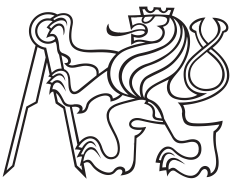
Diplomová práce



České
vysoké
učení technické
v Praze

F8

Fakulta informačních technologií
Katedra softwarového inženýrství

Dokumentačně-monitorovací systém pro správu prostředí s mnoha aplikacemi

Bc. Jakub Šimon
Obor: Webové a softwarové inženýrství

Červen 2015
Vedoucí práce: Ing. Ondřej Mysliveček

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Dokumentačně-monitorovací systém pro správu prostředí s mnoha aplikacemi
Student:	Bc. Jakub Šimon
Vedoucí:	Ing. Ondřej Mysliveček
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2015/16

Pokyny pro vypracování

Moderní SW aplikace jsou často velmi provázané pomocí API. Tato provázanost přináší i problémy a rizika v případě změn API některé z nich.
Cílem práce je návrh metodiky vývoje a dokumentace API v SW ekosystému tak, aby bylo možné jejich provázanost sledovat a řídit. Dále pak návrh, implementace a nasazení nástroje pro sledování provázanosti API aplikací v SW ekosystému.
1. Vypracujte studii existujících nástrojů, které řízení provázanosti a dokumentace API podporují.
2. Vypracujte metodiku pro návrh a dokumentaci API - zejména strukturu a verzování.
3. Navrhněte systém pro monitorování a dokumentaci API v rámci celého vývojářského ekosystému.
4. Diskutujte a zvolte vhodné implementační prostředky - pro dílčí úkoly lze použít hotové moduly, pokud existují.
5. Systém implementujte, nasaďte, zdokumentujte a na zvolených aplikacích a jejich API dokumentujte užitečnost vyvinutého systému.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof.Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 13. února 2015

Poděkování / Prohlášení

Děkuji vedoucímu práce Ing. Ondřeji Myslivečkovi za vedení práce a mnohé konzultace i užitečné rady.

Rád bych také poděkoval své rodině, přátelům a přítelkyni za jejich podporu a trpělivost po celou dobu studia.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl 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.

V Berouně dne 22. 6. 2015

.....

Abstrakt / Abstract

Tato práce se zabývá problematikou provázanosti aplikací a jejich komunikace přes webové API. Shrnuje základy REST architektonického stylu a prezentuje metodiku pro návrh a dokumentaci REST API. V práci byla prezentována analýza požadavků a vytvořen návrh pro systém monitorující a spravující vazby mezi aplikacemi. Hlavním přínosem práce je shrnutí problematiky REST API a metodiky pro návrh a dokumentaci.

Klíčová slova: API, REST, provázanost aplikací, API management, API monitoring, návrh API, API dokumentace, metologie tvorby REST API

The thesis deals with issue of the interconnection of applications and their communications through the Web API. It summarizes the basics of the REST architectural style and it presents a methodology for designing and documenting of REST API. In the thesis there is presented an analysis of the system's requirements and there was suggested a proposal for a system of monitoring and managing the relationships between applications. The main contribution of the thesis is a summary of the issue of REST API and methodology for design and documentation.

Keywords: API, REST, application relationship, API management, API monitoring, API design, API documentation, REST API design methology

Obsah /

Tabulky / Obrázky

Kapitola 1
Úvod

Uvnitř firmy, která provozuje jeden či více větších internetových produktu obvykle vzniká vnitřní ekosystém, který rozděluje produkt na menší aplikace a komponenty. Ty navzájem komunikují pomocí webových API. S přibývajícím počtem těchto spojení vzniká nutnost pro jejich administraci a monitoring. Cílem této práce je navrhnout a vytvořit nástroj, který ulehčí tuto správu.

Struktura práce je následující: V kapitole 2 představím popis řešeného problému, stanovení cílů DP a požadavků na implementaci v podobě user stories. V kapitole 3 vytvořím úvod do webových API, popíši základy REST, doplněné i o vybrané formáty reprezentace a formáty pro popis a definici API. V následující kapitole 4 popíši metodiku a doporučení pro návrh a dokumentaci API. Empirická část práce bude zpracována v kapitole 5, kde vytvořím analýzu pro stanovené požadavky a následně podle nich udělám v kapitole 6 rešerši stávajících nástrojů. V kapitole 7 představím návrh budoucího systému.

Kapitola 2
Popis řešeného problému, stanovení cílů DP a požadavků na implementaci

V této kapitole představím úvod do problematiky vývoje a provozu ekosystému aplikací samostatných, ale navzájem komunikujících aplikací. Dále vyložím terminologii důležitou pro potřeby této práce a stanovím cíle práce doplněné o detailnější požadavky, které se nevešly do zadání.

2.1 Úvod do problematiky

Uvnitř většího firemního ekosystému existuje obvykle několik nezávislých aplikací, které poskytují a konzumují různá API. Tyto aplikace mohou mezi sebou být propojené různými vazbami.

Aplikace poskytující API vytvářejí vztah konzument–producent, kdy konzumující získává závislost. Pro producenta tento vztah znamená uzavření „kontraktu“, že API bude fungovat dle své definice.

Pokud se tyto vztahy zmapují, vznikne orientovaný graf závislostí jednotlivých aplikací. Tento graf je následně využitelný při změnách a nasazování nových verzí jednotlivých aplikací, kdy lze poznat závislé aplikace, které mohou mít při nasazení také výpadek, či případně aplikace, které je při vývoji nutné otestovat, zda změna v závislé aplikaci neovlivnila jejich správnou funkčnost.

Tento vztah také vytváří nutnost monitoring (počet požadavků od jednotlivých konzumentů, počet chybných dotazů/odpovědí), který může následně odhalit chyby při nasazení nové verze aplikace a usnadnit efektivní rozvoj producenta, který zná své konzumenty.

Příkladem sdílené aplikace může být GIS[1]) poskytující informace o adresách – nalezení a validace adresy, hierarchie, geokódování[2]) a další. Tuto aplikaci a její API budou používat ostatní aplikace, pokud budou pracovat s adresami nebo jinými geografickými daty. Takovýchto konzumujících aplikací může být velké množství a pokud by nastala nutnost změny API, tak je třeba informovat všechny vlastníky těchto konzumentů a upravit jejich chování. Současně je dobré vědět, pokud některá část API přestane být používána a je možné ji zrušit a tím ušetřit systémové prostředky nebo odstranit nepoužívaný a neudržovaný kód.

2.2 Terminologie

V této práci jsem se rozhodl použít do češtiny nepřeložené anglické termíny. Hlavním důvodem je, že překlad může měnit význam jednotlivých termínů, navíc většina zdrojů používá právě anglické termíny, tudíž jejich počeštění by bylo umělé.

[1]) Geografický informační systém
[2]) převod adresy na zeměpisné souřadnice a naopak

V následujícím seznamu jsou dále použité termíny s jejich významy a českým překladem:

- **Resource**: Zdroj – abstrakce informace, která může být pojmenovaná – dokument, obrázek, dočasná služba (dnešní počasí v Praze), reálný objekt (např. osoba) a další. Definice z disertační práce R. Fieldinga[1][1])
- **Endpoint**: Koncový bod – kombinace HTTP metody a základní adresy resource (URI) bez volitelných parametrů a atributů (např. `http://example.com/api/resource/1?arg=2` má stejný endpoint jako `http://example.com/api/resource/2?arg=4`)
- **Hypertext**: Zobrazení textu jednoduše čitelného pro čtenáře na elektronickém zařízení.[2])
- **Hyperlink**: Hyperlink (odkaz) – část textu či obrázek, na který se dá kliknutím dostat na jiný dokument či část dokumentu
- **User story**: Uživatelský příběh – Český ekvivalent se v praxi nepoužívá. User story je popis úkolu a jeho zadání používaný u agilních metodik. Agilní kouč Zuzana Šochová ji na svém blogu [3] definuje jako:

  „Jako Uživatel chci Funkcionalitu, abych dostal Business Value“.

  Jiří Knesl ve svém článku Návod na User Stories [4] dodává vysvětlení:

  „User Story je takový způsob zadávání práce, kdy na konci vždy zůstane něco, co je pro někoho užitečné.“

Dále také v práci budu používat termíny *konzument* a *producent*:

- **Consumer**: Konzument – systém využívající API; klient v server-klient architektuře
- **Producer**: Producent – systém poskytující API; server v server-klient architektuře
- **API**: soubor jednotlivých endpointů a akcí nad nimi, které jsou součástí jedné aplikace. Jedna aplikace může obsahovat několik API.

2.3 Stanovení cílů DP

Stanovil jsem několik dílčích cílů, kterých bych chtěl v této práci dosáhnout. Prvním cílem je návrh metodiky vývoje a dokumentace API v SW ekosystému tak, aby bylo možné jejich provázanost sledovat a řídit.

Dalším cílem je pak návrh, implementace a nasazení nástroje pro sledování provázanosti API aplikací v SW ekosystému. Tento nástroj by měl být využitelný uvnitř organizace využívající primárně interní API.

2.3.1 Původní nástroj

Původní nástroj vyvinutý ve firmě LMC se sestával ze skriptu, který vygeneroval z poskytnutých dat statický dokument o aplikacích a jejich vztazích. Data byla uložena ve formě JSON souboru, který obsahoval ručně vyplněné informace o aplikaci, poskytovaných a konzumovaných API. Nástroj také uměl zobrazit graficky vztahy mezi aplikacemi jako orientovaný graf – aplikace byly zobrazeny jako uzly a vztahy jako orientované hrany.

Problémem původního řešení byla statičnost, kdy se pokaždé generovala nová verze dokumentu a bylo složité hledat změny ve vztazích. Statický dokument také neposkytoval žádnou možnost interaktivity s uživatelem, stejně tak chybělo upozornění, pokud

[1]) strana 88

[2]) V kontextu webu se o něm prvně zmiňuje T. Berners-Lee návrhu WorldWideWeb: Proposal for a HyperText Project [2]

nějaká vazba mezi systémy obsahovala chybu, nebo potřebovala pozornost. Neposkytovala ani řešení pro funkční požadavky specifikované dále.

2.3.2 Funkční požadavky – User stories

Funkční požadavky na aplikaci jsou specifikovány ve formě user stories a jejich detailnější rozbor bude popsán v kapitole 5.

User story 2.1. Jako vývojář chci znát endpointy, které poskytují moje aplikace, které nejsou konzumovány jinými aplikacemi, abych je mohl případně odstranit.

User story 2.2. Jako vývojář chci přehledně vidět závislosti aplikace, kterou právě upravuji, abych mohl analyzovat dopad změn na API v předstihu a případně diskutovat změny s vlastníky dotčených aplikací.

User story 2.3. Jako vývojář / DEVOPS[1]) chci znát výkonnost API mých aplikací, abych mohl zajistit požadovanou dostupnost a kvalitu služby.

User story 2.4. Jako vývojář chci mít přehled o API, které konzumují mé aplikace, abych v případě problémů věděl, které API je způsobuje a měl možnost kontaktovat jeho vlastníka.

User story 2.5. Jako vývojář chci být informován v případě, že verze API využívané mojí aplikací je nahrazené novější verzí (a stará verze se stane neaktuální a nepodporovanou), abych mohl přejít na aktuální verzi.

User story 2.6. Jako vývojář chci znát dostupnost konzumovaných API, abych mohl navrhnout svojí aplikaci pro maximální spolehlivost a funkčnost

User story 2.7. Jako vývojář chci získat na jednom místě podrobný popis všech dostupných API, které může moje aplikace využít, abych rychleji získal informace a mohl vytvořit novou funkcionalitu rychleji a neimplementoval již existující službu.

User story 2.8. Jako vývojář, DEVOPS a IT architekt chci vidět změny ve využití API a dostat upozornění, pokud se některé API chová zvláštně, abych mohl identifikovat chyby a problém a případně mohl lépe naplánovat systémové zdroje.

2.3.3 Nefunkční požadavky

Nefunkční požadavky na nástroj jsou následovné:

- Webově přístupná aplikace s webovým API
- Nedostupnost nebo výpadek aplikace nebude ovlivňovat komunikaci producenta a konzumenta. Pro měření konzumování API není preferované řešení přes centrální bod (jako API Gateway), ale přes asynchronní systém nezasahující do komunikace producenta a konzumenta.
- Použité technologie budou využitelné zdarma i ke komerčnímu použití

[1]) DEVOPS vzniklo jako kontaminace slov „development“ a „operations“. V tomto případě označuje roli na pomezí mezi vývojáři a systémovými administrátory. Více informací v knize DevOps for Developers [5]

Kapitola 3
Web API a REST

V této kapitole budu popisovat pohled na webové API. V části 3.1 se budu zabývat definicí API. V části 3.2 popíši architekturu microservices, která s API úzce souvisí. V části 3.3 popíši pojem Representational State Transfer (dále jen REST), který byl definován v disertační práci R. Fieldinga [1] a nastíním jeho základní myšlenky a principy. V návaznosti na definici REST uvedu v kapitole 3.4 alternativní zjednodušený model implementace REST API dle Richardsona. V následujících částech 3.5 a 3.6 budu diskutovat možnosti popisu a dokumentace API na různých úrovních a zmíním vybrané média typy. V 3.7 shrnu možnosti zabezpečení API, zvláště z pohledu autentizace, a v poslední části této kapitoly 3.8 popíši životní cyklus API.

3.1 API a Web API

Application Programming Interface (dále jen API) je technologická architektura, která umožňuje jednoduše jedné aplikaci konzumovat data nebo funkce z jiné aplikace. U Web API jsou data a aplikační logika dostupná konzumentovi přes síť.

V knize APIs: A strategy guide [6][1]) popisují autoři Jacobson, Brail a Woods definici API následovně:

API je v podstatě kontrakt. Jakmile tento kontrakt existuje, tak vývojáři jsou lákáni využít API, protože ví, že se na něj mohou spolehnout. Kontrakt zvyšuje důvěru a ta zvyšuje využití. Kontrakt také vytváří efektivnější spojení mezi producentem a konzumentem, protože rozhraní je zdokumentované, konzistentní a předvídatelné.

API se dají rozdělit do dvou hlavních skupin dle jejich publikace – *veřejné* a *interní* API. Existuje více pohledů na toto rozdělení např. Willmott z 3Scale popisuje v článku [7] o jejich nástroji tři skupiny API – interní, privátní a veřejné. O'Neill z Axway se v článku [8] zmiňuje o dvou ortogonálních osách:

- zveřejnění API – *externí* a *interní*
- API management – *otevřené všem*, *vyžadující registraci* a *„temná" API*[2])

V této práci se budu držet jen rozdělení na dvě hlavní skupiny – *veřejné* a *interní* API.

3.1.1 Veřejné API

API je veřejné, pokud ho má libovolný konzument možnost využít. Existují dva druhy veřejných API – API přístupné bez jakékoliv identifikace nebo autentizace a veřejné API dostupné komukoliv, kdo si vytvoří účet nebo jinak získá identifikační údaj. První přístup úplně otevřeného API není doporučitelný především z důvodu, že producent pak nedokáže jednoduše zjistit, kteří konzumenti API využívají. Existují různé modely a strategie využití veřejných API:

[1]) Strana 4.
[2]) API nejsou nijak explicitně dokumentovaná, zveřejňována a nabízena.

- přístupné všem zdarma
- omezení na počet či četnost dotazů
- placený přístup
- placené dotazy nad limit
- a spousta dalších

Veřejná API se obvykle snaží dosáhnout globálnějšího cíle – zpřístupnění vlastní služby pro strojové zpracování, integraci, poskytnutí zajímavých funkcí a dat. U veřejných API je pro producenta výhodou nabídnout své služby co největší škále konzumentů, které se snaží získat pro svůj prospěch.

Konzumenti se o veřejných API musí nějak dozvědět. Pokud služba poskytuje API, tak obvykle odkazuje na svůj vývojářský portál či API dokumentaci ze svého webu. Pokud konzument hledá API naplňující jeho potřebu, tak má možnost využít klasických webových vyhledávačů a hledat zmínky o dokumentaci API. Druhá možnost existuje ve využití adresářů pro veřejná API – např. *ProgrammableWeb* obsahující 13 000 API [9] nebo *PublicAPIs* s 5 300 API [10]. Tyto adresáře jsou ale ručně plněny a udržovány, tak často jejich data zastarávají.

Alternativou adresářů je experimentální služba APIS.io [11], která se snaží řešit problém objevitelnosti API pomocí formátu APIs.json [12]. APIs.json je navržen pro strojové zpracování a nachází se vždy v kořenu domény[1]). Jeho cílem je jednoduchý popis dostupných API na dané doméně a poskytnutí informací o jejich funkčnosti. Obdobně jako vyhledávače pro web by tento formát umožnil automatické objevení API specializovanými vyhledávači pro API.

3.1.2 Interní API

Druhým pólem k veřejným API jsou *interní*. Rozdílem mezi interními a veřejnými API je v záměru nabízet libovolným konzumentům. Interní API jsou k dispozici jen pro interní vývojáře nebo domluvené a smluvní externí partnery. Interní API také na rozdíl od veřejných neinzerují popis a dokumentaci API.

Vytváření a využívání interních API je jednou z možných strategií v přístupů k architektuře uvnitř organizace. Rozdělením na jasně definované API a jejich rozhraní dochází k přirozenému oddělení zodpovědností jednotlivých aplikací a to následně usnadňuje vývoj.

Výhodou interních API je také možnost rychlého vyvinutí mobilních aplikací či dalších integrací s dalšími vlastními či cizími systémy, protože API je již hotoé a nemusí se navrhovat a implementovat. Služby nebo data poskytovaná API jsou následně lépe dostupná.

Pokud jsou aplikace stavěny pomocí architektury microservices, tak jsou interní API často využity pro definování rozhraní mezi jednotlivými službami.

3.2 Microservices architektura

Jak bylo naznačeno v předchozí kapitole, tak s interními API má spojitost i architektura microservices. V této části představím její širší kontext.

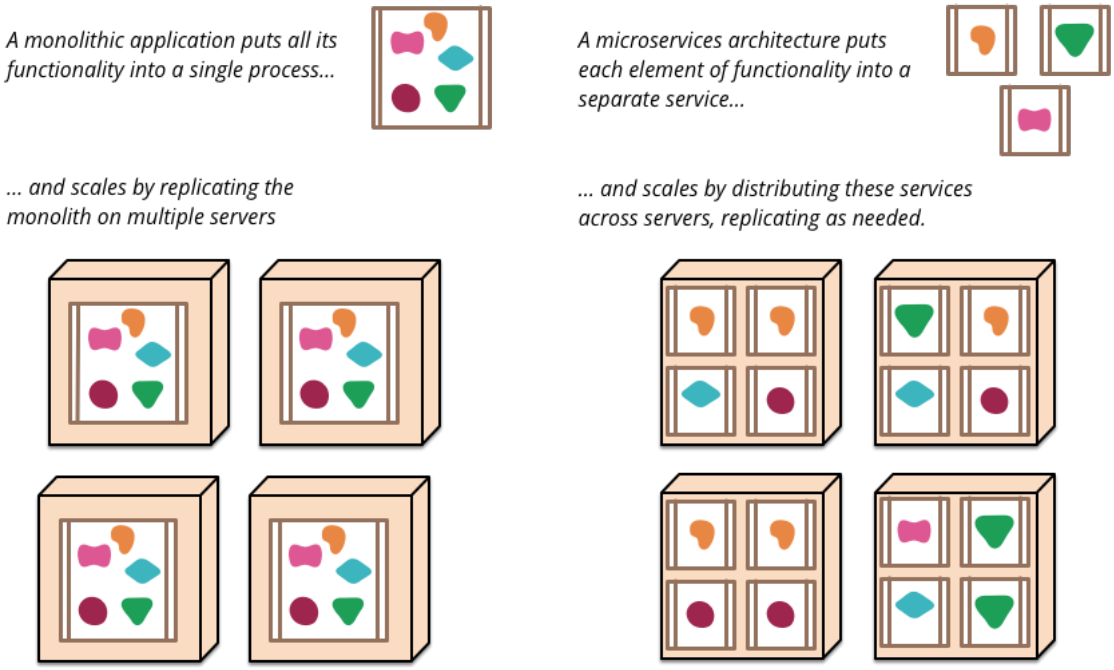
Lewis a Fowler definují microservices architekturu ve svém článku [13] následovně:

Architektonický styl microservices je přístup k vývoji aplikace jako souboru malých (mikro) služeb, každé běžící ve svém vlastním procesu a komunikující jednoduchým mechanismem, často pomocí HTTP API.

[1]) http://example.com/apis.json

Hlavní myšlenkou microservices je rozbití velkých monolitických aplikací na menší nezávislé části (služby či komponenty). Jedním z problémů monolitických systémů je jejich pomalá změna, kdy se kvůli malé změně musí vykonat build a nasazení nové verze celého systému, což může velmi pomalý proces. Navíc se špatně dohlíží na dopady i relativně jednoduchých změn, které pak často mají nečekané dopady v různých částech monolitu.

Microservices přináší výhody v možnosti použití různých programovacích jazyků na různé komponenty a v nezávislosti nasazení jednotlivých komponent. Důsledkem je lepší škálovatelnost po jednotlivých službách, kdy je možné poskytnout více prostředků komponentám, které je vyžadují. Porovnání škálovatelnosti microservices s monolitickým systémem je znázorněno na obrázku 3.1. Rozdělení na jednotlivé komponenty a samostatné nasazení také umožňuje vývoj více nezávislými týmy.



Obrázek 3.1. Microservices: Monolitická versus microservices architektura a jejich škálovatelnost, zdroj [13]. Monolitická architektura vkládá všechnu funkcionalitu do jednoho procesu a škáluje replikováním na více serverů. Microservices rozděluje jednotlivé funkcionality do samostatných služeb a škáluje je distribuováním přes servery a replikuje je dle potřeby.

Dále ve svém článku [13] Lewis a Fowler popisují hlavní charakteristiky microservices architektury:

- *Komponenty prostřednictvím služeb* – Komponentou je myšlena část softwaru, která je nezávisle vyměnitelná a vylepšitelná. Služby jsou procesy, které komunikují pomocí webových služeb či RPC[1]). Rozdělení na komponenty dává výhodu explicitní definice rozhraní každé komponenty, ale přináší také nevýhodu v nutnosti složitější komunikace mezi sebou.
- *Organizace postavená okolo komponent* – Přístup microservices doporučuje organizační rozdělení po týmech skládajících se z různých rolí (od expertů na uživatelské rozhraní až po databázové specialisty) namísto týmů specializovaných jen na jednu část. Tím je zaručena jasnější hranice daných komponent.

[1]) Remote Procedure Call – vzdálené volání procedur

- *Zaměření na produkty místo projektů* – Microservices razí přístup, kdy tým vlastní produkt po celý životní cyklus – od jeho vytvoření až po provoz. Tento přístup vytváří reálnější pohled na produkt jak ze strany vývojářů, tak i z pohledu jeho uživatelů a vede k lepšímu pochopení jejich problému a potřeb.
- *Chytré endpointy a jednoduché kanály* – Aplikace postavené z microservices se snaží být co nejvíce oddělené a zároveň soudržné, jak je to jen možné. Komunikace mezi komponentami by měla být co nejjednodušší a průhledná tak, že samotná logika zůstává v odpovědnosti komponent.

  První z hlavních dvou způsobů komunikace mezi komponentami je pomocí webového API (obvykle REST) a založené na principech a protokolu webu (HTTP).

  Druhým způsobem je použití posílání zpráv přes jednoduchou frontu (např. RabbitMQ[1]) nebo ZeroMQ[2])) zajišťující asynchronní komunikaci.
- *Decentralizovaná governance* – Centralizovaná governance přináší tendence standardizovat jednu technologickou platformu. Naopak microservices podporují její různorodost, kdy pro různé problémy lze použít různé nástroje, platformy a programovací jazyky, které řeší daný problém nejlépe.
- *Automatizace infrastruktury* – Produkty či systémy založené na microservices více prosazují principy Continous Integration[3]) a Continous Delivery[4]), které dávají důraz na automatické testy a automatické nasazení na různá prostředí. To zlepšuje důvěru ve správnou funkčnost systému a větší flexibilitu při vývoji.
- *Návrh odolný na chyby* – Důsledkem využití služeb jako komponent je nutnost návrhu jednotlivých celků být tolerantní k chybám ostatních služeb. Jakákoliv služba může být nedostupná a aplikace by měla být navržena tak, aby elegantně reagovala na chybu a odpověděla srozumitelně klientu (nezobrazení nedůležité komponenty, nebo zobrazení hlášky o nedostupnosti služby). V porovnání s monolitickou architekturou se jedná o nevýhodu, která přidává komplexitu, a zlepšuje chování aplikace při různých výpadcích, které mohou nastat. Důsledkem je větší důraz na real-time monitoring a sledování definovaných metrik, které odhalí případný problém velmi brzo.

3.3 Representational State Transfer

Representational State Transfer (dále jen REST) je architektonický styl pro distribuované hypermédiové systémy, který definoval a popsal Roy Fielding ve své disertační práci [1]. Fielding definuje REST jako soubor následujících podmínek aplikovaných na všechny elementy uvnitř architektury:

- *Client–Server* – styl klient-server definovaný Andrews [14] následovně:

  Klient je spouštěcí proces; server je reaktivní proces. Klient vytváří požadavky, které spouštějí reakce serveru. Tudíž klient si sám vybírá čas pro inicializaci aktivity a čeká na zpracování požadavku. Na druhou stranu server čeká na požadavky a reaguje na ně. Server je obvykle stále běžící proces a často poskytuje služby více klientům.

  Hlavní principem této podmínky je oddělení zodpovědností[5]), které umožňuje zjednodušení serveru a dosažení jeho lepší škálovatelnosti. Důležitým aspektem oddělení zodpovědností je také možnost nezávislého vývoje obou komponent.

[1]) Více o RabbitMQ na `http://www.rabbitmq.com/`
[2]) Více o ZeroMQ na `http://zeromq.org/`
[3]) Více o na `http://martinfowler.com/articles/continuousIntegration.html`
[4]) Více na `http://martinfowler.com/bliki/ContinuousDelivery.html`
[5]) Separation of Concerns

- *Stateless* – další podmínka je bezstavovost:

  Každý požadavek od klienta na server musí obsahovat všechny informace potřebné k jeho porozumění a nesmí spoléhat na žádná uložená data na serveru. Stav musí být vždy výhradně držen na klientu.

  Toto omezení zlepšuje viditelnost – požadavek je vždy jako jednotlivá část, která není závislá na jiných požadavcích nebo stavu na serveru. Neukládání stavu na serveru také zlepšuje škálovatelnost, kdy není třeba tyto data ukládat a synchronizovat.
- *Cache* – podmínka cache požaduje, aby data v odpovědi byla implicitně nebo explicitně označena zda mohou, nebo nesmí být uložena do cache. Pokud odpověď může být uložena do cache, pak ji může klientská cache znovu použít pro další ekvivalentní požadavky a ušetřit požadavek na server. Výhodou cache je potenciální částečné či úplné odstranění nutnosti posílat požadavky na server, což to velmi zvyšuje výkonnost, efektivitu i škálovatelnost. Nicméně přináší nevýhodu v tom, že se snižuje spolehlivost, když se data z cache mohou podstatně lišit od aktuálního stavu dat na serveru.
- *Uniform Interface* – důraz na jednotné rozhraní mezi komponentami označuje Fielding za centrální vlastnost architektonického stylu REST, která jej odlišuje od ostatních síťových stylů.

  Aplikováním principu všeobecnosti[1]) na rozhraní komponent je celá systémová architektura zjednodušena a je zpřehledněna interakce. Implementace jsou odděleny od poskytovaných služeb, což podporuje nezávislý vývoj.

  Na druhou stranu jednotné rozhraní degraduje efektivnost, protože informace je přenášena ve standardizovaném formátu namísto specializovaného formátu pro aktuální použití v aplikaci.

  Detailněji popíši tuto podmínku v následující části 3.3.1.
- *Layered System* – podmínka na systém skládající se z hierarchických vrstev, které spolu komunikují. Každá komponenta ví jen o bezprostředních vrstvách, se kterými komunikuje. Toto ohraničení jen na jednu vrstvu přináší zjednodušení celého systému a podporuje nezávislost jednotlivých komponent, kdy je možné vkládat prostředníky – proxy či gateway, které se chovají transparentně na obě strany.

  Nevýhodou vrstveného systému je přidání větší režie a latence, která může snižovat výkonnost. S přihlédnutím na podmínku cache není dopad tak velký a často se cache může použít i na sdíleném prostředníku.
- *Code on demand* – kód na vyžádání je poslední volitelná podmínka pro REST, která umožňuje klientu rozšíření funkcionality stažením a spuštěním skriptu či appletu. To může zjednodušit klienta tím, že snižuje počet před-implementovaných funkcionalit a umožňuje tím jeho rozšiřitelnost.

3.3.1 Uniform Interface – Jednotné rozhraní

Fielding ve své disertační práci [1] označil jednotné rozhraní jako centrální vlastnost architektonického stylu REST a uvedl následující čtyři podmínky, které jej definují:

- identifikace resources
- manipulace resources přes jejich reprezentace
- sebe-vysvětlující zprávy
- hypermédia jako aplikační stav (*hypermedia as the engine of application state* – HATEOAS)

[1]) Principle of Generality

Dvěma základními koncepty jsou resource a reprezentace. Jejich souvislost spolu s URI je znázorněna na obrázku 3.2.

Resource jsem definoval v sekci 2.2 podle Fieldinga ([1]) jako:

Abstrakce informace, která může být pojmenovaná – dokument, obrázek, dočasná služba (dnešní počasí v Praze), reálný objekt (např. osoba) a další.

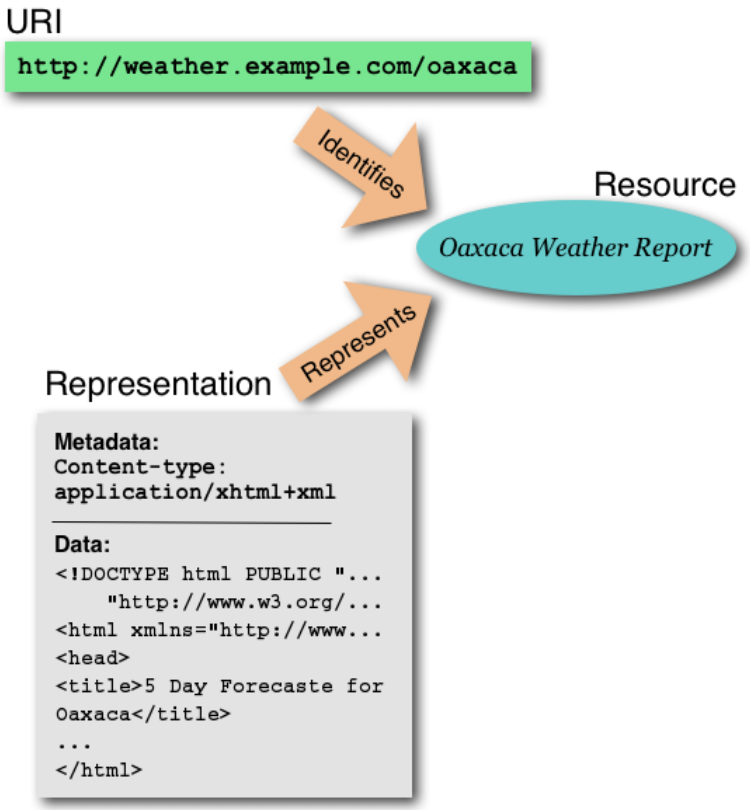
Ve W3C dokumentu *Architecture of the World Wide Web* [15] resource definují za podmínek:

„Pokud klient chce vytvořit odkaz pomocí hyperlinku; vytvářet nebo vyvracet tvrzení o něm; získat nebo uložit do cache reprezentaci nebo zahrnout celý nebo jeho část jako referenci na jinou reprezentaci; či provádět jiné operace.“

Jediné omezení je, že resource musí mít URI. URI[1]) je způsob identifikace resource definovaný v RFC 3986 [16], který může být ještě klasifikovaný na URL[2]) a URN[3]), které jsou podmnožinou URI. URL popisuje u resource jeho lokaci v síti společně s primárním přístupem (protokolem). URN označuje globální a unikátní identifikátor, jeho příkladem je např. ISBN.

Kniha RESTful Web APIs [17][4]) k vysvětlení resource dodává:

Z pohledu klienta není podstatné, co je resource, protože klient se nikdy nesetká s resource, ale vždy jen s jeho reprezentací na dané URI.



Obrázek 3.2. Souvislost mezi resource, URI a reprezentací. URI identifikuje resource, reprezentace resource reprezentuje ve strojově zpracovatelném formátu. Zdroj [15]

Reprezentace je zachycení či vysvětlení aktuálního či jiného stavu resource. Reprezentace se používá pro přenos mezi REST komponentami a může být v jakémkoliv

[1]) Uniform Resource Identification
[2]) Uniform Resource Locator
[3]) Uniform Resource Name
[4]) strana 32.

strojově zpracovatelném formátu a může obsahovat různé informace o resource. Kniha RESTful Web APIs [17] vysvětluje komunikaci komponent následovně:

Server odesílá reprezentace popisující stav resource. Klient posílá reprezentaci popisující stav, v jakém by resource chtěl mít. To je representation state transfer (REST).

Resource může mít více reprezentací. Například resources můžou být v různých jazycích nebo resources mohou mít stručnější a detailnější reprezentaci. Jiná API nabízí data v různých formátech – JSON, XML a další. Existují dva přístupy k těmto různým reprezentacím:

- Content negotiation – klient specifikuje požadovanou reprezentaci (pomocí HTTP hlaviček `Accept`)
- Rozdílné URI pro jednotlivé reprezentace – např. `http://example.com/res.json` nebo jako URI parameter `http://example.com/resource?format=json`

Podmínka **sebe-vysvětlujících zpráv** se při využití HTTP protokolu odráží v jeho sémantice, kterou v následující části krátce popíši s pomocí knihy RESTful Web APIs [17].

Jak bylo definováno dříve, tak resource může být cokoliv. Klient s ním nemůže dělat, co by chtěl, a měl by se držet dané sémantiky protokolu. HTTP standard popsaný v RFC 7231 [18] definuje osm druhů zpráv, HTTP metod:

- GET – získání reprezentace resource
- DELETE – smazání resource
- POST – vytvoření nového resource pomocí dané reprezentace
- PUT – nahrazení stavu stávajícího resource stavem dané reprezentace
- HEAD – získání hlaviček, které by byly poslány k dané reprezentaci resource, ale bez samotné reprezentace
- OPTIONS – zjištění dostupných HTTP metod, na které resource odpovídá
- TRACE a CONNECT – metody využitelné jen pro HTTP proxy. Tyto metody dále nebudu popisovat.

RFC 5789 [19] definuje rozšíření HTTP metod o:

- PATCH – nahrazení částečného stavu resource

GET

GET je definována jako *bezpečná* HTTP metoda, jejíž poslání je dotázání na reprezentaci resource identifikované URI. Poslání GET požadavku na server by mělo mít stejný efekt na stav resource jako v jeho neposlání – tj. žádný efekt. Obvyklá odpověď serveru je 200 (OK), případně 302 (Moved Permanently)

DELETE

DELETE posílá klient v případě, že chce smazat resource. V případě úspěchu jsou obvyklé odpovědi serveru 204 (No content, resource je smazaný a žádná další zpráva), 200 (OK – smazáno a zpráva o tom), 202 (Accepted – OK, bude smazáno později). Pokud klient pošle GET požadavek na smazaný resource, tak server vrátí chybový kód, obvykle 404 (Not Found) nebo 410 (Gone).

DELETE není bezpečná metoda, protože mění stav resource a má jinou vlastnost – *idempotenci*[1]). Jakmile je resource jednou smazaný, tak už bude permanentně smazaný. Pokud se pošle další DELETE požadavek, tak stav resource zůstane stejný. Tato vlastnost je vhodná v případě, kdy z nějakého důvodu selže spojení se serverem a klient nedostane odpověď, pak je možné poslat stejný požadavek znovu a výsledný stav resource bude totožný.

[1]) Definováno v [18] Sekce 4.2.2

POST

RFC 7231 [18] definuje jako funkce metody POST následující:

- Vytváření nových resource, které získají vlastní identifikaci od serveru
- Poskytnutí dat pro jejích následné zpracování ze zdrojů jako jsou např. pole z HTML formulářů
- Posílání zprávy do emailových a diskuzních skupin, na různá fóra, blogy či podobné skupiny článků
- Přidávání dat do existujících reprezentací daného resource

Kniha RESTful Web APIs [17] rozděluje tyto funkce na dvě skupiny – *POST-to-Append* a *Overloaded POST*.

Klient používá POST-to-Append v případě, že chce vytvořit nový resource a jeho reprezentaci posílá v těle požadavku. Pokud server resource vytvoří, měl by být vrácen návratový kód 201 (Created) spolu s hlavičkou Location, která obsahuje URI nového resource. Alternativně kód 202 (Accepted), kdy server oznamuje, že požadavek přijal, ale resource ještě nevytvořil.

POST není bezpečný ani idempotentní, protože pokud pošle 5 identických požadavků, tak bude vytvořeno 5 nových resource.

Druhá skupina pokrývající zbývající funkce je Overloaded POST, neboli přetížený POST. Velkou část webu tvoří HTML, které umožňuje jen GET[1]) a POST[2]). GET je bezpečný a idempotentní, a proto se POST v HTML se používá na vše ostatní včetně ostatních HTTP metod (PUT, DELETE). Chování přetíženého POST může být tedy libovolné a není podrobněji definované.

PUT

Metoda PUT má funkci na změnu stavu resource. Pomocí PUT server nahradí stav resource upravenou reprezentací poslanou v těle požadavku. Server může takovýto požadavek přijmout, nebo zamítnout. V úspěšném případě odpovídá pomocí 200 (OK) s reprezentací v odpovědi, nebo 204 (No content) bez těla odpovědi. Klient také může pomocí PUT vytvořit nový resource v případě, že zná jeho identifikátor, resp. jeho URI. V tom případě by server měl odpovědět obdobně jako u POST – 201 (Created) spolu s hlavičkou Location, která obsahuje odkaz na reprezentaci. PUT je stejně jako GET a DELETE idempotentní.

HEAD

Metoda HEAD je bezpečná metoda podobná GET, jediný rozdíl je v tom, že v případě HEAD není posláno tělo odpovědi. Odpověď tedy obsahuje jen návratový HTTP kód a HTTP hlavičky.

OPTIONS

Metoda OPTIONS je jednoduchý způsob zjištění dostupných HTTP metod, na které resource odpovídá a které podporuje. Server vrací tyto standardní metody v hlavičce *Allow*.

Tato metoda se ale obvykle příliš nepoužívá. Místo ní nabízí aktuální reprezentace dostupné metody a přechody pomocí hypermédií, či jsou případně specifikované v dokumentaci.

PATCH

PUT metoda upravuje vždy stav celé modifikované reprezentace. Pokud chce klient změnit jen malou část této reprezentace, tak to není možné pomocí PUT. Tuto funkci

[1]) přejít na jindou stránku pomocí odkazu přes tag A, zobrazení obrázku přes IMG, script pomocí SCRIPT a další

[2]) odeslání formuláře

ale dovoluje rozšíření HTTP – metoda PATCH definovaná v RFC 5789 [19], která umožňuje poslat jen část reprezentace v rozdílovém tvaru. PATCH není bezpečná ani idempotentní.

3.3.2 Hypermédia

Fielding ve své práci nedefinuje podrobněji termín hypermédia jako aplikační stav (HATEOAS), ale ve svém článku *REST APIs must be hypertext-driven* [20] dodává, že všechny REST API musejí být hypertextové.

Stav je takový, že klient vytváří HTTP požadavky na URI, které identifikují resources a server posílá reprezentace resource, se kterými může klient dále pracovat. Hypermédia následně řeší problémy typu:

- Jak se klient dozví, který dotaz může udělat
- Jak se klient dozví, jakou reprezentaci bude mít resource na daném URI
- Jaký je vztah mezi odkazovanými resource

Hypermédia spojují jednotlivé resource a popisují jejich možnosti ve strojově zpracovatelném formátu. Hypermédia nejsou jedna technologie, ale strategie. Cílem této strategie je umožnit serveru říci klientu, jaké požadavky může v budoucnu dělat a jak mají vypadat. Na klientu je potom, aby si z nabízených možností vybral.

Autoři knihy RESTful Web APIs [17] v části *An API Designer's Guide to the Fielding Dissertation*[1]) uvádějí následující vysvětlení hypermédií:

- Stav aplikace je držen na klientské straně a změny v aplikačním stavu jsou zodpovědností klienta.
- Klient může měnit vlastní aplikační stav jen pomocí požadavků a zpracováním odpovědí.
- Jak se klient dozví, které požadavky může vytvořit? Pomocí hypermédiových odkazů v reprezentacích, které získal v předchozích odpovědích ze serveru.
- Hypermédiové odkazy jsou hlavním prvkem změn v aplikačním stavu.

Hypermédia přinášejí rozšiřitelnost, kdy se chytří klienti mohou automaticky adaptovat na změny v serverové části. Serveru zase umožňují změnu v implementaci bez rozbití funkčnosti všech klientů, např. URI se mohou měnit. Pro klienta to znamená větší komplexitu, kdy si musí umět poradit se zpracováním hypermédii, uchováním aplikačního stavu a rozhodnutí o dalších akcích.

Jednou z hypermédia technologií je HTTP hlavička Link definovaná v RFC 5988 [21], která umožňuje přidat hypermédia i tam, kde to obsah odpovědi nepodporuje – např. text, JSON nebo obrázky. Ukázka z [17]:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Link: ⟨http://www.example.com/story/part2⟩;rel="next"

It was a dark and stormy night. Suddenly, a...
(continued in part 2)
```

Základní myšlenka je identická s prohlížením webu – klient dostane úvodní[2]) URI, ze které následně vedou přechody na dostupné resources. Další technologie spojené s hypermédii zmíním v sekci 3.6, kde se budu věnovat vybraným média typům pro odpovědi serveru.

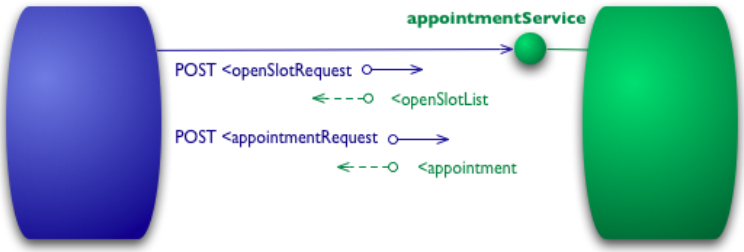
[1]) strana 348
[2]) také uváděné jako domácí nebo billboard URI

3.4 Richardson Maturity Model

Richardson Maturity Model (RMM) je zjednodušeným modelem, který popisuje úrovně implementace REST API dle Fieldinga [1]. Tento model představil Leon Richardson na konferenci QCon v roce 2008 [22] a dále ho budu popisovat s pomocí článku *Richardson Maturity Model: steps toward the glory of REST* [23] od Fowlera.

Model obsahuje čtyři úrovně:

- **Úroveň 0**: Bažina[1])



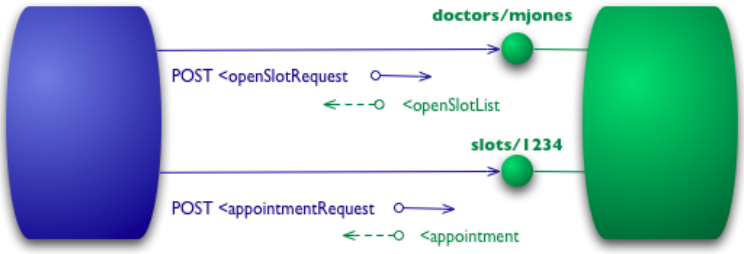
Obrázek 3.3. RMM Úroveň 0: Bažina – klient se serverem komunikuje přes HTTP přes jediný endpoint a posílá všechny požadavky v definovaném formátu (obvykle XML) pomocí POST metody, zdroj [23]

Nultá úroveň je využití protokolu HTTP pro interakce se webovým API. Pro API existuje jediný endpoint s URI, na který se odesílají všechny požadavky v definované formě - např. jako XML zprávy. Jako HTTP metoda je použita POST a návratový HTTP kód API 200. Při chybě v požadavku by server vrátil odpověď opět s HTTP 200 a odpověď bude obsahovat rozdílnou zprávu obsahující popis s chyby.

Jedná se prakticky o stejný mechanismus jako při využití SOAP či XML-RPC, rozdíl je jen v obálce zprávy.

- **Úroveň 1**: Resources

  Úroveň 1 přidává *resource* – všechny požadavky již nevedou na jeden endpoint, ale každý resource má svůj vlastní. Martin Fowler tento stupeň přirovnává k objektovému programování a volání metod na objektu.



Obrázek 3.4. RMM Úroveň 1: Resources – pro každý resource existuje vlastní endpoint, na který jsou odesílány požadavky, zdroj [23]

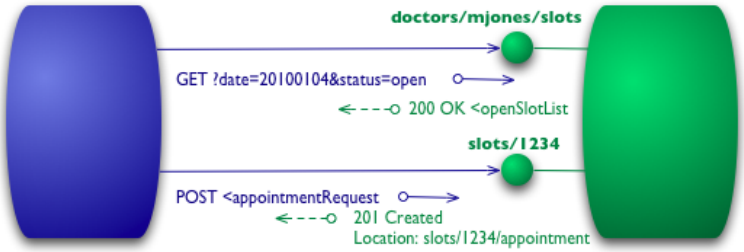
- **Úroveň 2**: HTTP metody

  Úroveň 2 využívá HTTP metod (GET, POST, PUT, ...) při práci s API a vyžaduje jejich sémantiku, popsanou v kapitole Jednotné rozhraní 3.3.1 – GET je bezpečný, POST a PUT mění stav, atd.

  Další podstatnou změnou oproti stupni 1 je využití návratových kódů HTTP jako nositely významu odpovědi.

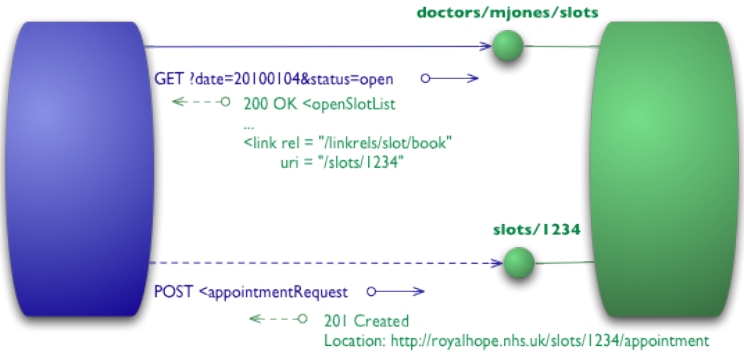
[1]) Fowler tuto úroveň poeticky nazval „The swamp of the POX“ (Plain Old XML)

Pokud je odpověď v pořádku, vrací se odpověď s HTTP kódem skupiny 200. V případě chyby klientského požadavku bude HTTP kód ze skupiny 400, např. 400 Bad Request označující špatný dotaz s detailnější chybou uvedenou v těle odpovědi. V případě chyby na straně serveru se použije HTTP kód skupiny 500.



Obrázek 3.5. RMM Úroveň 2: HTTP metody – pro požadavky na resource jsou využity různé HTTP metody a návratový kód HTTP určuje význam odpovědi (úspěch, chyba klienta, chyba serveru), zdroj [23]

- **Úroveň 3** – Hypermédia
  Poslední úroveň zahrnuje využití principu hypermédií, který popisuje aplikační stav a možné přechody na další stavy a odkazy na související resource.



Obrázek 3.6. RMM Úroveň 3: Hypermedia – pro znázornění stavu aplikace a možných přechodů je využito hypermédií, zdroj [23]

Fowler dále dodává význam jednotlivých úrovní:

- Úroveň 1 snižuje komplexitu pomocí metody „rozděl a panuj“ a zmenšuje jeden velký endpoint na více menších resource.
- Úroveň 2 zavádí standard do použití metod, takže podobné situace jsou řešené jednotně a je odstraněna zbytečná variace.
- Úroveň 3 umožňuje objevitelnost a způsob vytvoření více sebe-dokumentujícího protokolu.

V již zmíněném článku *REST APIs must be hypertext-driven* [20] Fielding zdůrazňuje, že REST API musí poskytovat přechody pomocí hypertextových odkazů (princip HATEOAS) a API, které to nesplňují, se nemohou označovat jako REST. Tudíž dle modelu všechny API s RMM úrovní menší než 3 nejsou REST dle definice Fieldinga.

V praxi ale málokteré REST API splňuje všechna omezení definované pro REST, proto mi tento Richardsonův model přišel jako dobré srovnání implementace REST stylu i návod pro metodiku tvorby REST API.

3.5 Popis a definice API

Pokud existuje API, ať už se jedná o interní nebo veřejné, tak jeho konzumenti potřebují jeho popis a specifikaci chování pro implementaci svých klientů. Pro tuto dokumentaci existuje hodně různých přístupů a formátů, proto v této části budu popisovat jen vybrané formáty, které často souvisejí s REST API a reprezentací v JSON.

Na popis API se dá nahlédnout z několika úrovní, které se dají rozdělit dle abstrakce. Nejvyšší pohled je na samotné aplikace a jejich API bez většího detailu. Tuto abstrakci popisuje APIs.json.

Nižší úroveň je na samotné API, jeho resource a jejich endpointy. Na tomto stupni jsou WADL, API Blueprint, Swagger a RAML.

Nejnižší stupeň je popis reprezentace požadavků a odpovědí, kterou popíši v další části.

3.5.1 APIs.json

Formát APIS.json [12] byl již zmíněn v části 3.1.1. Tento formát se sestává ze souboru pojmenovaného apis.json, který se nachází v kořenu domény tj. `http://example.com/apis.json` a jeho cílem je jednoduchý popis dostupných API na dané doméně a poskytnutí informací o jejich funkčnosti. Formát je aktuálně ve verzi 0.15 [24] a obsahuje popis následujících informací o API:

- Název a popis celé kolekce API na dané doméně
- Datum vytvoření, poslední modifikace a verzi Apis.json souboru
- Udržovatele daných API (osoba nebo organizace) s jmény, kontaktními údaji a dalšími informacemi
- Kolekce popisů jednotlivých API, kdy pro každé API je uvedeno:
  - název a popis API
  - base URI pro API, URI pro dokumentaci
  - verze API
  - dodatečné vlastnosti API – místo pro vytvoření účtu, Swagger, API Blueprint a další definice pro API, odkaz na blog a jiné
  - kontakt na provozovatele a podporu

Tento formát se snaží strojově popsat existenci API na zadané doméně a vystavit veřejně tyto informace pro další navigaci potencionálních klientů. Formát je ještě v rané specifikaci, ale do budoucna se může jednat o způsob, jak automatizovaně publikovat informace o veřejných API.

3.5.2 WADL

Web Application Description Language [25] (dále WADL) je jazyk pro strojově zpracovatelný popis webových aplikací založených na HTTP, který byl vytvořený v roce 2005 ve společnosti Sun. WADL je obdobný formát jako WSDL pro SOAP – na jednom místě je popsána celá definice API.

WADL definuje následující entity:

- Množina resources
- Popis vztahů mezi resources
- HTTP metody aplikovatelné na jednotlivé resources, očekávané vstupy a výstupy s jejich formáty
- Formáty reprezentace daných resources

Hlavními zamýšlenými případy užití WADL jsou:

- Modelování a vizualizace aplikace
- Automatické generování částí kódu klienta
- Konfigurace serveru i klienta pomocí stejného formátu

WADL je definovaný pomocí XML, jeho kořenový element je `application`, který obsahuje kolekci elementů `resources`, který obsahuje atribut s informací o základní URI (base URI). Ukázkový WADL popis je v příloze D.1 Element `resources` následně obsahuje kolekci elementů `resource`, které popisují jednotlivé resources.

Pro každý resource je definována URI jeho endpointu a dále je možné specifikovat elementy `param` a `method` jako potomky. Element `param` popisuje parametry pro dotazy na daný resource. Element `method` popisuje požadavek a odpověď resource na HTTP metody, které daný resource podporuje. Požadavek je popsán v elementu `request`, který obsahuje informace o parametrech a dané reprezentaci. Obdobné je to v elementu `response` specifikujícím odpověď, kde je také definována reprezentace a parametry, které popisují HTTP hlavičky. Reprezentace v elementu `representation` je definována pomocí média typu a profilu. Profil je dokument popisující význam atributů `rel` a `rev` u potomků typu `link`.

Element `link` identifikuje odkazy na resources uvnitř reprezentace resource. Tento element obsahuje atribut `resource_type`, který identifikuje odkazovaný resource, a také atributy `rel` a `rev`, které identifikují vztah resource a reprezentace, kdy `rel` popisuje vztah od definovaného resource a `rev` popisuje obrácený vztah.

Formát WADL je popisem API po jeho návrhu, kdy je jeho chování definováno. Nevýhodou je, že špatně podporuje změny v API jako přidání resource či odkazu, protože pokud je klient vygenerovaný z původního popisu, tak by přestal fungovat, a server tak musí zachovat původní chování a pro změny vytvořit novou verzi WADL i API.

Dalším problémem WADL je, že těsně spojuje popis rozhraní (resource a odkazy mezi nimi) a její implementací (použité HTTP metody a URI). Také možnosti popisu nejsou tak detailní.

3.5.3 API Blueprint

API Blueprint [26] je formát pro dokumentaci webových API od firmy Apiary [27]. API Blueprint popisuje API v syntaxi Markdown [28], které dává speciální význam.

Filozofie Markdown syntaxe je být syntaxí snadnou jak pro čtení, tak i pro psaní. Původní poslání Markdown syntaxe byl převod z prostého textu do HTML, ale API Blueprint přidává zpracování do strojově čitelného JSON formátu. Při tom ale zůstává i výhoda převodu do HTML pro vytvoření HTML dokumentace.

API Blueprint popisuje API pomocí sekcí, které nabývají různých hodnot. Jednou ze sekcí je `Resource`, která popisují daný resource pomocí URI šablony nebo resource pojmenovává. Další podstatnou sekcí je `Action`, která popisuje akce pro daný resource a pomocí ní lze specifikovat podporované HTTP metody a jejich ukázkové požadavky a odpovědi. Pro požadavky je možno specifikovat jejich média typ, požadované HTTP hlavičky a ukázkový obsah (payload), pro odpovědi je možné vyjmenovat návratové kódy, HTTP hlavičky, média typ a obsah odpovědi. Pro odpovědi ve formátu JSON lze také specifikovat typy pro jednotlivé hodnoty - daný klíč bude číslo, řetězec, objekt či pole. Detailněji popisuje formát jeho specifikace [29].

API Blueprint podporuje také abstrakci resource v podobě `resource modelu`, který definuje na jednom místě daný resource a následně se může znovu používat ve íce sekcích a nemusí se definovat zvlášť.

API Blueprint využívají další nástroje, které dodávají výhody tomuto formátu. Hlavním nástrojem je celá platforma Apiary [27], která umožňuje vytvářet API Blueprint pomocí editoru a generovat dokumentaci. Nabízí vygenerování kódu volání daných resource pro různé programovací jazyky a také vytváří automaticky mock server, který odpovídá na dotazy dle specifikovaného formátu a ukládá dotazy na tento mock pro jednodušší testování klientů.

Dalším zajímavým nástrojem je Dredd [30], který pro zadaný API Blueprint provede validaci oproti aktuální implementaci na serveru. Tak je možné ověřit aktuální dokumentaci, zda odpovídají implementaci.

Mezi další nástroje patří Drakov [31], který umožňuje vytvořit mock server, API-MATIC [32], který generuje kód pro různé programovací jazyky, a další nástroje, které formát API Blueprint propojují s jinými nástroji.

3.5.4 Swagger

Swagger [33] je formát a framework pro popis, vytváření, konzumování a vizualizaci RESTful API. Specifikace Swaggeru [34] definuje deklarativní reprezentaci REST API, podle které lze pomocí doplňujících nástrojů jednoduše vytvořit interaktivní dokumentaci či vygenerovat klienta pro různé programovací jazyky.

Specifikace nevnucuje způsob vytvoření REST API, ale jen se ho snaží popsat a tak Swagger nijak nepožaduje přepsání či modifikaci existujícího API a umožňuje i popsat úplně cizí API.

Pro vytvoření Swagger specifikace jsou tři způsoby:

- Vygenerování ze zdrojového kódu serveru – nejčastěji pomocí anotací. Generování může být statické (při buildu) nebo dynamické (při požadavku)
- Automaticky – Server swagger-node-express [35] vytvoří jak REST API, tak i Swagger specifikaci
- Manuálně – specifikaci je také možné napsat manuálně či pomocí Swagger Editoru[1]) a následně z ní vygenerovat kód pro server (např. pomocí generátoru serveru pro node.js [36])

Primární formát pro Swagger je JSON, ale ve Swagger Editoru je možné zapisovat formát i v YAML. Samotný formát obsahuje podobné položky jako u API Blueprintu:

- metadata v podobě `info`, které popisují API – název, popis, kontakt, licence, verze API a podmínky použití
- informace o umístění API a použitých protokolech – `host`, `basePath`, `schema` (zda je API na HTTP(S))
- volitelné informace o použitých média typech v elementech `consumes`, `produces`
- informace o autentizaci v elementu `security`, kde je možné popsat použití API klíčů, Basic Autentizace nebo OAuth2
- popis jednotlivých endpointů v podobě `paths` elementů, které popisují jednotlivé URI a na nich dostupné operace – HTTP metody
- pro každou metodu je možné dále specifikovat parametry pro požadavky a jejich typy a hodnoty; povinným elementem je `response` obsahující informace o možných odpovědích serveru včetně jejich návratové kódu a jeho struktuře
- specifikace [34] dále obsahuje další pokročilejší funkce pro popis API, snadnější zápis opakovaných dat a další zjednodušení

[1]) `http://editor.swagger.io/`

Swagger UI [37] je nástroj, který ze Swagger specifikace vytvoří dokumentaci REST API spolu s interaktivním klientem, pomocí kterého je možné testovat API přímo z prohlížeče.

Swagger Code Generator [38] umožňuje ze specifikace vygenerovat kód pro klienta v různých jazycích (Java, PHP, Python, Ruby, Scala atd.).

3.5.5 RESTful API Modeling Language

RESTful API Modeling Language (RAML) [39] je jednoduchý a stručný způsob popisu RESTful API.

RAML formát je neproprietární a otevřená specifikace založená na kombinaci YAML a JSON. V době psaní této práce (květen 2015) se nachází specifikace ve verzi 0.8 a je připravována verze 1.0 [40]. RAML má poměrně silné podporovatele – firma Mulesoft, Miško Hevery tvůrce AngularJS, Jason Harmon z PayPalu či John Musser zakladatel Programmable Web.

Formát RAML obdobně jako předchozí formáty obsahuje možnost popsat API:

- meta informace o názvu API, base URI, verzi, podporované protokoly a použité média typy a schémata
- popis jednotlivých endpointů a dostupných HTTP metod
- popis jednotlivých požadavků, odpovědí včetně jejich reprezentací, HTTP hlaviček a HTTP status kódů
- doplňující textová dokumentace může být napsána v Markdownu
- pro snadnější zápis opakujících se dat podporuje RAML traity, resource typy a security schema, více ve specifikaci [41]

Obdobně jako Swagger, tak i RAML obsahuje ekosystém nástrojů, které využívají definovaného formátu. Základem těchto nástrojů je parser dostupný pro JavaScript [42], Javu [43], Python [44] či Ruby [45].

Pro návrh API existuje nástroj RAML Editor [46], který umožňuje interaktivní vytvoření RAML v editoru. Uživatelsky příjemná vlastnost je i kontextová nápověda možné syntaxe. Doplňujícím nástrojem k editoru je API console [47], která zobrazuje vygenerovanou dokumentaci a je možné přímo z prohlížeče testovat volání API.

Mezi další nástroje patří API Notebook [48], který spojuje text se spustitelným kódem a je velmi vhodný pro rychlé prototypování či vysvětlení API pomocí interaktivního tutoriálu.

3.5.6 Srovnání API Blueprint, Swagger a RAML

Všechny tyto formáty umějí svým způsobem popsat a definovat nejdůležitější části API:

- Definice resource
- Podporované metody a akce
- Popis URL včetně parametrů v cestě i v query stringu
- Popis hlaviček
- Popis možných požadavků i odpovědí s jejich HTTP status kódy a popisem reprezentací (média typ)
- Dokumentace popisující význam a vysvětlení daných resources a reálných akcí při využití API

Pro srovnání všech tří formátů vyšlo několik článků a prezentací, např. Kin Lane *API Design: Do You Swagger, Blueprint or RAML?* [49], Mike Stove *RAML vs. Swagger vs. API Blueprint* [50], prezentace Ole Lensmara [51], Orlando Kalossakas *Which API editor to use?* [52] a další.

Propagovanou odlišností je využití různých formátů pro psaní – API Blueprint používá Markdown, Swagger JSON/YAML a RAML je založený na YAML. Pro každý formát také existují podpůrné nástroje – např. editory pro psaní formátu, parsery, nástroje pro kolaboraci, generování klientských i serverových knihoven, sdílení, vytváření mock serverů, nástroje pro podporu testování a další podpůrné knihovny pro různé programovací jazyky. Největší zastoupení má Swagger, který je na trhu také nejdéle. API Blueprint a RAML ale mají výhodu v SaaS[1]) platformách Apiary [27], resp. Mulesoft Anypoint [53], které nabízejí možnost využití funkčních nástrojů místo jejich zprovozňování na vlastní infrastruktuře.

Formáty a nástroje mají své přednosti, průběžně se vylepšují a jejich konkurence stále nutí k dalšímu vývoji. Vybrání optimálního nástroje pak souvisí s detailnějšími požadavky či osobní preferenci a zkušenostech.

3.6 Formát reprezentace požadavku a odpovědi

V této části popíši několik formátů, které se dají použít pro obsah zprávy požadavku nebo odpovědi. Takových formátů existuje velké množství, kdy standardizované jsou specifikovány v adresáři organizace IANA[2]), ale existují i různé návrhy formátů ve fázi vývoje. Zmíním formáty, které souvisejí s REST API a jsou založené na JSON, spolu s formáty podporujícími hypermédia.

Samotný JSON, resp. média typ `application/json`, je definován (IANA[3]), RFC [54]) jen jako soubor pravidel, jak lze v JSON data a strukturu zapsat. Nijak ale nedefinuje sémantiku struktury jednotlivých elementů. To doplňují specifičtější média typy.

Tvůrce API může použít samostatný JSON, a vytvořit tak vlastní specifikaci a dokumentaci. Výhoda použití již definovaného média typu je v znovuvyužití formátu, který může mít lépe vyřešeny možné problémy a také má již hotovou specifikaci. Pro tyto specifikované média typy můžou existovat různé knihovny, které ušetří práci i pro konzumenta. Nevýhody jsou ale v tom, že ne vždy definované média typy odpovídají přímo požadované problémové doméně.

3.6.1 JSON Home

JSON Home [55] je návrh pro speciální formát „domácího“ dokumentu pro HTTP klienty, kteří nejsou prohlížeč (tj. především pro API). Tento formát se snaží podpořit volnější vazbu mezi klientem a serverem než je např. u WADL pomocí principu hypermédií – API nabízí dostupné resources s dalšími přechody a klient se za běhu může rozhodnout, který přechod bude následovat. JSON Home používá média typ `application/json-home` a je definovaný pomocí JSON. Příklad formátu je znázorněn v příloze D.2.

Formát se skládá z objektu `resources`, který obsahuje jednotlivé vlastnosti, které jsou identifikovány typem odkazu a jejich hodnota je opět JSON objekt, který definuje detaily o daném resource objektu.

V detailech resource objektu jsou následující informace:

- URI k resource nebo URI šablona pomocí klíčů `href`, resp. `href-template`
- Resource Hints (nápovědy) – nápovědy klientu jak může s danými resource provádět interakci

[1]) Software as a Service – Software jako služba
[2]) `http://www.iana.org/assignments/media-types/`
[3]) `http://www.iana.org/assignments/media-types/application/json`

- `allow` – seznam HTTP metod, které je možné použít na resource
- `docs` – odkaz na dokumentaci
- `accept-post`, `accept-patch`, `accept-prefer`, `accept-ranges` – nápovědy k HTTP hlavičkám
- `precondition-req` a `auth-req` – označují předpoklady, resp. použitou autentizaci pro dotazy na daný resource
- `status` – stav resource s dvěma hodnotami – `deprecated` pro zastaralé resource a `gone` pro již nedostupné resource

Formát je stále ve vývoji, ale poslední verze 03 byla vydána v roce 2013 a od té doby se nejspíš nevyvíjí, přestože vzniklo několik ukázkových implementací v různých jazycích pro konzumaci a vytváření JSON Home. Za důvod považuji stále malý počet API využívajících hypermédií, které neřeší vytvoření „domácí“ stránky. Pro domácí stránku lze použít i jiné obecnější hypermédiové formáty (ale nemusí mít všechny schopnosti jako JSON Home), tak není jasné, zda-li má tento speciální formát své využití.

Následující ukázka zobrazuje „domácí“ stránku obsahující popis dvou resources – `http://example.org/rel/widgets` a `http://example.org/rel/widget`. Pro první z nich poskytuje jen informaci o URI, o druhém způsob vytvoření URI podle šablony a napovídá použitelné metody (GET, PUT, DELETE a PATCH) spolu s očekávaným média typem a dalšími HTTP hlavičkami.

```
{
  "resources": {
    "http://example.org/rel/widgets": {
      "href": "/widgets/"
    },
    "http://example.org/rel/widget": {
      "href-template": "/widgets/{widget_id}",
      "href-vars": {
        "widget_id": "http://example.org/param/widget"
      },
      "hints": {
        "allow": ["GET", "PUT", "DELETE", "PATCH"],
        "formats": {
          "application/json": {}
        },
        "accept-patch": ["application/json-patch"],
        "accept-post": ["application/xml"],
        "accept-ranges": ["bytes"]
      }
    }
  }
}
```

3.6.2 JSend

JSend [56] je jednoduchá specifikace pro odpovědi serveru ve formátu JSON. Ukázková odpověď je následující:

```
{
  status : "success",
  data : {
    "post" : {
```

```
      "id" : 1,
      "title" : "A blog post",
      "body" : "Some useful content"
    }
  }
}
```

V ukázce kódu jsou dva hlavní klíče *status*, obsahující úspěšný výsledek (`success`), a *data* obsahující samotná data. JSend definuje tři typy odpovědi – `success`, `fail`, `error`, které mají definované povinné a nepovinné klíče zobrazené v tabulce 3.1.

Typ	Popis	Povinné klíče	Volitelné klíče
success	Všechno v pořádku	status, data	
fail	Chyba v odeslaných datech	status, data	
error	Chyba při zpracování dotazu	status, message	code, data

Tabulka 3.1. JSend – klíče pro různé typy odpovědi. Tabulka je převzata z [56]

V tomto formátu není specifikováno využití HTTP status kódů, což znamená, že ho dle Richardson Maturity Model definovaného v 3.4 nelze zařadit výše než na úroveň 1[1]). Autoři JSend toto odůvodňují tím, že existuje mnoho interpretací návratových kódů a chtěli pro vlastní specifikaci je zjednodušit jen na tři. Dalším argumentem je možnost, že za určitých podmínek (např. JavaScript kód v prohlížeči) nemusí být návratový kód při zpracování odpovědi k dispozici. Na serverové straně ale doporučují doplnit JSend formát i s nejvíc korespondujícím návratovým kódem pro danou odpověď.

Pro specifikaci existují knihovny pro JavaScript [57], Python [58] a Ruby [59].

3.6.3 Problem Details for HTTP APIs

JSend formát navrhoval způsob, jak bude konzumentovi vrácena chyba jako jedna z možností odpovědi. Formát *Problem Details for HTTP APIs* (dále HTTP problem) [60] se přímo specializuje jen na případy chyby a jeho cílem je sjednocení používaných odpovědí. Chyba je při použití REST API identifikovat návratovým HTTP kódem (4XX a 5XX), který často nemusí být dostatečně konkrétní, a detailnější popis chyby je nutný poslat v těle odpovědi. Tento detailnější formát může také poskytovat popis chyby pro člověka využívající API, čímž a tímto se snaží vyhovět Fieldingově podmínce sebe-vysvětlujících zpráv.

Následující ukázka formátu (také v příloze D.3) obsahuje elementy:

```
HTTP/1.1 403 Forbidden
Content-Type: application/problem+json
Content-Language: en

{
 "type": "http://example.com/probs/out-of-credit",
 "title": "You do not have enough credit.",
 "detail": "Your current balance is 30, but that costs 50.",
 "instance": "http://example.net/account/12345/msgs/abc",
 "balance": 30,
 "accounts": ["http://example.net/account/12345",
              "http://example.net/account/67890"]
}
```

[1]) Specifikace se nijak nezmiňuje o URI či využití zdrojů a proto může být na úrovni 0 či 1. Pro úroveň 2 je vyžadováno využití HTTP status kódů

- `type` – absolutní URI identifikující chybu, která obsahuje dokumentaci chyby a její vysvětlení
- `title` – titulek pro danou chybu
- `status` – HTTP status kód
- `detail` – vysvětlení chyby
- `instance` – absolutní URI identifikující aktuální instanci chyby

Tento formát se snaží o sjednocení formátu pro chybu, ale nemá žádnou specifikaci pro nechybové odpovědi. To přináší nutnost použití dvou různých média typ, kdy některé dále zmíněné formáty speficikují oba případy. Výhodu si tento formát najde ale u obecných řešení, které nepoužívají žádný specifičtější formát pro nechybové odpovědi. Formát již podporuje hypermédia ve dvou speciálních případech – popis chyby a instance chyby (`type` a `instance` elementy)

3.6.4 Hypertext Application Language

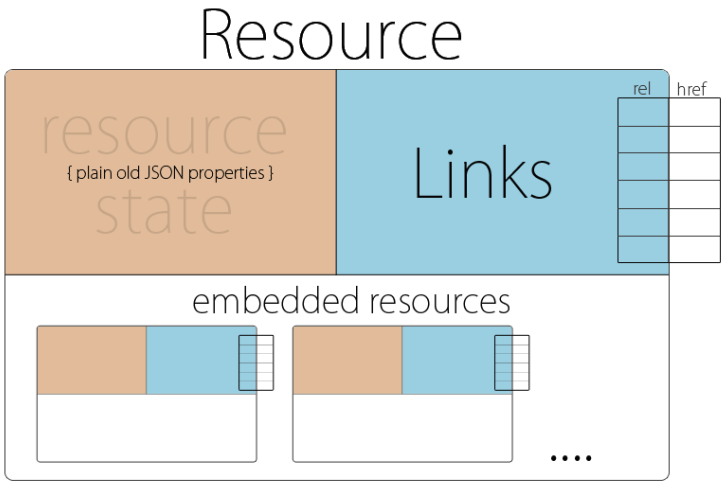
Hypertext Application Language (dále HAL) [61] je formát nabízející způsob provázání resources pomocí hyperlinků v API a je implementací hypermédií.

Cílem formátu navrženého Mikem Kelly bylo vytvořit jednoduše prozkoumatelné API obsahující vlastní dokumentaci přímo uvnitř dokumentu, s hlavním přínosem, kterým je zjednodušením práce pro konzumenty.

HAL je specifikace pro definování hyperlinků pro JSON či XML, které doplňují zbytek dokumentu popisující samotný resource, a má vlastní média typ pro obě varianty (JSON i XML): `application/hal+json` a `application/hal+xml`. HAL model obsahuje reprezentaci dvou základních konceptů – *resources* a *odkazy*[1]), graficky znázorněné na obrázku 3.7.

- *Resources* samotný stav a obsahují odkazy na URI, další rekurzivně vložené (embedded) resources.
- *Odkazy* mají svůj cíl (URI), vztah (`rel`, název pro odkaz) a případně doplňující vlastnosti pro content negotiation, verzování a další.

Podrobnější struktura modelu je popsána na stránce specifikace [61].



[1]) Links

Obrázek 3.7. Hypertext Application Language model znázorňující doplnění odkazů (linků) do stávajícího dokumentu se stavem resource a další rekurzivně vložené resource, zdroj [61].

Využití HAL v API je hlavně v doplnění odkazů, které pomůžou konzumentovi mít přehled o možných akcích a vztazích aktuálních resourců. Tyto odkazy mají výhodu pro vývojáře, že jsou skutečnými URL, které je možné navštívit pro další dokumentaci. Tímto je docíleno efektu, že samotné API je samodokumentující a je možné přes odkazy prozkoumat možnosti API.

Nevýhodou HAL je, že nedefinuje pro odkazy jejich sémantiku – jakou metodu (GET, POST, PUT, ..) je možné na daný odkaz použít. Tato informace, která by mohla být v tomto strojovém formátu, musí být definována v externí dokumentaci.

Pro praktické využití existuje velké množství knihoven pro různé jazyky, které zjednodušují vytvoření API podporující HAL formát. Jejich seznam je odkazovaný na Githubu specifikace [62]. Výhodou HAL je také snadná možnost jeho doplnění do již stávajících odpovědí ve formátu JSON.

3.6.5 Siren

Siren [63] je hypermédia formát pro reprezentování entit používajících média typ `vnd.siren+json`. Siren formát popisuje informace o entitách, akce pro stavové přechody a odkazy pro navigaci klientů. Ukázka formátu je v příloze D.4.

Entita je resource adresovaný URI, který obsahuje vlastnosti (properties) a akce. Může také obsahovat vnořené entity a navigační odkazy. Vlastnosti popisují danou entitu a jsou to její data, která mohou být libovolně strukturovaná. Akce popisují možné chování, které je umožněno danou entitou – obsahuje informace o názvu akce, URI, HTTP metodě (GET, POST, PUT, ...) a volitelných polích popisujících následující požadavek. To poskytuje možnost zkonstruovat dotazy dynamicky, podle dostupných dat v odpovědi.

Siren je podobný formát jako HAL, ale výhodu má v akcích, které mohou lépe popsat metody jiné než GET.

3.6.6 JSON API

JSON API vznikl z JSON formátu používaném v Ember.js Data REST adaptéru[1]) a jeho hlavními autory jsou Steve Klabnik a Yehuda Katz.

Hlavním cílem formátu je vybalancovat následující:

- Generický media typ, který bude fungovat na širokém spektru případů užití včetně různých typu vztahů
- Podobnost s existující praktikou implementace na serveru a čitelností pro vývojáře (pro debugging)
- Jednoduchá implementace serverové části
- Jednoduchá implementace klientské části

JSON API je specifikace, jak by měl klient vytvářet požadavky pro získání či modifikaci resource, a jak by měl server odpovídat. Formát je navržen pro minimalizaci požadavků a počtu dat přenesených mezi klientem a serverem, kdy tato efektivnost je dosažena bez ohrožení čitelnosti, flexibility a objevitelnosti.

JSON API formát používá média typ `application/vnd.api+json`, který musí klient odesílat v hlavičce `Accept` a jehož přítomnost server musí kontrolovat a odpovídat vždy s hlavičkou `Content-Type` nastavenou na stejný média typ.

[1]) `http://emberjs.com/guides/models/the-rest-adapter/`

Samotný dokument obsahuje stejnou strukturu pro požadavky i odpovědi:

- Kořenový element je vždy alespoň jeden z elementů `data` (obsahující data aplikace), `errors` (zobrazení chyby v odpovědi), `meta` (meta informace o API)
- Element `data` obsahuje data aplikace ve tvaru objektu, nebo jako pole více objektů (pro kolekce). Každý data objekt obsahuje identifikaci (`id` a `type`) a dále může obsahovat atributy `attributes`, vztahy daného resource s jinými resources v elementu `relationships` a odkazy související s resources v elementu `links`.
- JSON API poskytuje také možnost skládání více resources do jedné odpovědi tzv. *compound documents*, kdy jsou tyto objekty přidány do elementu `included`. Tato metoda ušetří počet HTTP požadavků klienta, ale není tak efektivní při použití cache.

Specifikace dále upřesňuje komunikaci klienta a serveru při získávání (GET), úpravě (PATCH), vytváření (POST) a mazání (DELETE) resources, spolu s očekávanými status kódy (dle sémantiky HTTP v RFC 7231 [18]) a strukturou odpovědi. Dále jsou také definovány i méně časté operace s resource, jako vkládání klientem požadovaných resource do odpovědi přes URI parametr `include`, vypsání specifikovaných atributů (parametr `fields`), podpora stránkování při výpise kolekcí objektů, filtrování a řazení.

JSON API také obsahuje již zmíněnou podporu elementu pro chyby (`errors`), která definuje obsah odpovědi. Tato odpověď obsahuje informace o identifikátoru chyby, lidský popis a detail, status kód a zdroj chyby (odkaz na specifický atribut či parametr). Jedná se o podobné položky jako u již zmíněného *HTTP error* v části 3.6.3.

JSON API také kromě formátu obsahuje sadu doporučení především pro návrh URI:

- URI pro kolekce objektů dle názvu typu: `/photos`
- URI pro jednotlivé resource se tvoří připojením unikátního identifikátoru k URI kolekce: `/photos/1`
- URI pro vztahy, které umožňuje manipulovat klientu se samotným vztahem (např. odstranění autora k článku), doporučuje připojení cesty `/relationships/` a následně názvu vztahu, např. `/articles/1/relationships/author`
- URI pro související resource jen s doplněním názvu resouce: `/photos/1/comments` pro komentáře k fotce ID 1

Dalším doporučením pro URI je mít při filtrování následující tvar:

```
GET /comments?filter[post]=1,2&filter[author]=12
```

Pro třídění je použito klíčové slovo `filter` jako pole v URI, kdy klíč je název domény pro třídění. Klíč a hodnota jsou čárkou oddělené argumenty.

JSON API nepodporuje přímo hypermédiové šablony, kdy je možné zjistit, jak vytvářet další dotazy, a doporučuje HTTP metodu OPTIONS pro zjištění možných operací na resource.

3.7 Identifikace konzumentů API

Když producent poskytuje API, tak je vždy vhodné vědět, kdo API konzumuje a jaké vytváří dotazy. U veřejných API je vhodné sledovat počet dotazů od jednotlivých konzumentů, jestli některý z nich nepřetěžuje API. U interních API je podle opačného počtu požadavků možné sledovat, zda aplikace konzumenta správně funguje a zda posílá předpokládané množsví dotazů. Dalším důvodem může být analýza chyb konzumentů posílajících špatné dotazy způsobující chyby.

V knize APIs: A Strategy Guide [6][1]) se zmiňují základní tři přístupy:

[1]) strana 75

- Identifikace – kdo posílá požadavek na API
- Autentizace – ověření identity konzumenta API
- Autorizace – umožnění přístupu autentizovaného konzumenta k danému resource – součást implementace API, kterou nebudu dále popisovat

3.7.1 Identifikace

Nejjednodušší formou je identifikace, kdy se konzument označí v požadavku nějakým svým identifikátorem, který ho unikátně identifikuje mezi ostatními konzumenty. Nejčastěji se tento identifikátor označuje jako API klíč[1]), který konzumentovi přidělí producent API. Může se jednat o náhodný řetězec, ale třeba i název konzumenta. Konzument následně vloží tento API klíč do každého svého požadavku na API.

Pro přenos API klíče v požadavku se používají především dva způsoby – parametr v URL nebo HTTP hlavička. Výhodou parametru v URL je, že je možné přistoupit k API jednoduše pomocí prohlížeče.

Nevýhodou API klíčů je, že se nijak dále neověřuje, kdo API klíč poslal. Pokud by někdo získal a použil cizí API klíč, tak ho producent bude považovat za jiného konzumenta.

Některé API používají způsob autentizace, kdy se před voláním API vytvoří dočasný token, který se využívá v dalších požadavcích jako API klíč. Tento přístup jde ale proti REST principu bezestavosti (zmíněno v kapitole 3.3) a zhoršuje škálovatelnost (server si musí držet vygenerovaný token) i následnou identifikaci problémů (token se musí přiřadit k původním údajům).

3.7.2 Autentizace

Autentizace přidává způsob potvrzení, že konzument je skutečně tím, za koho se vydává.

Nejjednodušší způsob je přidělením uživatelského jména a hesla (které se alternativně mohou jmenovat API klíč a API secret). Pro přenos uživatelského jména a hesla přes HTTP existují dva standardy definované v RFC 2617 [64] – **HTTP Basic** a **HTTP Digest**.

Oba způsoby posílají dvojici uživatelské jméno a heslo přímo v HTTP hlavičce `Authorization`: HTTP Basic heslo jen zakóduje pomocí Base64 a HTTP Digest posíla hash hesla pomocí MD5. Ani jeden způsob není bezpečný bez použití zabezpečeného spojení pomocí HTTPS. Dále je také nutné bezpečně ukládat heslo jak na serveru, tak i na klientu.

HMAC (Keyed-hash Message Authentication Code, definován v RFC 2104 [65]) je autentizační kód zprávy vytvořený pomocí hašovací funkce v kombinaci s tajným šifrovacím klíčem. HMAC zajišťuje integritu zprávy a autentizaci odesílatele i přes nezabezpečený kanál (HTTP). Klient stále musí poslat nějakou formu své identifikace (API klíč), aby server poznal, kdo posílá požadavek. Klient vždy každý svůj požadavek podepíše tak, že připojí parametr obsahující hash celé url společně s tajným klíčem. Server tento hash ověří stejným způsobem a klient je autentizován.

OAuth [66] je otevřený protokol umožňující bezpečnou a jednoduchou autorizaci pro webové, mobilní a desktopové aplikace. OAuth vznikl především z důvodu potřeby autentizace pro aplikace třetích stran, kdy zadávání použití jednotných uživatelských údajů je nevhodné nehledě na bezpečnostní riziko. Existují dvě verze OAuth – verze 1.0 definovaná v RFC 5849 [67] a verze 2.0 v RFC 6749 [68]. Hlavní myšlenkou je dát každému klientu individuální pověření, které lze jednotlivě revokovat v případě

[1]) API key

problému s některým klientem. OAuth 1.0 je navržený především pro integraci, kdy se používá webový prohlížeč, ale nefunguje dobře pro mobilní či desktopové aplikace, které lépe podporuje novější verze OAuth 2.0.

Při použití OAuth 1.0 získá bezpečnou cestou[1]) klient od serveru *token* a sdílené tajné heslo[2]). Následně při každém dalším požadavku klienta na server je posílána hlavička `Authorization` obsahující OAuth parametry – *consumer key*, *token*, *signature*, který podepisuje celý požadavek, a tak autentizuje daného uživatele; *timestamp* a *nonce* pro ochranu proti „replay útokům“[3]). OAuth 2.0 je nový protokol přidávající lepší podporu pro aplikace, které nejsou přímo založené na prohlížeči, a snaží se řešit další problém OAuth 1.0. Detailnější popis obsahuje RFC 6749 [68].

Detailnějšímu popisu bezpečnosti API a dalším způsobům autentizace se věnuje kniha *Advanced API Security* [69].

Tabulky 3.2 a 3.3 porovnávají způsoby autentizace, resp. doporučují jednotlivé způsoby p

Technologie	Nutné HTTPS	Stejné heslo pro 3. strany	Náročnost implementace
HTTP Basic	Ano	Ano	Nízká
HTTP Digest	Ano	Ano	Nízká
HMAC	Ne	Záleží na implementaci	Střední
OAuth 1.0	Ne	Ne	Vyšší
OAuth 2.0	Ano	Ne	Vyšší

Tabulka 3.2. Autentizace – porovnání technologií. Sloupec *Nutné HTTPS* obsahuje informaci, zda je nutné využít bezpečného kanálu HTTPS pro posílání požadavků obsahující získanou autentizaci, u OAuth 1.0 se nevztahuje na získání autentizačních tokenů. Položka *Stejné heslo pro 3. strany* obsahuje informaci, zda aplikace třetích stran budou používat sdílené heslo nebo budou používat vlastní token.

Technologie	Vhodné pro
HTTP Basic	Interní API – jednoduché na implementaci, podpora i v prohlížečích
HTTP Digest	Interní API – lehce složitější implementace než Basic, která ale nemá lepší zabezpečení
HMAC	Bezpečné pro jednoduché použití u jak veřejného API, tak i interního API
OAuth 1.0	Veřejné API poskytující data webovým aplikacím třetích stran
OAuth 2.0	Veřejné API poskytující data aplikacím třetích stran (i mobilním a desktopovým)

Tabulka 3.3. Autentizace – využití technologií pro API

3.8 Životní cyklus API

Životní cyklus API se skládá z několika kroků od prvotní myšlenky o vytvoření API až po jeho vypnutí. Každý z kroků má svá specifika a rozhodnutí, které mohou ovlivnit další kroky. Při úspěšnosti API se často iteruje mezi kroky *Návrh a implementace* a *Nasazení a provoz*, když se API průběžně vylepšuje.

[1]) proces získání údajů je popsaný v RFC 5849 [67]
[2]) shared-secret
[3]) opakování stejného odchyceného požadavku

3.8.1 Plán a strategie

Rozhodnutí o vytvoření nového API je obvykle stimulováno potřebou zveřejnění některých dat či funkcí. API by pak mělo být bráno jako produkt s jasným vlastníkem a strategií, co bude API poskytovat. Pokud je API interní, tak jsou obvykle známi jeho konzumenti, a je vhodné se seznámit s jejich případy využití. U veřejných API by měl být jasný byznys plán přínosu API či způsob monetizace.

3.8.2 Návrh a implementace API

Při návrhu API se dělá několik důležitých rozhodnutí, které následně ovlivní jeho implementaci, nasazení, konzumování i budoucí úpravy a vývoj.

Identifikace

Jedním z rozhodnutí je, jak se budou konzumenti identifikovat – bude potřeba identifikace či autentizace? Jaký způsob se využije a který nejlépe sedí předpokládaným konzumentům. Pro interní API je obvykle dostačující jen identifikace, nebo případně Basic autentizace, pro veřejné API se využívá OAuth. Změna způsobu bývá nekompatibilní, a tudíž problematickou změnou pro všechny konzumenty.

Verzování

Další důležitým rozhodnutím je způsob verzování API. Pokud se na verzování nemyslí od začátku, vede to obvykle k problémům při nekompatibilní změně a nutnosti vydat novou verzi API. Následně je nutné donutit stávající konzumenty k přechodu na novou verzi ve stejný okamžik, čímž se zvyšuje komplexita na obou stranách. Konzumenti musí upravovat svojí část, producent musí po nějakou dobu držet více verzí svého API.

Hlavním důvodem pro verzování jsou nekompatibilní změny, dle sémantického verzování[1]) změny *major* verze. *Minor* a *patch* verze se obvykle u API nepoužívají, protože přidání nové kompatibilní funkce nebo oprava chyby nemají vliv na původní kontrakt. Jejich případné použití může být informativní v HTTP hlavičce odpovědi.

Pro tradiční verzování API pomocí *major* verze existují následující přístupy:

- Verzování v URI – nejpoužívanější řešení, které vkládá číslo verze do URI `http://example.com/api/v1/product`, případně jako parametr. Tento přístup ale vytváří duplicitní URI pro stejný resource (reprezentace se liší verzí)
- Verzování ve vlastní HTTP hlavičce – např. `api-version: 2`. Tento způsob má nevýhodu v tom, že nelze API jednoduše použít z prohlížeče a nová hlavička není úplně správně podle sémantiky HTTP protokolu pro toto využití
- Verzování v `Accept` hlavičce pomocí *content negotiation* – např. `Accept: application/vnd.example.v2+json`. Nevýhodou této varianty může být zmatení klienta tím, zda se verzování týká daného resource, či daného média typu.

V knize *APIs: A strategy guide* [6][2]) je také popsána strategie API bez verzování, která ale vyžaduje následující omezení:

- při přidávání funkcionalit není potřeba zvyšovat verzi, a proto je dobré se držet principu „lepší neúplný než nepřesný“.
- návrh URI a formátů pro odpověď by měla co nejvíce generická – např. místo elementu `home_phone` lze používat `phone` s atributem `type='home'`
- důležité je znát vlastní konzumenty a jejich potřeby – to je jednodušší u interních API, kde jsou konzumenti známí a obvykle dobře dostupní

[1]) `http://semver.org/`
[2]) Strana 81.

- zachování jedné verze API může vést k omezení funkcionalit

Pokud se ale nějakým způsobem verzuje, tak je nutné vytvořit proces, jak se budou vypínat zastaralé verze spolu s procesem, jak se konzumenti dozví, že používají zastaralou verzi, a do kdy musí přejít na novější.

Pokud je využito v API hypermédií, tak se některé problémy nemusí řešit verzováním. Např. URI jsou obvykle v tomto případě implementační detail – pokud klient využívá čistě přechodů pomocí hypermédiových šablon. Pokud je využit formát popisující i způsob vytvoření požadavku včetně metody (např. Siren), tak se i tato metoda může změnit a klient bude vytvářet jiné požadavky, ale nebude muset měnit své chování.

Návrh API

Samotný návrh API obvykle popisuje dostupné resource a metody, které je možné použít nad daným resource. Pro jednotlivé akce nad resource popis se navrhne způsob vytvoření validního požadavku a jeho reprezentace včetně formátu (média typu), HTTP hlaviček a popisu jednotlivých dat. Obdobně pro odpověď také jeho reprezentaci včetně formátu (média typu), HTTP hlaviček a popis dat spolu s HTTP kódem odpovědi. Tento návrh umožňují a ulehčují popsat formáty zmíněné v sekci 3.5 – API Blueprint, RAML, Swagger či další.

Dokumentace

Důležitým článkem, který spojuje všechny předchozí body je lidská dokumentace. Ta by měla obsahovat co nejvíce informací pro klienty, aby mohli konzumovat API. Všechny předchozí body jsou pro klienty důležité – použitá identifikace a autentizace, způsob verzování a samotný popis API (resource, metody, URI, atd).

Dalším rozměrem je popis co znamenají jednotlivý resource v realitě a popis objektu, který je takto reprezentován. Stejně tak vysvětlení pro jednotlivé metody, především POST, který může reprezentovat různé akce nad resource a jeho sémantika je hodně volná. Všechny tyto informace nutné k pochopení chování API by měli být součástí dokumentace. Nevýhodou dokumentace je její zastarávání, protože při každé změně v API se stává dokumentace neaktuální a musí se aktualizovat.

Monitoring

Součástí návrhu je také návrh monitoringu API:

- testování bezpečných metod – např. zda API odpovídá na GET se správnými daty či strukturou
- očekávaná doba pro odpověď
- očekávaný počet dotazů na jednotlivé resources nebo konkrétní metody

Cache

Do implementace patří také podpora cache. Strategie jakým způsobem se budou či nebudou ukládat jednotlivé reprezentace do cache může ovlivnit výslednou odezvu API, ale může taky způsobit neaktuálnost některých dat. API využívající protokol HTTP by měli využívat mechanismy popsané v RFC 2616 [70].

Součástí je také strategie spojení cache a autentizace – cache pro jednotlivé uživatele nebo pro všechny stejné může mít dopad na počet dotazů na API nebo monitoring využívání API jednotlivými konzumenty.

3.8.3 Nasazení a provoz API

Z pohledu provozu API je nutný především monitoring jeho chování:

- monitoring použití jednotlivých konzumentů včetně použitých verzí
- monitoring výkonnosti – odezvy API na jednotlivé dotazy

- monitoring chybových stavů (chybné dotazy a odpovědi)
- monitoring výkyvů použití jednotlivých konzumentů – prudký pokles či nárůst dotazů může identifikovat chybu v konzumentu nebo jiný problém

Do provozu také patří sledování dostupnosti API (pro domluvená SLA) a ochrana před útoky (přetížení API dotazy, pokusy o vložení nebezpečných dat).

3.8.4 Vypnutí a zrušení API

Vypnutí API je poslední částí životního cyklu API. API se může zrušit z několika důvodů:

- nejsou již žádní konzumenti
- nahrazení novějším API
- strategické rozhodnutí neposkytovat API veřejně (přechod na interní)
- ekonomické či technologické důvody (již se nevyplatí API udržovat či provozovat)

Častějším případem je vypnutí konkrétní verze API (nahrazené novější verzí) či případně vypnutí konkrétních endpointů, které přestanou být využívány nebo spravovány.

Vypnutím API končí jeho životní cyklus.

Kapitola 4
Metodika pro návrh a dokumentaci API

Jedním z cílů této práce bylo vypracovat metodiku pro návrh a dokumentaci API – zejména strukturu a verzování. Strukturou API je myšleno především návrh jednotlivých resources a jejich vztahů, návrh použitých endpointů a struktura reprezentací.

V části 3.8 o životním cyklu API jsem definoval čtyři kroky: *Plán a strategie*, *Návrh a implementace API*, *Nasazení a provoz API* a *Vypnutí a zrušení API*. Pro metodiku návrhu a dokumentace API předpokládám, že krok *Plán a strategie* je hotový a je jasný rozsah budoucího API.

Z bibliografie jsem vybral dva přístupy k návrhu API popsané v 4.1.1 a 4.1.2. V 4.2 doplním vlastní doporučení pro návrh a v 4.3 přidám doporučení pro dokumentaci.

4.1 Vybrané metodiky z bibliografie

4.1.1 Pragmatický REST

V knize APIs: A strategy guide [6][1]) je doporučen „pragmatický přístup k návrhu REST API“ obsahující následující principy:

- *Záleží na URI a parametrech* – dobře navržené URI dělá API lépe konzumovatelné, objevitelné a rozšiřitelné. Ve Fieldingově definici pro REST je URI zapouzdřena pomocí hypermédií.
- *Záleží na formátu dat* – jaké data a formát API vyžaduje při požadavku a jakým způsobem jsou tvořeny odpovědi
- *Záleží na HTTP status kódu* – použití různých HTTP status kódů podle jejich sémantiky HTTP je lepší než obecná chyba
- *Ostatní informace skryté* – bezpečnost, limit API a další mají být schované v HTTP hlavičkách
- *Jasná pravidla pro verzování* – rozhodnutí, kde bude verze API – v URI, v hlavičce či bude verzována jinak
- *Pravidla pro URI*
  - URI pro kolekce resource by měla obsahovat množné číslo – např. `/customers`
  - URI pro jednotlivé resource by měla obsahovat jednotné číslo následované unikátním primárním klíčem – `/customer/Bob`
  - Je možné použít na začátku URI *identifikující cestu* pro verzi nebo prostředí – `/v1/customer/Bob`
  - Po *identifikující cestě* musí být v URI jen kolekce nebo resource
- *Pravidla pro resource*
  - Resource by měl používat metody GET pro čtení, PUT pro aktualizaci a DELETE pro smazání.

[1]) Strana 62.

- Kolekce by měla používat metodu GET pro čtení celé, či části kolekce a metodu POST pro přidání nového resource do kolekce.
- Resource mohou používat metodu POST pro změnu jejich stavu.

Tuto metodiku lze shrnout jako úroveň 2 z Richardson Maturity Modelu zmíněného v sekci 3.4. Pro úroveň 3 chybí podpora hypermédií, kterou autoři záměrně vynechávají, protože v ní vidí „akademickou čistotu“ a větší komplexitu při implementaci serveru a zároveň malý užitek pro klienty. Tato metodika ale shrnuje stávající doporučené praktiky pro tvorbu REST API.

4.1.2 API s podporou hypermédií

Druhý příklad metodiky od Richardsona a Amundsena je popsaný v knize *RESTful Web APIs* [17][1]) a obsahuje sedmikrokový návod pro návrh hypermédiových API:

1. Sepište všechny informace, které může klient chtít získat nebo vložit pomocí API. Ty se stanou *sémantickými deskriptory*, které mají tendenci vytvářet hierarchii. Deskriptory odkazující na reálné objekty obsahují obvykle i detailnější deskriptory (např. u osoby jméno a příjmení). Seskupte tyto deskriptory intuitivně do souvisejících skupin.
2. Nakreslete stavový diagram pro API. Každý box na diagramu znázorňuje reprezentaci – dokument, který seskupuje deskriptory. Tyto reprezentace spojte orientovanými hranami tak, jak by klient použil. Tyto hrany jsou stavové přechody vyvolané HTTP požadavky. Není nutné přímo specifikovat HTTP metody, ale je vhodné označit bezpečné, idempotetní a ostatní přechody.

   V této chvíli se mohou z některých sémantických deskriptorů stát vztahy[2]) (ze zákazníka byznysu se může stát odkaz na osobu pomocí vztahu `zákazník`). Iterujte kroky 1 a 2 dokud nejste spokojeni s vyznačenými vztahy a deskriptory.
3. Po prvních dvou krocích je znázorněna sémantika protokolu API (které HTTP požadavky bude klient vytvářet) a aplikační sémantika (která data se budou posílat). Ve třetím kroku doporučují autoři srovnat a případně upravit názvy sémantických deskriptorů a vztahů s již existujícími – registrovanými na IANA[3]), `schema.org`, `alps.io` či dalšími specifičtějšími média typy. Tento krok může změnit sémantiku protokolu, kdy se mohou měnit typy přechodů. Tento krok opět zahrnuje iterace zpět přes 1. a 2. krok.
4. Vyberte existující média typ, nebo definujte nový. Média typ musí být kompatibilní s navrženou sémantikou protokolu a aplikační sémantikou z předchozích kroků. Pro některé domény již existuje média typ, který pokrývá část aplikační sémantiky, ale jeho použitím bude nutné přejmenovat některé deskriptory nebo vztahy z kroku 3.
5. Napište profil, který bude dokumentovat aplikační sémantiku a vysvětlovat všechny deskriptory a vztahy, které nejsou standardizovány, nebo popsány ve vybraném média typu.
6. Implementujte stavový diagram z kroku 3, kdy odpovědi budou dle vybraných média typů z kroku 4, do kterých bude doplněn profil z kroku 5. Samotná data budou obsahovat názvy sémantických deskriptorů z kroku 1 a budou také obsahovat hypermédia pro přechod do dalších stavů z kroku 2.
7. Vystavte „domácí“ URI. Pokud jste udělali předchozí kroky správně, tak URI je jediná informace nutná pro konzumenty API a vše podstané se dozvědí z použitého média typu, profilu a dalších informací obsažených přímo v odpovědi serveru. Dále

[1]) strana 158
[2]) link relation
[3]) `https://www.iana.org/assignments/media-types/`

je možné napsat další návody a ukázkové klienty pro jednodušší ukázku fungování API.

Autoři této metodiky jsou velkými zastánci hypermédií a snaží se o jejich rozšíření a změnu pohledu na API z jednotlivých resources více na jejich vztahy jako stavový diagram. Rozšíření API podporující hypermédií je v aktuální době stále malé a pro jeho rozšíření je nutné větší počet serverů, které následně budou inspirovat klienty a další servery. Nevýhodou této metodiky je větší úvodní pracnost a návrh před samotnou implementací, kdy je třeba najít vhodný média typ, nebo vytvořit nový a popsat další dokumentaci. Výhodou hypermédií je pak menší provázanost klienta a serveru (např. se nemusí používat zadrátované URI), které se vyplatí v budoucích změnách a rozšíření API. Nevýhodou je potřeba chytřejšího klienta, který bude umět pracovat s daným API.

4.2 Metodika návrhu API

V předchozích kapitolách 4.1.1 a 4.1.2 jsem popsal dvě různé metodiky z bibliografie, kdy *Pragmatický REST* dle mého názoru ukazuje aktuálně nejčastěji používaný přístup k návrhu REST API. Druhý přístup *Procedura návrhu API s podporou hypermédií* vychází ze samotné definice REST od Fieldinga a snaží se využít principu hypermédií. V následujícím doporučení pro návrh API jsem se pokusil extrahovat z obou přístupů to nejlepší a nejužitečnější, doplněné o další doporučené postupy.

Základem pro návrh by měla být znalost sémantiky HTTP protokolu zmíněné v sekci 3.3.1. Do toho patří zejména správné použití HTTP metod (bezpečný GET, idempotence PUT a DELETE, atd.) a HTTP status kódů používaných pro odpověď (200 pro požadavek v pořádku, 400 chyba klienta, 500 chyba serveru).

Se znalostí a dodržením předchozích pravidel je dalším krokem návrh jednotlivých resources. Resource jsou ale dost abstraktní, proto je mi bližší přístup dle Richardsona a Amundsena (sekce 4.1.2) – začít se sémantickými deskriptory a definicí jejich vztahů. Důležité je také zmínit, že API není obraz databáze a mělo by klienta odstiňovat od vnitřní implementace uložení dat.

Při vytváření reprezentací je vhodné počítat a umožňovat budoucí rozšíření při zachování zpětné kompatibility. U JSON to znamená např. použití objektu místo pole v kořenovém elementu, zamyšlení nad budoucím počtem jednotlivých prvků – pokud není jasné, že bude mít vždy jen jednu skalární hodnotu, tak je nutno použít pole nebo objekt umožňující přidání dalších hodnot.

Pokud je nutné vytvořit nekompatibilní změnu, která ovlivní stávající klienty, tak je potřeba vybrat jednu ze strategií zmíněných v sekci 3.8.2. Nejpřímočařejší a nejjednoduší je strategie verzování v cestě URI, kterou bych doporučil. Pokud je reprezentace rozšiřitelná, tak verzování je potřeba ve výjimečných případech. Pokud se radikálně změní aplikace poskytující API, tak se změna dá považadovat spíše za nové API než novou verzi starého API.

Pokud reprezentace obsahuje URI, tak by měl být označeno jako odkaz i se svým vztahem pomocí hypermédií. Dalším krokem je vyznačení dalších možných přechodů nebo akcí s daným resource – např. formát Siren (zmíněný v sekci 3.6.5) umožňuje popsat tyto akce a klient je následně může využít namísto implementování přesného popisu dle dokumentace, který se při změně API může také změnit.

Pro reprezentanci bych doporučil použít nejspecifičtější média typ – pokud existuje přímo pro doménu, pokud ne, tak použít média typ podporující hypermédia (např. HAL zmíněný v 3.6.4 nebo Siren v 3.6.5).

Pokud API přijímá argumenty nebo data, tak je vhodné, aby je přijímala asertivně. V případě chyby oznámí klientovi dostatek informací pro identifikaci problému – např. pomocí implementace média typu *HTTP problem* zmíněného v sekci 3.6.3.

Častým tématem metodik pro tvorbu REST API je struktura URI a pojmenování resources a endpointů. Hlavním doporučením je zachování konzistence a čitelnosti. Obvykle se resource označující kolekce popisují množným číslem, jednotlivé resource připojují k URL kolekce unikátní identifikátor. Podle stupně využití hypermédií ale URI ztrácejí důležitost. V plně hypermédiovém API, kde klient využívá *domácího* dokumentu a dále mění stav jen dle nabízených možností, mohou být URI libovolná a mohou se měnit bez dopadu na klienta, který se naviguje jen dle možností a jemu známým relacím.

Takových klientů je ale obvykle menšina a typický klient požaduje obvykle jeden specifický resource a využívá jeho URI. Pro tyto potřeby by mělo být URI srozumitelné pro klienty a jednoduše vytvořitelné.

Všechna tato doporučení shrnuje Richardson Maturity Model zmíněný v sekci 3.4 a jeho úrovně 2 a 3. Každé nové API by mělo být alespoň na úrovni dva a případně používat hypermédia alespoň pro již zmíněné URI v datech.

Shrnutí metodiky pro návrh API:

- Používat sémantiku HTTP:
  - Správné použití metod – bezpečné operace, idempotence, použití specifické metody
  - Použití HTTP status kódů dle jejich nejbližšího významu pro daný případ
- Návrh resources by neměl být obrazem vnitřní implementace (databáze), ale vyjadřovat a podporovat použití API konzumenty
- Použít formát odpovědi tak, aby umožňoval rozšíření se zpětnou kompatibilitou
- Pro URI v datech použít hypermédia
- Používat co nejvíce specifický média typ
- Pokud klienti nebudou plně využívat hypermédií, tak by URI jednotlivých endpointů mělo být srozumitelné a jednoduše sestrojitelné

4.2.1 Kolaborace při návrhu API

Při tvorbě API, zvláště interních, mohou spolu producent a konzument kolaborovat při návrhu a implementaci API. Dále popíši dva způsoby této kolaborace a jejich výhody a nevýhody.

1. Producent vytvoří návrh a dokumentaci, kterou konzument připomínkuje a následně vytvoří klienta oproti mock implementaci serveru (např. pomocí Apiary).
2. Producent implementuje návrh a konzument zkontroluje svého klienta oproti finálnímu serveru.

Výhody:

- Zmenšení budoucích změn v implementaci serveru
- Rychlejší agilní vývoj s jedním konzumentem
- Méně nekompatibilních verzí API
- Konzument může navrhnout API pro jeho jednodušší a optimálnější implementaci

Nevýhody:

- Nutná kooperace mezi producentem a konzumentem
- Využití mock serveru nemusí pokrývat všechny krajní případy
- Složitější kooperace s více konzumenty

1. Producent vytvoří implementaci API a důkladnou dokumentaci
2. Konzument implementuje a případně požaduje od producenta změny

Výhody:

- Oddělená implementace producenta a klienta, producent není závislý na jednom konzumentu a jeho připomínkách

Nevýhody:

- Klient může mít složitější implementaci
- Klient může požadovat nekompatibilní změny, a tím může vznikat více verzí API

4.3 Metodika dokumentace API

Dokumentace je jedním z prvních míst, které navštíví konzument API při implementaci svého klienta.

Základem dokumentace by mělo být vysvětlení základních pojmů a sémantických deskriptorů popisující danou doménu a cílů API.

Pokud API využívá nějaký způsob identifikace či autentizace (zmíněné v sekci 3.7), tak by mělo být součástí dokumentace společně s ukázkou, jak daný klíč či údaje získat, a také příklad použití.

Pro popis samotných resource, jejich endpointů, akcí a dalších detailů existují tři hlavní formáty zmíněné v sekci 3.5 – API Blueprint [26], RAML [39] a Swagger [33]. Pro každé API bych jeden z těchto formátů využil, dle srovnání v sekci 3.5.6 preferuji API Blueprint.

Jejich definice by měla obsahovat všechny reprezentace a formáty jak pro úspěšné požadavky, tak i pro požadavky, které skončily chybou (400 nebo 500). Stejně tak vyjmenování a vysvětlení všech možných HTTP status kódu, se kterými se může klient setkat. V případě chyb 400 (chyba požadavku) by měla být zobrazena ukázková odpověď, např. po neúspěšné validaci parametrů, aby klient měl představu, jak takovou chybu zpracovat.

Předchozí body popisují API jako kontrakt a podle něho jsou konzumenti schopni správně implementovat své klienty. Další části dokumentace je zjednodušení tohoto procesu pro konzumenta:

- Interaktivní konzole umožňující vyzkoušet reálné odpovědi API
- Vytvoření ukázkového klienta, který využívá API
- Poskytnutí SDK[1]), které poskytuje hotovou implementaci v programovacím jazyce
- Možnost vygenerovat ukázkový kód volání

Shrnutí metodiky pro dokumentaci API:

- Vysvětlení základních pojmů a cílů daného API
- Způsob identifikace nebo autentizace včetně informace, jak potřebné údaje získat
- Detailní popis struktury API a jeho chování, včetně možných chybových stavů
- Doplňující návody, ukázkový klient, SDK nebo možnost vygenerovat či vyzkoušet konkrétní volání API

[1]) software development kit

Kapitola 5
Analýza požadavků na aplikaci

Základní požadavky napsané ve formě user stories byly specifikovány v kapitole 2.3.2 a v této kapitole je budu podrobovat analýze.

5.1 Analýza User stories

Jako první krok jsem prošel jednotlivé user story a pokusil se analyzovat detailnější požadavky a úkoly, které jsou nutné pro jejich naplnění.

5.1.1 User story 1

Jako vývojář chci znát API, které poskytují moje aplikace, které nejsou konzumovány jinými aplikacemi, abych je mohl případně odstranit.

Z user story vyplývá několik faktů:

1. V systému bude entita *Application*, která bude obsahovat atributy:
   - *UID* – unikátní identifikátor pro rozlišení jednotlivých aplikací
   - *name* – název aplikace
   - *API list* – seznam poskytovaných API
2. Entita *API* s atributy:
   - *Application* – aplikace, ke které se dané API vztahuje
   - *Consumer list* – seznam konzumentů (aplikací), které využívají toto API
3. V systému bude výpis seznamu aplikací, který bude umožňovat filtr dle vlastníka či týmu.
4. U aplikace bude označeno API, které nemá žádné konzumenty. Toto zjištění by mělo být potvrzeno analýzou a využitím reálných požadavků, zda API skutečně není konzumováno.

5.1.2 User story 2

Jako vývojář chci přehledně vidět závislosti aplikace, kterou právě upravuji, abych mohl analyzovat dopad změn na API v předstihu a případně diskutovat změny s vlastníky dotčených aplikací.

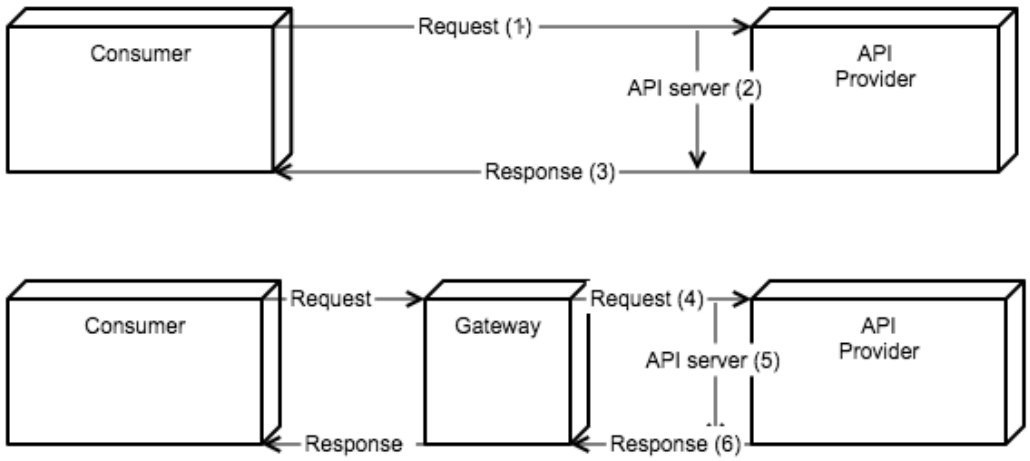
V předchozí user story 5.1.1 jsem již definoval základní kameny aplikace, které budu dále rozšiřovat.

Tato user story specifikuje, že bude existovat pohled na detail aplikace, která bude zobrazovat všechny její API a všechny konzumenty těchto API – tj. dotčené aplikace. Pro dotčené aplikace by měl být viditelný jejich vlastník, aby bylo možné s ním případně diskutovat změny, jak je v user story uvedeno.

5.1.3 User story 3

Jako vývojář / DEVOPS chci znát výkonnost API mých aplikací, abych mohl zajistit požadovanou dostupnost a kvalitu služby.

Výkonnost API lze měřit několika způsoby zobrazených na obrázku 5.1:



Obrázek 5.1. Měření výkonnosti API: Horní varianta zahrnuje doby při přímém spojení konzumenta a producenta. Dolní varianta popisuje zapojení gateway.

- *Client Request to Reponse* – měření celkové doby požadavku od jeho odeslání, až k přijetí odpovědi. Na obrázku 5.1 se jedná o součet doby označené (1), (2) a (3). Nejlepší metrika, která zohledňuje veškerou dobu od vytvoření požadavku včetně otevření síťového spojení, odeslání požadavku, až do přijetí odpovědi.

  Pseudokód způsobu měření v klientovi muže být následovný:

```
request = new request()
start = timestamp.now()
response = send(req)
length = timestamp.now() - start
```

  Tato metrika vyžaduje zapojení klienta, který sbírá informace a následně je i odesílá do databáze. Nevýhodou je nutná implementace u všech klientů.
- *Provider Request to Response* – měření doby odezvy producenta od přijetí požadavku, do odeslání odpovědi. Na obrázku 5.1 se jedná o dobu označenou (2). Na straně producenta je možné měřit samotné zpracování na serveru, které je následně vhodné při určení, zda je pomalé místo v síťové komunikaci či v aplikaci.

  Pokud je k dispozici měření od konzumenta i od producenta, je možné vypočítat[1]) dobu síťového spojení.
- *Gateway* je řešením, kdy mezi konzumentem a producentem je ještě jedna vrstva, přes kterou konzument s producentem komunikují. Pak je možné měřit komunikaci od gateway až po odpověď producenta. Na obrázku 5.1 se jedná o součet doby označené (4), (5) a (6).

  Hlavní výhodou je nezávislost na implementaci klienta i serveru, které nemusejí implementovat žádnou logiku měření. Toto řešení je také nejvhodnější pro měření

[1]) Odečtením doby *Provider Request to Response* od celkové doby *Client Request to Reponse*

veřejných API, které konzumují různí klienti, které mohou být spravování různými organizacemi. Je to také způsob, kterým funguje většina stávajících nástrojů, které jsem následně vyzkoušel a popsal v sekci rešerše 6. Velkou nevýhodou je, že všechny volání musí jít přes tento jediný bod a v případě jeho selhání nefunguje samotná služba.

Řešení pomocí Gateway také nesplňuje nefunkční požadavky specifikované v sekci 2.3.3 a není preferované řešení z důvodu možného problému v případě výpadku.

Pro kvalitu služby je nutné ukládat ke každému záznamu HTTP status kód odpovědi, pro rozlišení různých stavů (200 a hlavně chybových 400, 500).

Nová entita *API call* – jedno volání API. Tato entita má atributy:

- *API* – cílové API
- *Consumer* – konzument
- *Response time* – doba trvání odpovědi
- *Response code* – návratový kód volání (pro následné odlišení chybových volání)
- *Timestamp* – datum a čas, kdy bylo volání API (v případě producenta čas odeslání odpovědi, v případě konzumenta čas přijetí požadavku)

5.1.4 User story 4

Jako vývojář chci mít přehled o API, které konzumují mé aplikace, abych v případě problémů věděl, které API je způsobuje a měl možnost kontaktovat jeho vlastníka.

Tato user story je obdobná jako 5.1.1 a jedná se jen o pohled z druhé strany – od aplikace jako konzumenta.

Pro každou aplikaci budou vypsány API, které konzumuje na základě dokumentace i monitoringu využití a porovnání obou způsobů. Pro konzumaci by se zdálo vhodné přidat i popis, z jakého důvodu či za jakým účelem se API konzumuje, ale obvykle tato hodnota rychle zastarává a stává se neaktuální.

Aplikace bude také obsahovat seznam chybných odpovědí (400, 500) včetně jejich požadavku a odpovědi. Proto doplním entitu *API call* o dodatečné atributy:

- *Response* – serializovaná odpověď pro analýzu v případě chyby
- *Request* – serializovaný požadavek pro analýzu v případě chyby

Pro konzumenta následně bude existovat pohled, kdy bude zobrazen detail chybného volání, včetně dat o požadavku a odpovědi.

5.1.5 User story 5

Jako vývojář chci být informován v případě že verze API využívané mojí aplikací je nahrazené novější verzí (a stará verze se stane neaktuální a nepodporovanou), abych mohl přejít na aktuální verzi

Pro entitu *API* přibude kolekce verze obsahující entity *API Version*. *API Version* bude obsahovat atributy:

- *Version* – označení verze, např. v1, v2
- *Publish date* – datum vydání verze
- *Termination date* – datum ukončení verze
- *State* – stav verze, které bude v jednom ze stavů:
  - *ACTIVE* – verze je aktivní

- *DEPRECATED* – verze je aktivní, ale má nastavený datum, kdy bude vypnuta (Termination date)
- *TERMINATED* – již vypnutá verze

Pro stav *DEPRECATED* bude aplikace zobrazovat:

- Zvýraznění těchto API v detailu konzumujících aplikací spolu s datem jejich vypnutí
- Konzumující aplikace v detailu producentské aplikace
- Speciální přehled všech konzumentů využívající různé *DEPRECATED* API verze

Při změně API verze na *DEPRECATED* a *TERMINATED* bude provedena notifikace konzumenta.

5.1.6 User story 6

Jako vývojář chci znát dostupnost konzumovaných API abych mohl navrhnout svojí aplikaci pro maximální spolehlivost a funkčnost

Podstatou této user story je přidání informace o dostupnosti jednotlivých API. Dostupnost ve vnitřním prostředí lze rozdělit na dvě hlavní skupiny:

- *high-availability* (HA) – dostupnost 99,99% a více, kdy je možné se spolehnout na funkčnost API téměř vždy. Obvykle implementováno pomocí clusteru, který zajišťuje vysokou spolehlivost a dostupnost.
- *non high-availability* (non HA) – menší dostupnost, která může být způsobena během aplikace na jednom stroji a případném výpadkovém nasazení nové verze

Pro entitu *API* přibude atribut *availability* obsahující jednu z předchozích hodnot.

Pokud je dostupnost jen *non HA*, tak by programátor při napojování měl v návrhu klienta počítat s občasnou nedostupností – např. odpovědi z API (typicky GET) ukládat do cache, případně unikátní požadavky (POST, PUT) ukládat do fronty, která zajistí spolehlivé odeslání do API při jeho nedostupnosti. Při *HA* lze počítat s vysokou spolehlivostí, kde případný výpadek celého clusteru je málo pravděpodobný a případná nedostupnost by měla být ošetřena, ale přidání návrhu obdobného jako u *non HA* (fronta) je obvykle zbytečné. Vždy ale závisí na požadované spolehlivosti a způsobu zpracování chyb.

5.1.7 User story 7

Jako vývojář chci získat na jednom místě podrobný popis všech dostupných API, které může moje aplikace využít, abych rychleji získal informace a mohl vytvořit novou funkcionalitu rychleji.

Nástroj by měl obsahovat pohled na všechny dostupné aplikace a jejich API. Pro podrobnější popis API bude existovat v entitě *API* atribut *documentation* s odkazem na interní dokumentaci.

V seznamu aplikací a jejich poskytovaných API by mělo jít vyhledávat dle názvu aplikace či dle vlastníka.

5.1.8 User story 8

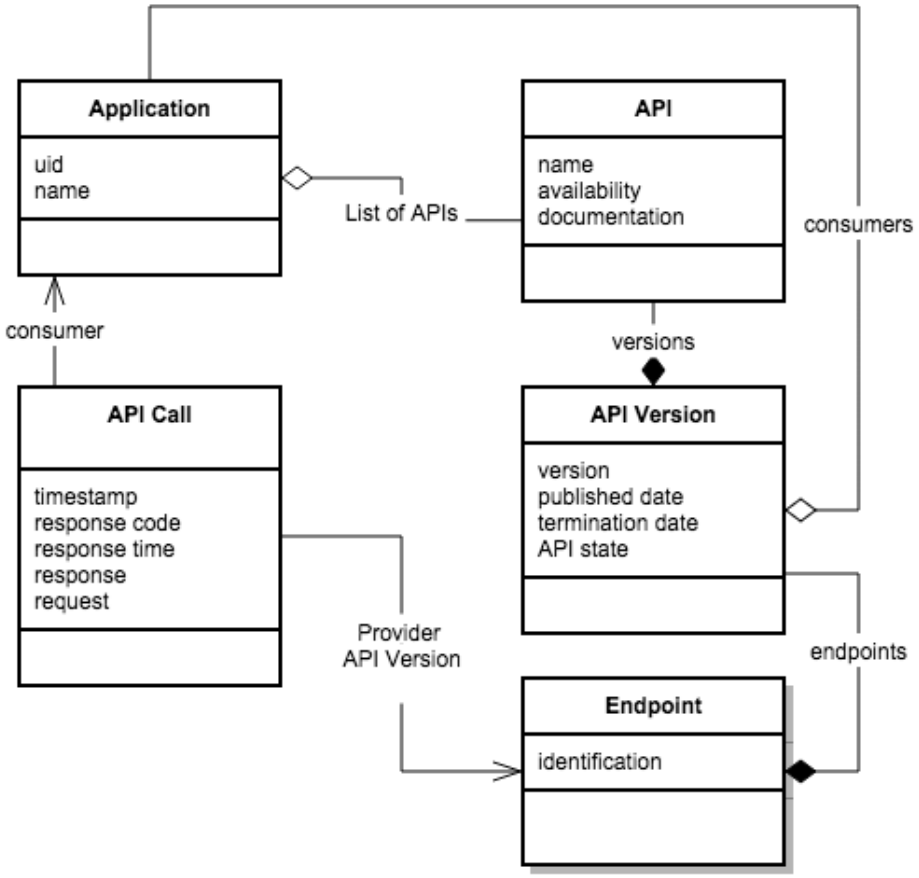
Jako vývojář, DEVOPS a IT architekt chci vidět změny ve využití API a dostat upozornění, pokud se některé API chová zvláštně, abych mohl identifikovat chyby a problém a případně mohl lépe naplánovat systémové zdroje.

Aplikace bude sledovat využití jednotlivých API a jejich konzumenty. V případě výkyvu (extrémní nárůst nebo pokles) požadavků na API bude vlastník notifikován.

Detail využití API bude viditelný i v detailu aplikace.

5.2 Shrnutí analýzy

Z požadavků vyplývající z user stories jsem vytvořil doménový model zobrazený na obrázku 5.2.



Obrázek 5.2. Návrh doménového modelu

5.2.1 Uživatelské role

User stories zmiňují uživatelské role vývojář, DEVOPS a IT architekt. Pro jednoduchost bude uživatelská role jednotná – uživatel systému, protože nástroj bude sloužit jen k interním účelům.

5.2.2 Požadavky na uživatelské rozhraní

- Výpis aplikací dle vlastníka (US1 5.1.1)
- Označení API, které nemají konzumenta (US1 5.1.1)
- Pohled na detail aplikace se zobrazením všech poskytovaných API a jejich konzumentů (US2 5.1.2)
- Pohled na detail aplikace se zobrazením konzumovaných API (US4 5.1.4)
- Pohled na detail chyby při volání API (zobrazení požadavku a odpovědi) (US4 5.1.4)
- Zvýraznění *DEPRECATED* API v detailu konzumující aplikace (US5 5.1.5)
- Vyhledávání v aplikacích a API dle názvu či vlastníka (US7 5.1.7)

5.2.3 Požadavky na notifikační systém

- Notifikace konzumentů v případě změny API verze na *DEPRECATED* nebo *TERMINATED* (US5 5.1.5)
- Notifikace vlastníka aplikace v případě problému s dotazy na API – extrémní výkyv (snížení/zvýšení) počtu požadavků (US8 5.1.5)

Kapitola 6
Rešerše nástrojů pro API management a dokumentaci

Jako první jsem udělal rešerši existujících nástrojů, které řeší podobné problémy a funkce. Pokud to bylo možné (nástroj je zdarma či obsahuje testovací verze), tak jsem nástroj vyzkoušel i prakticky. Cílem této rešerše bylo zkusit najít nástroj, který by se dal dále využít či integrovat pro požadované řešení. Druhotným bodem byla případná inspirace v cizích řešeních či prozkoumání nástrojů, které budou vhodné i pro jiné účely využití.

Úvodním krokem při vