a pak provozuje na cloudu pod veřejně dostupnou doménou. K takovému Web App slotu se váže obrovské množství funkcí a nastavení, mezi které patří např. **automatické nasazování - AutoDeploy** nebo **automatické testování - AutoTest** při commitu nového kódu do určitého branche, k čemuž slouží funkce **Deployment Options**. To znamená, že veškeré nové verze, které chceme nasadit do provozu, stačí pouze pushnout na daný Git-repozitář do nějakého master branche a Azure si kód z tohoto branche automaticky stáhne, zkompiluje, nasadí a případně otestuje.

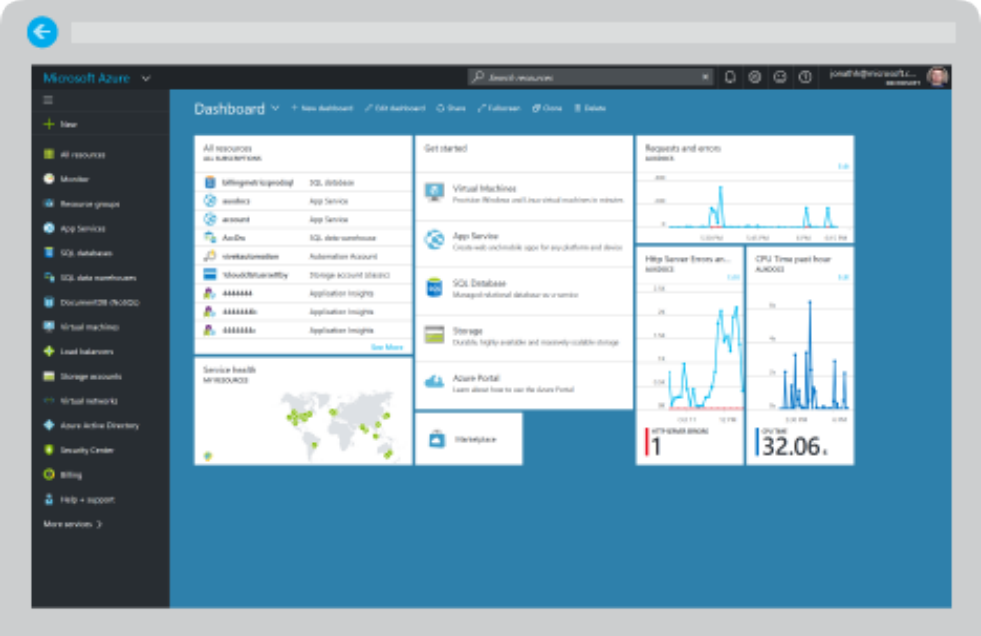
Rozhraní **Azure** je uživatelsky přívětivé, ale vzhledem k obrovskému množství funkcionalit, které poskytuje, trochu nepřehledné. Obdobně je to ale i u konkurenčních poskytovatelů cloudových služeb.

K dalším zajímavým službám, které bychom například mohli pro REST API aplikaci využít, můžeme řadit např. **Notifications Hub**[30], což je služba k odesílání push notifikací (různých dat) na mobilní zařízení, včetně nejpopulárnějších systémů **Android** a **iOS**. V REST API aplikaci bychom toto mohli použít např. při informování klientů o úpravě nějakých dat (zdrojů).

Jednou z hlavních výhod je zabezpečení. [31] Azure splňuje různé mezinárodní a oborové standardy pro dodržování předpisů, jako jsou **ISO 27001,**

HIPAA, FedRAMP, SOC 1 a SOC 2, stejně tak jako standardy specifické pro konkrétní země, třeba IRAP (Austrálie), G-Cloud (Velká Británie) nebo MTCS (Singapur). Přísné audity třetích stran, např. organizace BSI (British Standards Institute), ověřují, že Azure splňuje striktní bezpečnostní opatření vyžadovaná těmito standardy. Azure poskytuje svým uživatelům službu **Azure Security Center**, což je jednotná správa zabezpečení a pokročilá ochrana před hrozbami napříč hybridními cloudovými úlohami. Spadá sem:

- Monitorování zabezpečení místních i cloudových úloh
- Blokování škodlivých aktivit pomocí ovládacích prvků přístupu a aplikací
- Detekce útoků s využitím pokročilých analýz a analýz hrozeb
- Nalezení a opravení ohrožení zabezpečení dříve, než se dají zneužít
- atd..



Obrázek 3.6: Webové rozhraní pro správu cloudových služeb Microsoft Azure. Zdroj: `https://azure.microsoft.com/en-us/account/`

Další populární **cloudové služby**:

- Amazon Web Services - `https://aws.amazon.com/`

- Google Cloud Platform - `https://cloud.google.com/`

Nasazování aplikací lze také řešit pomocí **systémů průběžné integrace**, které kód automaticky zkompilují, otestují a případně odešlou k nasazení. Mezi populární zástupce patří:

- Jenkins - `https://jenkins.io/`
- Team City - `https://www.jetbrains.com/teamcity/`

3.4 Tvorba automatické dokumentace

V první sekci jsme již probírali dokumentaci v rámci namockupovaných metod. Taková dokumentace může být velice užitečná při počátečních pracích na projektu, avšak nikdy nebude tak přesná jako dokumentace generovaná přímo z kódu, která se za průběhu vývoje může různě měnit. V této sekci shrneme některé nástroje pro automatickou generaci dokumentace z kódu

3.4.1 Výhody a použití

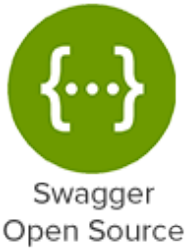
Automaticky generovaná dokumentace poskytuje následující výhody:

Snadné změny v projektu Snadné vygenerování nové dokumentace při různých změnách v kódu.

Formát dokumentace Většina projektů používá automaticky vygenerovanou dokumentaci, tudíž lide jsou zvyklí s touto dokumentací pracovat.

Automatické generování modelů Některé nástroje si pomocí automaticky generované dokumentace dokážou například vygenerovat identické modely na straně klientských aplikací.

3.4.2 Technologie: Swagger



Obrázek 3.7: Logo Swagger. Zdroj: `https://swagger.io/download-swagger-ui/`

[32] **Swagger** je open-source framework pro podporu tvorby RESTful API od návrhu, přes dokumentaci až po testování samotných metod API. Co se týče automatické tvorby dokumentace, tak ho lze stáhnout jako dodatečný balíček pro většinu technologiích shrnutých v předchozí aplikaci. Dokumentace se generuje z komentářů přidružených k metodám (typicky se jedná o XML formát). Pro náš případ si tento framework demonstrujeme na technologii **ASP.NET Core**. Nejprve se podíváme, jak bychom okomentovali nějakou třídu z vrstvy **model**:

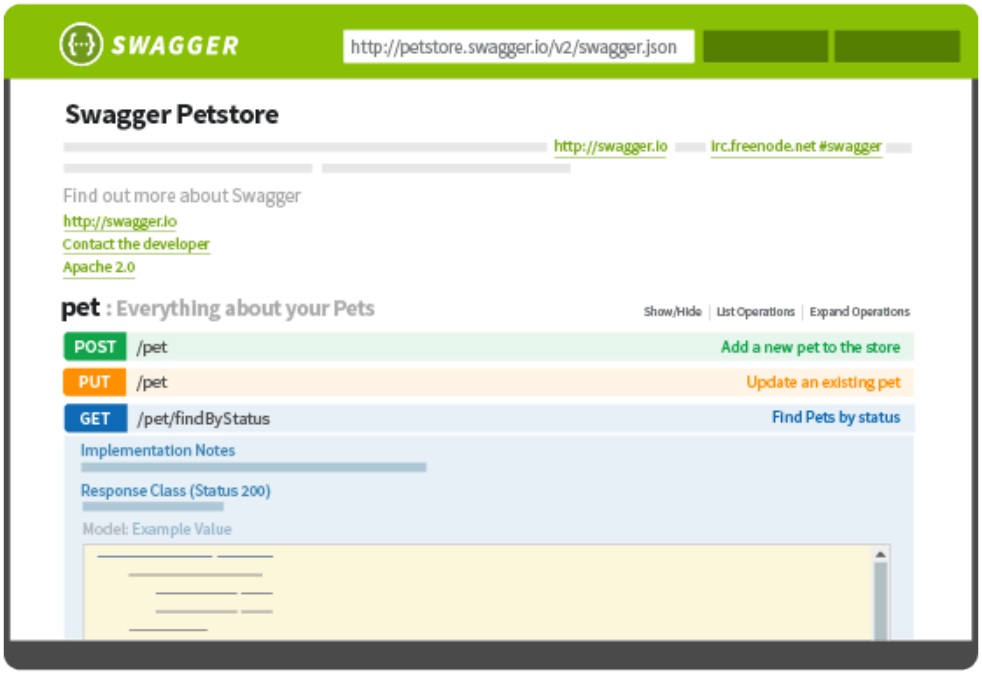
```
public class BasicResource
{
 public int Id { get; set; }
 /// <summary>
 /// Id of user that created this resource
 /// </summary>
 [Required]
 public string CreatedBy { get; set; }
 /// <summary>
 /// Date when resource was created.
 /// </summary>
 public DateTime CreatedTime { get; set; }
}
```

Máme nějaký základní zdroj s atributy, jako např. kdo a kdy tento zdroj vytvořil. Vidíme zde nějaké XML komentáře, které tyto atributy popisují. Nyní se do **controlleru** k tomuto modelu a na jednu jeho konkrétní metodu podíváme:

```
/// <summary> Returns Resource data for ID </summary>
/// <returns> User data in JSON </returns>
/// <response code="200">Returns Resource</response>
/// <response code="404">Resource with following id does not
/// exists </response>
// GET api/[controller]/5
[HttpGet("{id}")]
[ProducesResponseType(typeof(BasicResource), 200)]
public IActionResult Get(int id)
{
 ...
}
```

Swagger pro ASP.NET Core je tak propracovaný, že i bez výše zmíněných tagů dokáže vytvořit zdánlivě kvalitní dokumentaci. Výsledná dokumentace vypadá tak, že se jedná o webovou stránku, kde jsou vypsané všechny metody daného RESTful API. Ty jsou zdokumentované podle dat, která vrací, dále podle struktury modelu a také orávě podle těchto XML komentářů. Tato on-

line dokumentace je velice interaktivní a umožňuje i snadno vyzkoušet dané metody posíláním HTTP requestů ze Swagger rozhraní, které přehledně připraví formáty modelů.



Obrázek 3.8: Online rozhraní dokumentace, vygenerované frameworkem Swagger. Zdroj: `http://swagger.io/`

Mezi další populární nástroje pro tvorbu dokumentace patří:

- Mashery - `mashery.com/`
- Apiary - `https://apiary.io/` - viz sekce Mockup pro REST API

3.5 Shrnutí

V kapitole jsme stanovili odvětví podpůrných nástrojů používaných při tvorbě RESTful API, následně tyto nástroje shrnuli včetně rozebrání jejich výhod. Pro čtenáře tedy poslouží jako přehled podpůrných nástrojů, které by mohl využít, kdyby sám měl v plánu vyvíjet podobnou webovou aplikaci a usnadní mu tak jejich výběr. Na tyto nástroje následně navážeme v další kapitole, kde se budeme zabývat vytvořením stacku tachnologií a nástrojů pro nejvyšší míru podpory tvorby RESTful API.

Následující tabulka shrnuje probrané oblasti pro podporu tvorby RESTful API a nástroje, které je pro takovou oblast možno zvolit:

	Zajímavé technologie a nástroje	Přínosy
O1	**Apiary**, **JSONServer**, Swagger, Mockable, Mocky	Předběžná dokumentace, testování
O2	**Bitbucket**, Gitlab, Github, OpenProject. Verzování: **Git**, Mercurial, SVN	Práce v týmu, kontrola nad kódem
O3	**Azure**, Google Cloud, Amazon WS	Škálování, testování, nasazování aplikace
O4	**Swagger**, Mashery, Apiary	Snadná kvalitní průběžná dokumentace

- **O1** - Mockup pro RESTful API
- **O2** - Issua tracking a verzování kódu
- **O3** - Cloudové řešení - hostování, nasazování a další služby
- **O4** - Tvorba automatické dokumentace

Tabulka 3.1: Tato tabulka shrnuje zajímavé technologie a nástroje pro podporu tvorby RESTful API aplikací vzhledem k oblasti výhod, které přináší. Tučně zvýrazněné pojmy jsou podrobněji probrány v této kapitole.

Kapitola 4

Stanovení sestavy technologií a nástrojů pro podporu tvorby RESTful API

Tato kapitola se zabývá **navrhnutím sestavy technologií a nástrojů (stacku)**, které zajistí nejvyšší míru podpory při tvorbě RESTful API. Sestava bude tvořena z teoretického hlediska a to podle toho, jak by mohlo být nejvýhodnější při tvorbě API postupovat a co použít. Její skutečnou efektivitu ověříme v následující praktické části práce, ve které tuto sestavu použijeme. Sestava bude nejdříve stanovena obecně a následně pro ni vybereme konkrétní technologie.

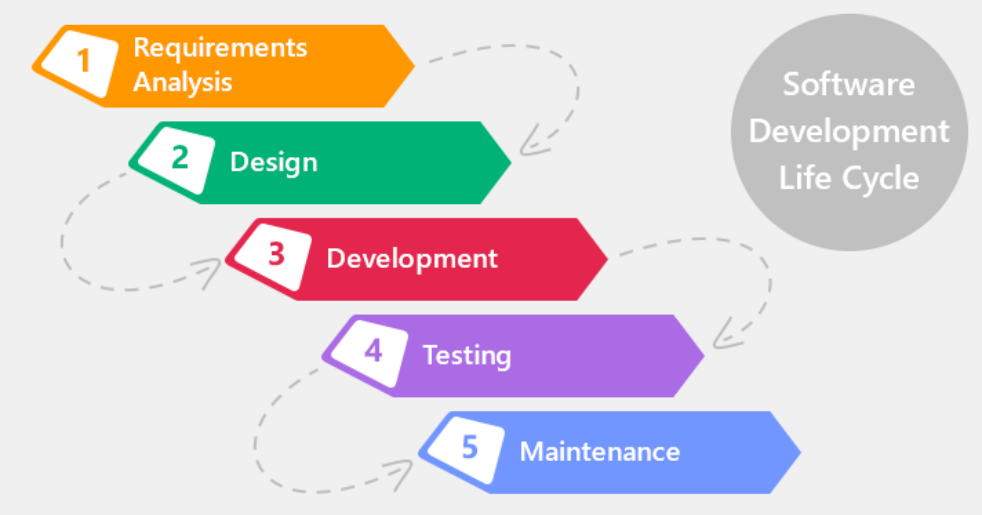
Výsledkem kapitoly tedy bude navržený seznam konkrétních technologií, které použijeme v další kapitole při praktické části. Praktickou částí ověříme, jak je tato navržená sestava užitečná v praxi, proto se bude jednat o implementaci RESTful API na opravdovém projektu, které poté bude reálně fungovat. Další kapitola tedy bude obsahovat zhodnocení a případné úpravy této sestavy.

4.1 Stanovení sestavy obecně

Sestava se bude skládat z frameworku pro tvorbu RESTful API aplikace, což je zmíněno v **2. kapitole**, a z dalších podpůrných nástrojů probraných v **3. kapitole**. V předchozí kapitole, ve které jsme probírali podpůrné nástroje / technologie jsme již postupovali podle klasického **životního cyklu vývoje software** a stanovovali oblasti, ve kterých by nám tyto nástroje mohly být nápomocny. Výsledná sestava pro podporu bude tedy logicky vycházet z těchto oblastí, z čehož usuzujeme, že **většina diskuzí ohledně stanovení této sestavy již proběhla v předchozích kapitolách**.

Celý cyklus vývoje software zde však v rychlosti připomeneme a namapujeme na něj podpůrné nástroje, které budou v této fázi užitečné. Existuje mo-

noho různých metodik vývoje software, avšak pro základní představu ohledně stanovení této sestavy postačí, když se podíváme na základní **vodopádový model**[43].



Obrázek 4.1: Obrázek znázorňuje životní cyklus vývoje software - jednoduchý vodopádový model. Zdroj: `https://xbsoftware.com/blog/software-development-life-cycle-waterfall-model/`

4.1.1 Analýza a návrh

Prvotní fáze **analýzy požadavků** se zabývá především předběžnou dokumentací ohledně toho, co by měl software umět a komunikací s případným klientem. Pro tuto fázi mohou jako výstup postačit i jednoduché dokumenty stanovující různé funkční požadavky, důležité je však pochopení toho, co se od software očekává.

Ve fázi **návrhu** se pak můžeme setkávat s různými papírovými návrhy tohoto software a případně různými architektonickými návrhy. Jak jsme si ale již říkali, tak architektura REST a frameworky stanovují většinu návrhu za nás. Pokud se nejedná o rozsáhle komplikovanou aplikaci, na které by pracoval široký tým vývojářů, tak různé architektonické nástroje typu Enterprise architect postrádají význam. V RESTful API aplikaci rovněž příliš nevyužijeme nástroje pro návrh grafické podoby aplikace, jelikož se jedná pouze o textové rozhraní přes HTTP protokol.

V obou těchto fázích nám budou velmi užitečné právě **nástroje pro Mockup**, které poskytují předběžnou dokumentaci a jednoduchý návrh RESTful API poskytující rovněž základní funkčnost, který lze využít i pro předběžné testování. Do výsledné sestavy tedy přidáme **nástroje pro Mockup**:

1.	Nástroj pro Mockup RESTful API aplikace

4.1.2 Vývoj

Fáze se zabývá samotnou implementací RESTful API aplikace. Zde logicky v první řadě využijeme právě nějaký **framework** pro vývoj takové aplikace, probraný v **2. kapitole**. Taková technologie je však absolutním základem celého vývoje a můžou se od ní odvíjet další nástroje, proto ji v našem stacku umístíme na začátek.

1.	Framework pro vývoj RESTful API aplikace
2.	Nástroj pro Mockup RESTful API aplikace

Při vývoji se však typicky pracuje v týmech, proto je vhodné práci v týmu koordinovat nějakým systémem pro rozdělování úkolů, kde se rovněž reaguje na změny požadavků a řešení bugů. K tomu můžeme použít již zmíněný **Issue Tracking Systém**

1.	Framework pro vývoj RESTful API aplikace
2.	Nástroj pro Mockup RESTful API aplikace
3.	Issue Tracking Systém

Rovněž se musíme zabývat kontrolou nad změnami v kódu, aby při práci v týmu nedocházelo k jeho nežádoucím deformacím. K tomuto účelu se hojně využívají také již zmiňované **verzovací systémy**.

1.	Framework pro vývoj RESTful API aplikace
2.	Nástroj pro Mockup RESTful API aplikace
3.	Issue Tracking Systém
4.	Verzovací sytém

4.1.3 Testování, nasazování a údržba

V konečných fázích **testování a údržby** se zabýváme hlavně dodáváním kvalitního a funkčního softwaru. Zde se nabízí široké spektrum možností pro to, jak aplikaci automaticky testovat, nasazovat a udržovat. V této práci bylo vybráno populární řešení, které obsahuje vše v jednom a to je **cloudové řešení**. Na populárních cloudech lze nastavit automatické testy, nasazení a kontrola provozu naší RESTful API aplikace, jak již bylo probráno v předchozí kapitole.

1.	Framework pro vývoj RESTful API aplikace
2.	Nástroj pro Mockup RESTful API aplikace
3.	Issue Tracking Systém
4.	Verzovací sytém
5.	Cloudová platforma

Vhod nám při údržbě přijde také **automaticky generovaná dokumentace**, která se bude udržovat aktuální i při změnách v kódu. Po přidání nástroje pro tuto dokumentaci již dostaneme výslednou sestavu zajišťující vysokou míru podpory tvorby RESTful API aplikací.

1.	Framework pro vývoj RESTful API aplikace
2.	Nástroj pro Mockup RESTful API aplikace
3.	Issue Tracking Systém
4.	Verzovací sytém
5.	Cloudová platforma
6.	Nástroj pro automatickou tvorbu dokumentace

Tabulka 4.1: Výsledná tabulka znázorňující obecnou sestavu technologií a nástrojů pro vysokou míru podpory tvorby RESTful API

4.1.4 Efektivita sestavy vůči alternativám

Jelikož existuje nespočetné množství podobných alternativních sestav nástrojů, které by bylo možné při tvorbě RESTful API použít, zaměříme se pouze na zhodnocení a maximalizaci efektivity sestavy vytvořené v této práci. Její skutečnou efektivitu tedy ověříme v praktické části práce, kde sestavu použijeme.

Avšak jako možnou alternativu pro tuto sestavu lze také brát použití pouze jednoho podpůrného nástroje, který poskytuje podporu po celou dobu životního cyklu vývoje software. Mezi takové nástroje lze řadit i **Swagger**, který byl probrán v předchozí kapitole. Ten mimo jiné poskytuje možnosti návrhu, generování a testování RESTful API. Vytvořená sestava oproti takovému řešení přináší vysokou variabilitu, kdy v každé fázi vývoje můžeme použít jiný nástroj, který pro tuto fázi bude nejvíce vyhovovat. Sestava také navíc řeší prvky v rámci týmové komunikace a obsahuje moderní prvky typu Cloudových řešení, které ve Swaggeru nenalezneme.

4.2 Vybrání konkrétních technologií pro sestavu

V předchozí sekci jsme stanovili obecnou sestavu technologií a nástrojů pro vysokou míru podpory tvorby RESTful API, v této sekci pro ní vybereme konkrétní nástroje. Z předchozích kapitol také víme, že výběr technologií nemá vždy jen jednu konkrétní cestu a často je řešení vybíráno podle požadavků klienta.

Tato konkrétní sestava je vybrána pro demonstraci praktické tvorby RESTful API na reálném projektu, ke kterému se dostaneme v další kapitole. V tomto projektu byl stanoven požadavek použít jako cloudovou platformu **Microsoft Azure**. To nám i usnadňuje výběr frameworku, jelikož je jediný z probraných frameworků, který má pro tuto cloudovou platformu vysokou podporu a tím je **ASP.NET Core**. Jako nástroj pro Mockup bylo zvoleno probrané **Apiary**, které umožňuje snadnou a rychlou dokumentaci API v začátcích práce. V případě Issue Tracking Systémů a verzovacích systémů se rovněž budeme držet probraných nástrojů a použijeme **Bitbucket + Git**. Na závěr nám zbývá volba nástroje pro tvorbu automatické dokumentace, zde použijeme probraný **Swagger**, který má dobrou podporu pro vybraný framework.

Konečný seznam vybraných nástrojů tedy bude:

- **ASP.NET Core**, jako framework pro vývoj
- **Apiary**, jako nástroj pro Mockup
- **Bitbucket**, jako Issue Tracking Systém
- **Git**, jako verzovací systém
- **Microsoft Azure**, jako cloudová platforma
- **Swagger**, jako nástroj pro generování dokumentace

4.3 Shrnutí

V této kapitole jsme stanovili obecnou i konkrétní sestavu technologií a nástrojů pro co nejvyšší míru podpory tvorby RESTful API aplikací. Efektivita konkrétní sestavy pak bude ověřena v následující praktické kapitole v projektu implementující reálné RESTful API. Některé technologie do konkrétní sestavy byly také vybírány podle požadavků tohoto projektu.

Následující tabulka shrnuje stanovenou obecnou i konkrétní sestavu:

Typ technologie (obecná sestava)	Konkrétní vybraná technologie
Fremework pro vývoj RESTful API aplikace	ASP.NET Core
Nástroj pro Mockup RESTful API aplikace	Apiary
Issue Tracking Systém	Bitbucket
Verzovací sytém	Git
Cloudová platforma	Microsoft Azure
Nástroj pro automatickou tvorbu dokumentace	Swagger

Tabulka 4.2: Výsledná tabulka znázorňující sestavu technologií a nástrojů pro vysokou míru podpory tvorby RESTful API, včetně konkrétních vybraných technologií, které budou použity v další kapitole.

Kapitola 5

Implementace RESTful API

Tato kapitola shrnuje návrh a implementací vybraného RESTful API. V této kapitole byla použita sestava technologií stanovená v předchozí kapitole. Mezi hlavní cíle této části patří ověřit užitečnost zmiňované sestavy v praxi a případně ji optimalizovat. Na konci kapitoly tedy rozebereme, které z vybraných nástrojů tvorbu RESTful API nejvíce podporují a které z nich v praxi skutečně použijeme. Dalším výsledkem této kapitoly bude pak implementované rozsáhlé RESTful API, které bude reálně fungovat u příslušného projektu.

5.1 Definice projektu a stanovení požadavků

Projekt byl vybírán tak, aby se co nejvíce přiblížil praxi, tedy pracím běžného vývojáře RESTových API. Jedná se o systém pro odborníky pracující v oblasti služeb pro ohrožené rodiny a děti v regionu jižních Čech. Projekt byl ve stavu naprostého počátku, tudíž bylo v plánu paralelně vyvíjet klientskou webovou aplikaci, RESTful API aplikaci a později klientskou mobilní aplikaci. V této práci se podíváme právě na implementaci RESTful API aplikace pro tento projekt, kterou budu vyvíjet sám. Ostatní členové týmu pak dostali na starost zbylé klientské aplikace.

Požadavky klienta byly původně stanoveny spíše na celý systém, než na konkrétní části. Z těchto požadavků se pak po konzultaci s členy týmu daly odvodit hlavní požadavky podstatné pro RESTful API:

1. **RESTful API** Rozhraní bude poskytovat synchronizovaná data jak pro webovou aplikaci, tak pro budoucí mobilní aplikaci.

2. **Uživatelské účty** V systému bude klasické přihlašování s registrací. Ke každému uživateli bude možno uchovávat různá data typu: jméno, příjmení, fotografie, informace o profesi, pozici, organizaci, minulosti, zaměření, specializaci.

3. **Zabezpečení** RESTful API bude zabezpečené například pomocí bezpečnostního tokenu generovaného uživatelům. Podle tokenu bude možné identifikovat uživatele a jeho oprávnění v systému.

4. **Microsoft Azure** RESTful API bude běžet na cloudové platformě Microsoft Azure, je vyvíjeno pro neziskovou organizaci, která pro tuto platformu dostala grant.

5. **Chat + zprávy** V systému bude možný chat mezi uživateli a zasílání systémových zpráv.

6. **Pracovní skupiny** V systému bude možné vytvářet pracovní skupiny uživatelů, kteří si mezi sebou budou sdílet různé soubory a dokumenty.

7. **ARES** API bude umět zpracovávat data ze systému **ARES**[45], což je informační systém, který umožňuje vyhledávání mezi ekonomickými subjekty registrovanými v České republice. Tato data budou použita při vyhledávání základních informací organizací.

Několik dalších požadavků na API však bylo zjištěno a přidáno až ve fázi implementace. Takové požadavky vznikaly hlavně ze strany klientských aplikací. Jak již bylo zmiňováno, při veškerých pracích na tomto projektu bude použita **sestava technologií stanovená v předchozí kapitole**, proto ke zpracování těchto dodatečných požadavků byl použit Issue Tracking Systém **Bitbucket**.

5.2 Návrh

Návrh RESTful API byl prováděn hlavně pomocí předem stanovené webové aplikace **Apiary**, ta poskytla snadnou předběžnou dokumentaci včetně Mockupu, na kterém se také mohly metody pro lepší pochopení předběžně odzkoušet. V průběhu vývoje však docházelo k častým drobným úpravám tohoto návrhu a od **Apiary** se kompletně upustilo. Předběžnou dokumentaci nahradil **Swagger**, který poskytoval aktuální automaticky generovanou dokumentaci, která i v případě odzkoušení metod pracovala se skutečnými daty. Swagger se jakožto nástroj pro dokumentaci ukázal být přehlednější a pochopitelnější než Apiary, které se využilo pouze v prvních fázích prací na projektu.

Další zajímavou věcí při návrhu bylo oddělit část s přihlašováním od druhé samostatně funkční aplikace založené na frameworku **IdentityServer4**[44] pro **ASP.NET Core**, který přinesl snadné zabezpečení k přístupu do API pomocí **OpenID-Connect** protokolu probraného v první kapitole. To zajistilo přihlašovací bránu dostupnou jak z webové, tak z mobilní aplikace, která těmto aplikacím poskytla token identifikující uživatele včetně jeho oprávnění.

Samotné RESTful API pak v konečném návrhu bylo z požadavků stanovené následovně, zdroje až na výjimky obsahují metody:

- GET /resources - získání všech zdrojů daného typu
- POST /resources - vytvoření nového zdroje
- DELETE /resources/id - smazání konkrétního zdroje podle jeho ID
- GET /resources/id - získání konkrétního zdroje podle ID
- PATCH /resources/id - upravení konkrétního zdrohe podle ID, attributy které nebyly v novém zdroji zaslány se nemění

Zdroje:

/users Zdroj obsahující základní informace k uživateli (jméno, příjmení, apod.., vázán na databázi uživatelů na identity serveru. Také se k němu váže několik dalších zdrojů)

/specializations Zdroj obsahující specializaci vázanou na konkrétního uživatele.

/roles Zdroj obsahující roli vázanou na konkrétního uživatele.

/images Zdroj obsahující profilovou fotku. Její ID je přiřazeno konkrétnímu uživateli.

/positioninorganizations Zdroj obsahující pozici uživatele v nějaké organizaci. Váže se na konkrétního uživatele a organizaci

/organizations Zdroj obsahující podrobné informace o konkrétní organizaci. Adresa této organizace je pak rozdělena do dalších zdrojů, které se k této organizaci vážou.

/professions Zdroj vázaný na pozici v organizaci, obsahuje konkrétní informace o dané profesi.

/addresss Zdroj obsahující adresu příslušící k jedné nebo více organizacím.

/countrys /citys /regions Zdroje upřesňující stát, město a region ke konkrétní adrese. Byly použity proto, aby nedocházelo k duplicitám, kdy je například jedno město zaznamenáno u adres v různých formátech (např "Č. Budějovice"a "České Budějovice").

/workgroups Zdroj obsahující pracovní skupiny. Váže se k němu jeden vedoucí uživatel, několik příslušících uživatelů a soubory.

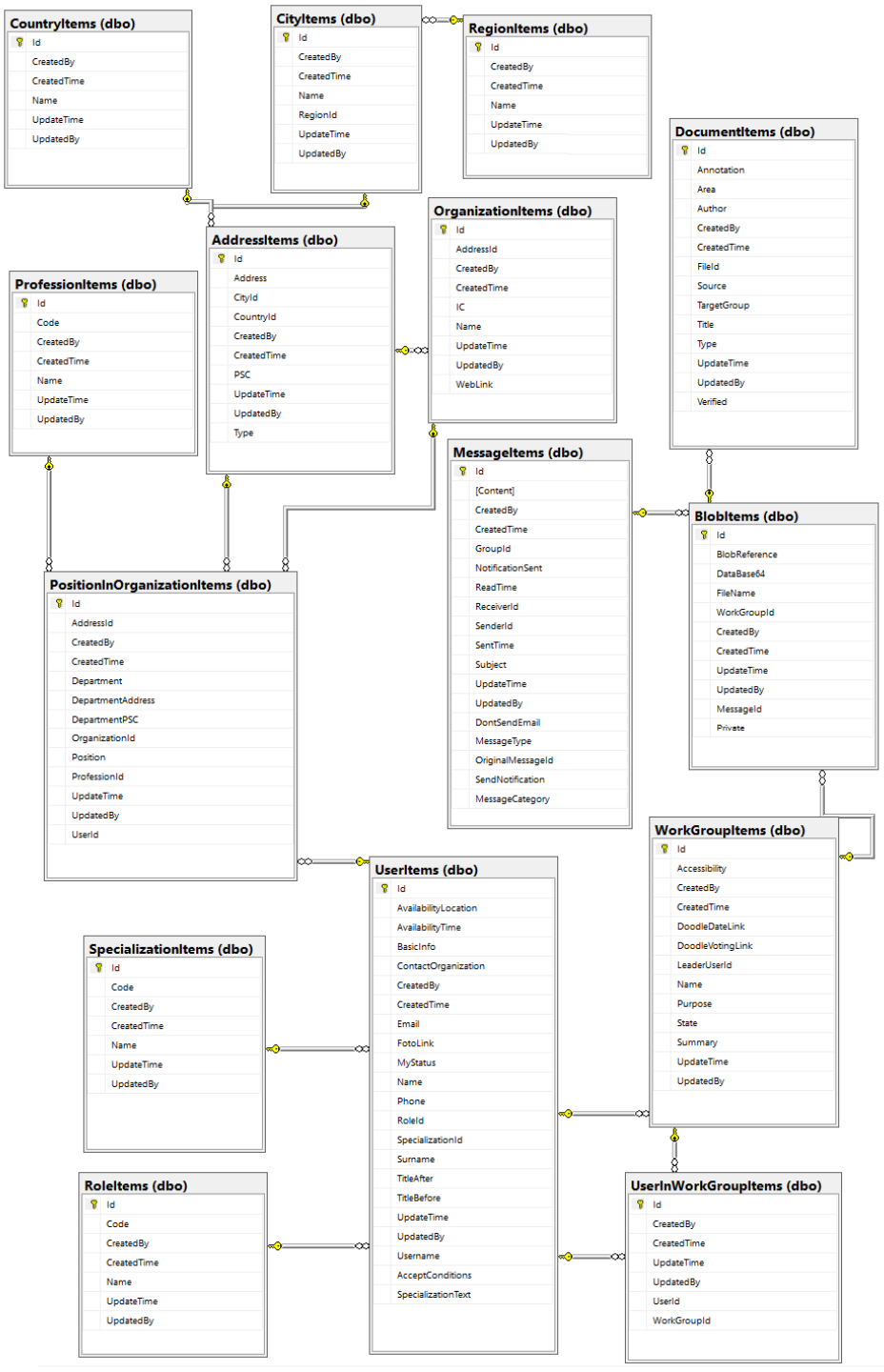
/userinworkgroup Zdroj pro zaznamenání konkrétních uživatelů v pracovní skupině

/files Zdroj pro nahrávání souborů.

/documents Zdroj, který se váže na konkrétní soubor, obsahuje specifikaci tohoto souboru jakožto dokumentu.

/messages Zdroj obsahující zprávy mezi uživateli, včetně systémových zpráv.

Zdroje obsahují filtrování pomocí url parametrů, pro zjednodušení práce v klientských aplikacích. Všechny zdroje také budou obsahovat informace o tom, kdy byly vytvořeny a kdy / kdo je upravil. Jejich provázanost si také můžeme lépe představit i na následujícím **diagramu databáze** (obrázek je ve vyšší kvalitě v přiloženém CD):



Obrázek 5.1: Diagram znázorňuje podobu databáze vyvíjeného RESTful API

5.3 Implementace

Aplikace byla implementovaná podle konečného návrhu stanoveného v předchozí sekci. V této sekci si shrneme, jak implementace probíhala a zajímavosti, na které se narazilo.

Již víme, že k implementaci samotné RESTful API aplikace byl vybrán framework **ASP.NET Core**, podrobně probraný v druhé kapitole. Aplikace byla naprogramovaná klasicky podle **Model-View-Controller** vzoru, tudíž každý zdroj má vlastní model definující jeho strukturu a controller obstarávající logiku společně se zpracováním HTTP požadavků. Všechny modely a controllery pak rozšiřují základní třídy uchovávající sdílené attributy a metody, které se často opakovaly - (třídy **BasicDBModel** a **BasicControllery**). Značné zjednodušení při implementaci přináší napojení na databázi vygenerovanou pomocí frameworku **EntityFramework**, který umožnil snadné a rychlé propojení všech zdrojů s touto perzistentní databází.

Při implementaci se často opakovaly identické postupy pro vytvoření jednotlivých zdrojů, některé controllery jsou ve výsledku skoro totožné. Tomu by pro příště mohly pomoci generátory kódu, které by automatizovaly neustále opakované postupy vytváření kódu. ASP.NET Core sice již nějaké generované šablony obsahoval, ale ty se neukázaly být pro tento projekt vyhovující.

Kód je dokumentován tak, aby z něj bylo možné správně vygenerovat dokumentaci pomocí nástroje **Swagger**. Díky napojení klientských aplikací na tuto automaticky generovanou dokumentaci bylo možné automaticky generovat modely na straně klientských aplikací. Tato dokumentace byla však z bezpečnostních důvodů zabezpečena a je k ní možno přistoupit pouze po zadání hesla. Pro ukázku kódu a jeho dokumentace se můžeme podívat například do třídy **UserModel**, což je intuitivně model pro zdroj uživatel:

```
public class UserModel : BasicDbModel
{
 /// <summary>
 /// Unique ID of the user.
 /// </summary>
 [Key]
 public string Id { get; set; }
 /// <summary>
 /// Unique Username, used for login actions.
 /// </summary>
 [StringLength(100)]
 public string Username { get; set; }
 /// <summary>
 ...
}
```

Po celou dobu vývoje byl používán Issue Tracking Systém **Bitbucket** ke zpracování dodatečných požadavků vývojářů klientských aplikací a k evidenci zjištěných chyb či oprav. S tímto systémem je, jak už víme, přímo propojený **Git**, který zajistil dobrou správu nad všemi změnami provedenými v kódu.

Microsoft Azure byl použit k hostování všech funkčních částí aplikace, tedy samotného API, k hostování identity serveru pro přihlašování a k hostování databází. Rovněž nám poskytoval možnost snadného a rychlého nasazení aplikace. **Master** branch v Gitu napojena na Azure, který při každém commitu do tohoto master branche automaticky stáhl nové zdrojové kódy a aplikaci nasadil do produkce. Testování bylo prováděno převážně manuálně.

Mezi zajímavosti implementovaných funkcí patří např. napojení zdroje **organizations** na systém **ARES**[45], který poskytuje data pouze ve formátu XML, tudíž byla použita knihovna **XDocument**[46] pro zjednodušené parsování XML souborů. Dále vznikl požadavek vývojářů klientských aplikací určovat rozměry uložených profilových obrázků (tedy zdroje **images**) podle parametru u příslušného GET požadavku. K tomuto účelu byla využita knihovna **ImageSharp**[47]. Také bylo potřeba nahrávat a stahovat soubory, k čemuž bylo využito **Azure BLOB Storage**.

Výsledné implementované API je tedy aktuálně plně funkční a provozované na cloudových serverech od Azure, navíc bylo obohaceno o další bezpečnostní prvky (HTTPs certifikát + již zmiňovaný IdentityServer). Pro ukázku si můžeme výslednou aplikaci zjednodušeně otestovat na instanci pro **localhost**, která není napojená na hlavní databázi a neobsahuje kontrolu autorizačních tokenů:

HTTP GET požadavek na zdroj Countrys:

```
curl -X GET --header 'Accept: application/json'
 'http://localhost:5001/api/Countrys'
```

Získáváme odpověď v JSON formátu všech dostupných zdrojů typu country (aktuálně tato databáze obsahuje pouze organizace sídlící v České Republice):

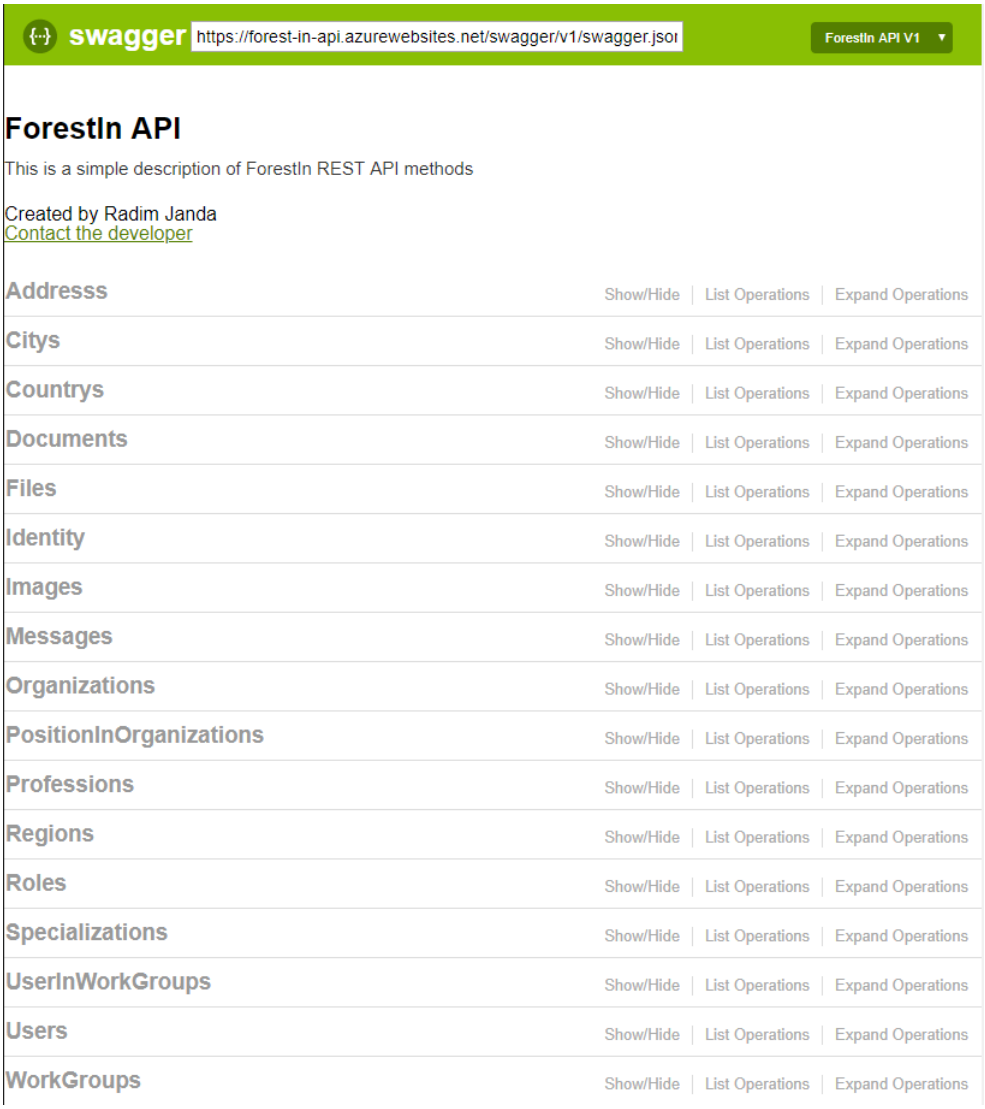

```
[
  {
    "id": 3,
    "name": "Ceska Republika",
    "createdBy": "default",
    "createdTime": "2017-12-11T01:40:25.2186832",
    "updatedBy": "default",
    "updateTime": "2017-12-11T01:40:25.2186832"
  }
]
```

5.4 Ukázky výsledného RESTful API

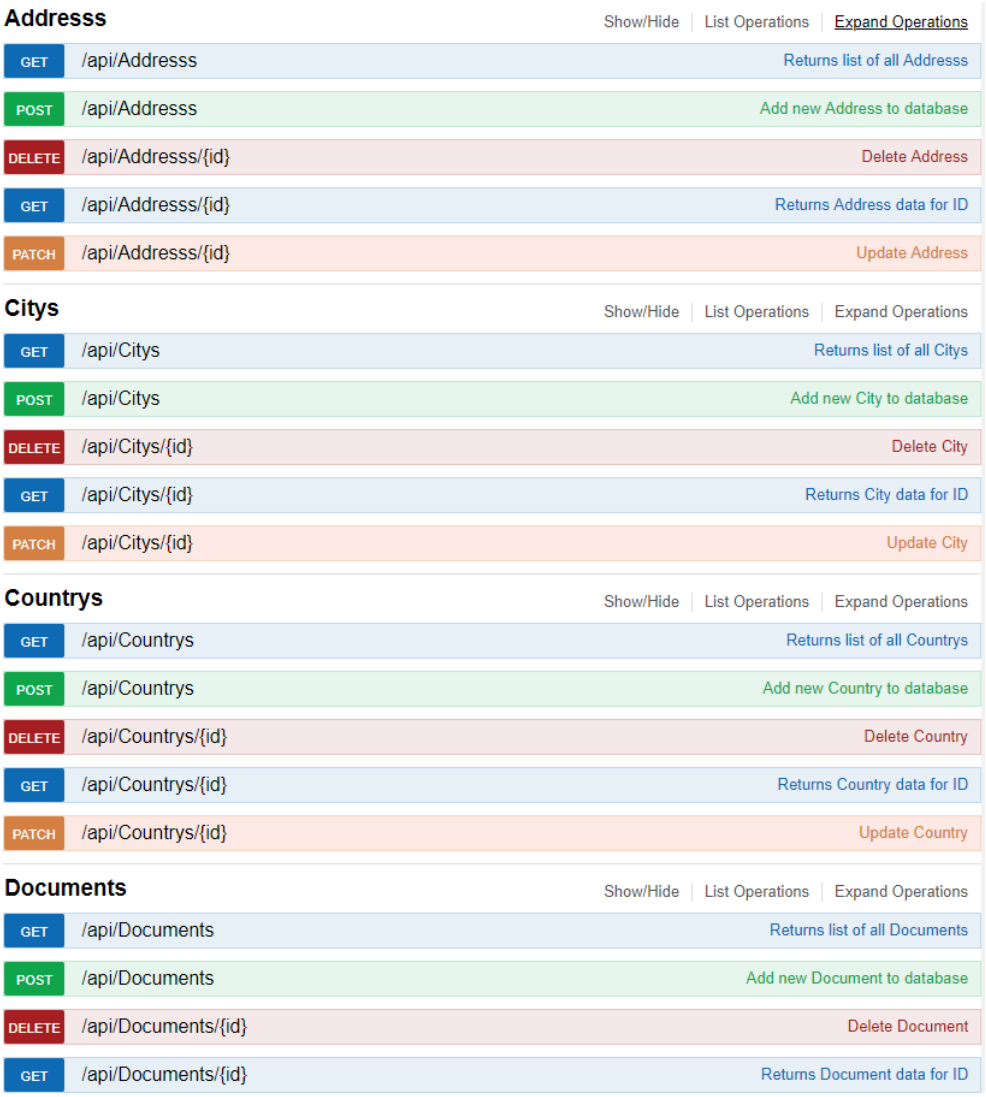
Výsledné kódy RESTful API aplikace obsahují **19 controllerů a modelů**. Dále také obsahují řadu konfiguračních tříd, služeb a zabezpečení. Aplikace kromě běžného přístupu ke zdrojům poskytuje i řadu dalších funkcí, mimo jiné:

- Napojení na API ARES (probráno v předchozí sekci)
- Práce s obrázky a dynamické formátování jejich rozlišení (rovněž zmíněno v předchozí sekci)
- Zasílání emailů - např. při odeslání zprávy, tedy vytvoření v rámci zdroje **messages**
- Zabezpečení - možnost přiřazovat uživatelům **role** a omezit jim tak přístup k některým zdrojům. Celkově je přístup k API povolen pouze s autorizačním tokenem získaným z aplikace Identity Server
- Pracuje skrze zabezpečený **HTTPS** protokol
- Nahrávání souborů na **Azure BLOB storage**
- Přehlednou dokumentaci, tato dokumentace je však z bezpečnostních důvodů zaheslovaná. Přístup mají pouze vývojáři klientských aplikací
- Uchovává údaje o zdrojích, jejich změnách a uživatelích, kteří tuto změnu provedli
- Loguje všechny důležité události. Tyto logy pak lze vidět v portálu Azure)
- Široké možnosti filtrování nad zdroji podle URL parametrů (typu: **GET /messages?senderid=123**)
- ...

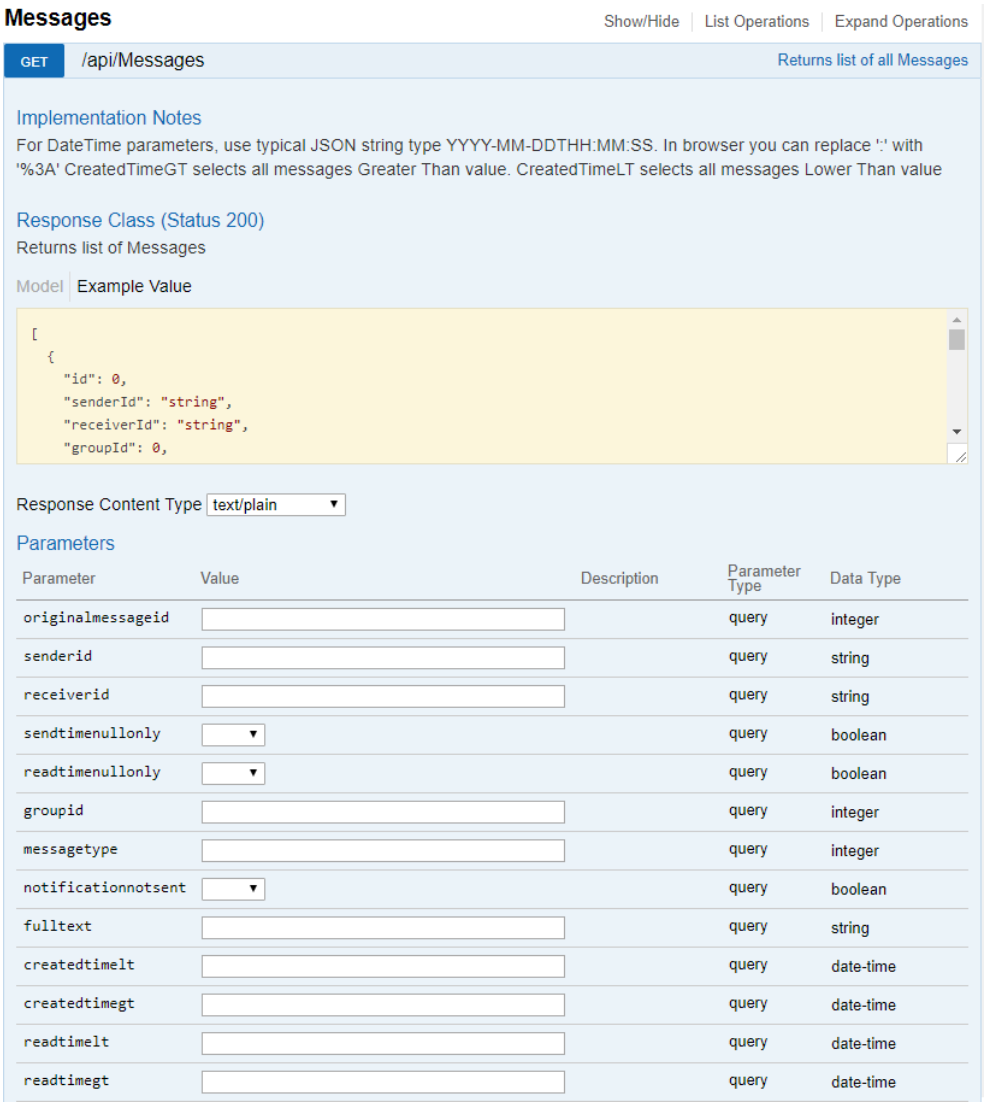
  Jelikož se s tímto API ve výsledku pracuje pouze v textovém rozhraní skrze HTTP požadavky, použijeme pro ukázku obrázky pořízené z automaticky vygenerované dokumentace:



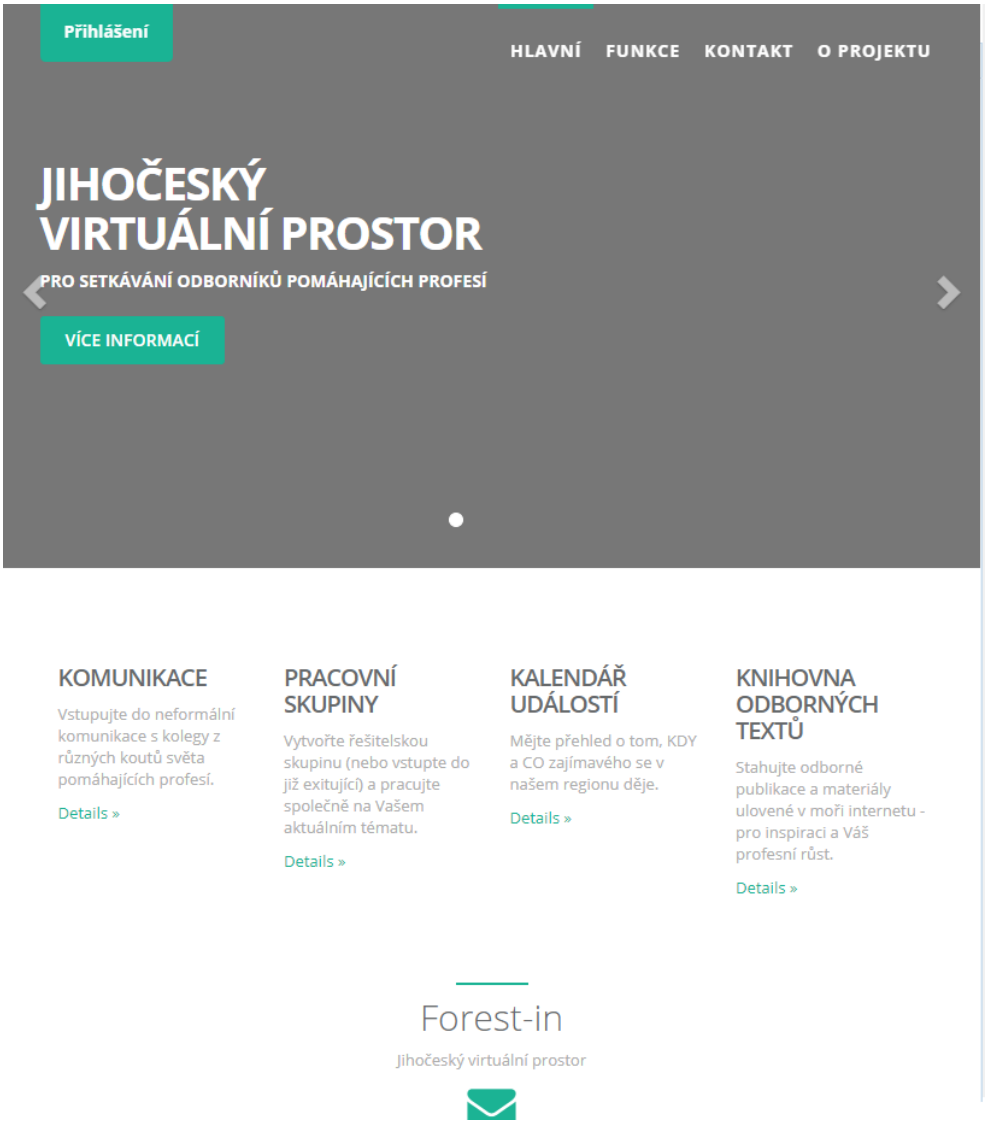
Obrázek 5.2: Ukázka z dokumentace výsledného RESTful API, vystihuje všechny dostupné zdroje



Obrázek 5.3: Ukázka z dokumentace výsledného RESTful API, vystihuje běžně dostupné metody nad zdroji



Obrázek 5.4: Ukázka z dokumentace výsledného RESTful API, vystihuje široké možnosti filtrování nad metodou GET



Obrázek 5.5: Ukázka z klientské aplikace používající implementované RESTful API. Zdroj: `https://forest-in.cz/`

5.5 Shrnutí

Vybraný projekt byl z hlediska části RESTful API aplikace úspěšně dokončen. Příslušné klientské aplikace jsou však stále ještě ve vývoji a lze je nalézt na adrese: `https://forest-in.cz/`, implementované RESTful API pak běží na adrese `https://forest-in-api.azurewebsites.net/swagger` (přístup k dokumentaci je však z bezpečnostních důvodů omezený). Při návrhu a imple-

mentaci byly použity všechny technologie ze sestavy stanovené v předchozí kapitole, rovněž se vycházelo z obecných znalostí REST architektury stanovených v první kapitole. Užitečnost konkrétních technologií pak byla shrnuta do následující tabulky:

Typ technologie	Vybraná technologie	Poznatky (užitečnost)
Fremework pro vývoj RESTful API aplikace	ASP.NET Core	Vysoce užitečné. Klíčové pro zjednodušení tvorby webových aplikací.
Nástroj pro Mockup RESTful API aplikace	Apiary	Užitečné pro úvodní návrhy. V dalších fázích se však ztratil význam Mockup udržovat.
Issue Tracking Systém	Bitbucket	Vysoce užitečné. Klíčové pro zaznamenávání změn a komunikaci v týmu.
Verzovací sytém	Git	Vysoce užitečné. Klíčové pro správu nad kódem.
Cloudová platforma	Microsoft Azure	Vysoce užitečné. Přináší spoustu funkcí a spolehlivé hostování aplikací.
Nástroj pro automatickou tvorbu dokumentace	Swagger	Vysoce užitečné. Klíčové pro udržení přehledu nad současným stavem aplikace.

Tabulka 5.1: Výsledná tabulka shrnující technologie ze sestavy stanovené v minulé kapitole a jejich skutečnou užitečnost při práci na tomto projektu

Zvolená sestava tedy v praxi obstála velice dobře. Žádná technologie v této sestavě výrazně nechyběla. Jediné, co by ještě mohlo ulehčit práci, by byly **generátory kódu**, které by automatizovaly neustále opakované postupy při psaní kódu. Generátory šablon obsažené v ASP.NET Core se neukázaly být pro tento projekt zcela vyhovující.

Kapitola 6

Návrh nástroje pro sjednocení technologií

V předchozích kapitolách jsme rozebírali spoustu technologií a nástrojů vhodných při tvorbě RESTful API. Tato kapitola na závěr diskutuje možnosti sjednocení těchto technologií a to návrhem univerzálního nástroje. Výstup této kapitoly lze pak brát jako úvod do možného tématu pro disertační práci.

6.1 Diskuze ohledně možností sjednocování technologií

V diskuzi budeme vycházet z informací získaných v předchozích kapitolách, tedy z rešerší technologií a ze zkušeností získaných v praktické části. Předpokládáme, že chceme vytvořit jednoduché RESTful API pro blog, tedy články, které mají komentáře, také chceme paralelně provozovat totožně fungující instance této aplikace založené na různých technologiích.

Z předchozích kapitol víme, že všechny probrané frameworky pro implementaci RESTful API jsou založené podle vzoru **model-view-controller**, což nám nabízí obrovské možnosti pro sjednocení, jelikož výsledné kódy těchto aplikací budou mít ve všech těchto frameworcích podobnou strukturu. Z 5. kapitoly také víme, že v praxi se nám při implementaci samotného RESTful API opakují pro každý zdroj identické postupy vytváření prakticky stejného kódu, jen s pozměněnými parametry vycházejícími z modelu daného zdroje. Tyto skutečnosti nám nabízí možnost navrhnout nástroj pro sjednocení těchto technologií jako **generátor kódu**. Cílem tohoto nástroje by tedy mohlo být připravit šablony kódů pro každý požadovaný framework, kde už by přidávání nových zdrojů mohlo tvořit automatizovanou činnost. Také bude třeba popsat požadované RESTful API univerzálním jazykem, který by tento nástroj rozpoznal a z tohoto popisu jej automaticky portovat do šablony zvoleného frameworku. K tomuto účelu již existují standardy pro podrobnou specifikaci

RESTful API typu **OpenAPI**[48], avšak ty jsou složité a obsahují až příliš podrobnou specifikaci. V našem případě chceme popsat RESTful API rychle a jednoduše, můžeme tedy vycházet z generátorů kódů ve frameworku **Rails**.

Zjednodušený univerzální jazyk pro popis RESTful API však zcela jistě bude mít jistá omezení, jelikož každá technologie má omezené možnosti. Pakliže se implementuje nástroj, který bude umět z tohoto popisu generovat totožné ale jednoduché RESTful API aplikace v různých frameworcích, zcela jistě bude třeba v těchto vygenerovaných kódech provádět jisté úpravy závislé na zvoleném frameworku, jinak by se tento vygenerovaný kód nelišil například od jednoduchého mockupovaného **JSON-Serveru** probraného v 3. kapitole. V takovém případě by bylo možné stanovit v šablonách místa pro vlastní kód a místa pro vygenerovaný kód, který by se neměl měnit. To nám přinese možnost například znovu přegenerovat celý projekt s novou verzí šablony, ale zachovat vlastní úpravy v kódu. Takové úpravy kódu by se často nacházely v **controllerech**, můžeme se podívat pro příklad na metodu pro vrácení všech zdrojů typu **Role** z našeho implementovaného API:

```
[HttpGet]
public IEnumerable<RoleModel> Get()
{
 // Zde by byl vlastni kod
 return _context.Set<RoleModel>();
}
```

Defaultně by se vracel celý set dat, avšak do části vlastního kódu by bylo možné přidat vlastní podmínky, které by za určitých okolností vrátily jiný upravený set nebo chybovou hlášku. To, jestli tato metoda přijímá nějaké parametry (tedy URL parametry HTTP požadavku), by pak muselo být v stanovené v univerzálním popisu.

Z podpůrných nástrojů by do tohoto univerzálního nástroje dávalo smysl zabudovat např. nástroje pro mockup a automatickou dokumentaci. Mockup by se pro ukázku v případě **Apiary** dal udělat snadno generováním zdrojového souboru pro Apiary vycházejícího z univerzálního popisu. Nástroje pro automatické generování dokumentaci typu **Swagger** by pak mohly být defaultně zabudovány v šabloně.

6.2 Teoretický návrh nástroje pro sjednocení těchnologií

Z diskuze provedené v předchozí sekci již víme, že se budeme zabývat navrhnutím nástroje, který by z šablony pro daný framework a univerzálního popisu RESTful API generoval projekt s požadovanými vlastnostmi. Takový nástroj by mohl fungovat v příkazové řádce, pojmenujeme ho prozatím **temprestgen**

(jako **Template RESTful Generator**).

První, co pro takovýto nástroj potřebujeme, je samostatně funkční šablona pro požadovaný framework, pro kterou by přidávání dalších zdrojů již tvořilo automatizovaný proces. Víme, že taková šablona existuje, jelikož v praktické části se tyto postupy téměř identicky opakovaly (vytvoření modelu podle parametrů definujících atributy tohoto zdroje, následně vytvoření controlleru, atd..). Také víme, že některé frameworky (například Rails) již automatické generování modelů a controllerů podporují. Pro ostatní frameworky s **model-view-controller** by to tedy rovněž neměl být problém. V našem nástroji bychom nejprve zvolili požadovanou šablonu, příkazem **template**:

```
temprestgen template aspnetcore -version 1.0
```

Následně by bylo zapotřebí popsat naše RESTful API univerzálním jazykem, podle kterého pak proběhne jeho vygenerování do šablon. Šablony by výjimečně mohly rozšiřovat tento univerzální jazyk i o vlastní příkazy, avšak hlavní symboly pro definování zdrojů by měly fungovat u všech šablon. K definice zdroje použijeme příkaz **resource**

```
temprestgen resource Article title:string content:string
temprestgen resource Comment author:string text:string
 article:reference
```

Z příkazů již lze logicky vyvodit, že jsme stanovili zdroj **Article**, jehož model bude mít atributy title a content, následně zdroj **Comment**, jehož model bude mít atributy author, text a bude přiřazen k nějakému článku.

Vygenerované **controllery** by pak podporovaly klasické GET, PUT, PATCH, DELETE, POST **HTTP** požadavky na tento zdroj. Často se však setkáme s tím, že chceme k určité metodě přiřadit nějaký URL parametr, podle kterého budeme filtrovat. K tomu můžeme přidat příkaz **resourceparams**

```
temprestgen resourceparams Article GET::(title:string
 content:string) GET:/{int}/comments:(author:string)
```

Tento příkaz již může být poněkud hůře čitelný. Příkaz by nejprve vybral zdroj, jehož controller budeme modifikovat, následně volíme ve formátu odděleném dvojtečkou:

- HTTP metodu kterou se budeme zabývat (tedy GET, PUT, atd.)
- URI pro konkrétní HTTP metodu, protože víme, že pro jeden zdroj typicky máme např. metodu GET na koncové body **/articles** nebo **/articles/id**, což jsou ve výsledku 2 rozdílné metody, které mohou přijímat různé parametry. Výchozí prázdná hodnota by představovala defalutní uri zdroje, tedy v tomto případě **/articles**.
- Atributy. Následuje výše v závorce uvedený seznam všech dostupných parametrů včetně jejich typů

Parametry by se tak přidaly do požadovaných metod v controllerech, ale jejich zpracování by však již záleželo na vlastním přidaném kódu. Přidání těchto parametrů by pak v controllerech mohlo vypadat takto - vidíme tyto parametry připravené v předem vygenerované metodě pro daný požadavek, jejich zpracování již závisí na vlastním kódu (demonstrováno na ASP.NET Core)

```
[HttpGet]
public IEnumerable<ArticleModel> Get(string title,
 string content)
{
 // misto pro vlastni kod
 if(title!=null)
  return _context.Set<ArticleModel>()
   .Where(x => title.Equals(x.title)));
 // konec mista pro vlastni kod

 return _context.Set<ArticleModel>();
}
```

Pokud by se nám již naše API podařilo popsat stanoveným univerzálním jazykem (prozatím si tedy vystačíme s příkazy **resource** a **resourceparams**), můžeme vše dokončit a nechat si vygenerovat vlastní projekt v požadovaném frameworku a s požadovanými vlastnostmi. tanovíme příkaz **finish**.

```
temprestgen resourceparams finish
```

6.3 Shrnutí

V této kapitole jsme diskutovali možnosti sjednocení různých technologií a nástrojů pro tvorbu RESTful API do univerzálního nástroje. Výsledkem kapitoly je předběžný teoretický návrh nástroje jehož **vstupem** je popsání daného RESTful API v univerzálním jazyce a **výstupem** portace tohoto API do zvoleného frameworku.

Univerzální jazyk pro popis RESTful API se aktuálně skládá z následujících příkazů:

template Výběr šablony pro výsledný vygenerovaný projekt

resource Popis zdrojů, tedy hlavně jeho modelu

resourceparams Popis parametrů, se kterými je možné pracovat v controlleru daného zdroje

finish Zakončení popisů a generování projektu

Oproti běžným jazykům pro specifikaci RESTful API typu **OpenAPI** přináší tento jazyk značné zjednodušení, a to v případě generování jednoduchých RESTful API aplikací.

Tento nástroj by také mohl doplňovat sestavu technologií stanovenou ve 4. kapitole, jelikož automatizuje časté opakování některých totožných postupů, což byl jediný velký zjištěný nedostatek při použití této sestavy.

Implementování, rozšíření a doladění tohoto nástroje by pak mohlo tvořit téma pro disertační práci.

Závěr

Cílem práce bylo rozebrat dostupné technologie a nástroje pro tvorbu RESTful API aplikací. Mezi další cíle patřilo i stanovení sestavy těchto technologií pro maximální efektivnost vývoje těchto aplikací a otestovat takovou sestavu při praktickém vývoji. Závěrečným cílem bylo diskutování možností sjednocení těchto technologií do univerzálního nástroje.

Rešeršní část dává čtenáři značný přehled ohledně těchto technologií a stanovuje zmiňovanou sestavu. Sestava byla použita k vytvoření RESTful API aplikace, která aktuálně plně funguje v systému dobročinného projektu. Sestava se při tomto praktickém vývoji ukázala být velmi efektivní a neprojevuje žádné zásadní vady.

Závěr práce shrnuje možnosti sjednocení těchto technologií do univerzálního nástroje. Nástroj funguje na principu generování kódu a stanovuje základy pro univerzální jazyk popisující RESTful API. Tímto nástrojem a jazykem je nadále možné se zabývat v disertační práci.

Bod 1	Rešerše technologií a nástrojů vypracována v kapitolách **2** a **3**
Bod 2	Sestava pro maximální podporu tvorby RESTful API navržena v kapitole **4**
Bod 3	Praktická část - shrnuto v kapitole **5**
Bod 4	Univerzální sjednocení technologií probráno v kapitole **6**

Tabulka 6.1: Tabulka shrnující splnění dílčích částí zadání práce. Vše také vychází z první kapitoly, která shrnuje základy architektury REST.

- Bod 1 - Vypracujte rešerši technologií a nástrojů, které se používají pro tvorbu RESTful API, včetně dalších podpůrných technologií a nástrojů například mockup pro AP.
- Bod 2 - Na základě předchozího bodu navrhněte konkrétní sestavu (stack) technologií a nástrojů, které budou tvorbu RESTful API v co největší míře podporovat.
- Bod 3 - Společně s vedoucím práce specifikujte vhodný příklad a demonstrujte návrh a implementaci RESTful API na sestavě technologií z bodu 2 tohoto zadání
- Bod 4 - Diskutujte možnost sjednocení technologií a nástrojů do univerzálního systému pro tvorbu jednoduchého RESTful API a jeho následné portace do zvolených technologií.

Literatura

[1] Architektura REST - Fielding. *www.ics.uci.edu* [online]. [cit. 2017-11-25]. Dostupné z: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[2] Architektura REST - itnetwork. *itnetwork.cz* [online]. [cit. 2017-11-25]. Dostupné z: https://www.itnetwork.cz/nezarazene/stoparuv-pruvodce-rest-api

[3] HTTP Patch - dokumentace. *tools.ietf.org* [online]. [cit. 2017-11-25]. Dostupné z: https://tools.ietf.org/html/rfc5789

[4] HTTP Status codes - dokumentace. *www.w3.org* [online]. [cit. 2017-11-25]. Dostupné z: https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

[5] MIME types - seznam. *developer.mozilla.org* [online]. [cit. 2017-11-25]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Complete_list_of_MIME_types

[6] REST - základní možnosti zabezpečení. *jamiekurtz.com* [online]. [cit. 2017-11-25]. Dostupné z: http://www.jamiekurtz.com/2013/01/14/asp-net-web-api-security-basics/

[7] OAuth 1.0 a OAuth 2.0. *oauth.com* [online]. [cit. 2017-11-25]. Dostupné z: https://www.oauth.com/oauth2-servers/differences-between-oauth-1-2/

[8] Časté chyby v architektuře REST. *programmableweb.com* [online]. [cit. 2017-11-25]. Dostupné z: https://www.programmableweb.com/news/api-anti-patterns-how-to-avoid-common-rest-mistakes/2010/08/13

[9] OAuth 2. *backstage.forgerock.com* [online]. [cit. 2017-11-25]. Dostupné z: https://backstage.forgerock.com/docs/am/5/oauth2-guide/

[10] REST API Mock up. *soapui.org* [online]. [cit. 2017-11-25]. Dostupné z: https://www.soapui.org/rest-testing-mocking/service-mocking-overview.html

[11] ASP.NET Core. *microsoft.comorg* [online]. [cit. 2018-09-03]. Dostupné z: https://docs.microsoft.com/cs-cz/aspnet/core/

[12] Visual Studio. *visualstudio.com* [online]. [cit. 2018-09-03]. Dostupné z: https://www.visualstudio.com/cs

[13] Microsoft Azure. *microsoft.com* [online]. [cit. 2018-09-03]. Dostupné z: https://azure.microsoft.com/cs-cz/

[14] MVC Architektura. *microsoft.com* [online]. [cit. 2018-09-03]. Dostupné z: https://docs.microsoft.com/cs-cz/aspnet/core/mvc/overview

[15] ASP.NET Core - Entity Framework. *microsoft.com* [online]. [cit. 2018-13-03]. Dostupné z: https://docs.microsoft.com/cs-cz/aspnet/core/data/ef-mvc/intro

[16] NuGet Package Manager. *microsoft.com* [online]. [cit. 2018-13-03]. Dostupné z: https://docs.microsoft.com/en-us/nuget/tools/package-manager-ui

[17] Diskuze - popularita technologií. *quora.com* [online]. [cit. 2018-13-03]. Dostupné z: https://www.quora.com/How-popular-is-ASP-NET

[18] ASP.NET Core - zabezpečení. *microsoft.com* [online]. [cit. 2018-13-03]. Dostupné z: https://docs.microsoft.com/cs-cz/aspnet/core/security/

[19] Java Spring - REST služby. *spring.io* [online]. [cit. 2018-13-03]. Dostupné z: https://spring.io/guides/gs/rest-service/

[20] JacksonJSON. *fasterxml.com* [online]. [cit. 2018-14-03]. Dostupné z: http://wiki.fasterxml.com/JacksonHome

[21] MongoDB. *mongodb.com* [online]. [cit. 2018-14-03]. Dostupné z: https://www.mongodb.com/

[22] Gradle. *gradle.org* [online]. [cit. 2018-14-03]. Dostupné z: https://gradle.org//

[23] Maven. *apache.org* [online]. [cit. 2018-14-03]. Dostupné z: https://maven.apache.org/

[24] Java Spring - Bezpečnost. *spring.io* [online]. [cit. 2018-14-03]. Dostupné z: https://projects.spring.io/spring-security/

[25] Apiary. *apiary.io* [online]. [cit. 2018-14-03]. Dostupné z: https://apiary.io/

[26] JSON Server. *apiary.io* [online]. [cit. 2018-14-03]. Dostupné z: https://github.com/typicode/json-server

[27] Issue Tracking Software. *capterra.com* [online]. [cit. 2018-16-03]. Dostupné z: https://www.capterra.com/issue-tracking-software/

[28] Bitbucket. *bitbucket.org* [online]. [cit. 2018-17-03]. Dostupné z: https://bitbucket.org/

[29] Git. *git-scm.com* [online]. [cit. 2018-17-03]. Dostupné z: https://git-scm.com/

[30] Notifications Hub. *microsoft.com* [online]. [cit. 2018-17-03]. Dostupné z: https://docs.microsoft.com/cs-cz/azure/notification-hubs/

[31] Microsoft Azure zabezpečení. *microsoft.com* [online]. [cit. 2018-17-03]. Dostupné z: https://azure.microsoft.com/cs-cz/overview/trusted-cloud/

[32] Swagger. *swagger.io* [online]. [cit. 2018-18-03]. Dostupné z: https://swagger.io/

[33] Rails. *rubyonrails.org* [online]. [cit. 2018-22-03]. Dostupné z: http://rubyonrails.org/

[34] Rails - Rspec. *github.com* [online]. [cit. 2018-22-03]. Dostupné z: https://github.com/rspec/rspec-rails

[35] Popularita Frameworků. *medium.com* [online]. [cit. 2018-22-03]. Dostupné z: https://medium.com/@yoelblum_45935/is-rails-still-popular-in-2018-d17f3b062b18

[36] Rails popularita. *medium.com* [online]. [cit. 2018-22-03]. Dostupné z: https://medium.com/@TechMagic/nodejs-vs-ruby-on-rails-comparison-2017-which-is-the-best-for-web-development-9aae7a3f08bf

[37] Django. *djangoproject.com* [online]. [cit. 2018-22-03]. Dostupné z: https://www.djangoproject.com/

[38] Django REST framework. *django-rest-framework.org* [online]. [cit. 2018-22-03]. Dostupné z: http://www.django-rest-framework.org/

[39] Node.js. *nodejs.org* [online]. [cit. 2018-23-03]. Dostupné z: https://nodejs.org/en/

[40] ExpressJS. *xpressjs.com* [online]. [cit. 2018-23-03]. Dostupné z: https://expressjs.com/

[41] ExpressJS bezpečnost. *xpressjs.com* [online]. [cit. 2018-23-03]. Dostupné z: https://expressjs.com/en/advanced/best-practice-security.html

[42] Popularita vybraných frameworků - Stackoverflow. *stackoverflow.com* [online]. [cit. 2018-09-04]. Dostupné z: https://insights.stackoverflow.com/trends?tags=django%2Cspring%2Cruby-on-rails%2Cexpress%2Casp.net-core

[43] Životní cyklus vývoje software. *xbsoftware.com* [online]. [cit. 2018-09-04]. Dostupné z: https://xbsoftware.com/blog/software-development-life-cycle-waterfall-model/

[44] Identity Server. *identityserver.io* [online]. [cit. 2018-09-04]. Dostupné z: http://docs.identityserver.io/en/release/

[45] ARES. *wwwinfo.mfcr.cz* [online]. [cit. 2018-09-04]. Dostupné z: http://wwwinfo.mfcr.cz/ares/ares_es.html.cz

[46] XDocument. *microsoft.com* [online]. [cit. 2018-09-04]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/system.xml.linq.xdocument(v=vs.110).aspx

[47] ImageSharp. *github.com* [online]. [cit. 2018-09-04]. Dostupné z: https://github.com/SixLabors/ImageSharp

[48] OpenAPI specifikace. *swagger.io* [online]. [cit. 2018-09-04]. Dostupné z: https://swagger.io/specification/

[49] RICHARDSON, Leonard a Michael AMUNDSEN. *RESTful Web APIs.* Beijing: O'Reilly, 2013. ISBN 9781449358068.

Příloha **A**

Seznam použitých zkratek

REST Representational state transfer

API Application Programming Interface

GUI Graphical User Interface

HTTP Hypertext Transfer Protocol

XML Extensible Markup Language

JSON JavaScript Object Notation

IT Informační Technologie

URI (URL) Unique Resource Identifier (Locator)

Příloha B

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace v praktické části
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF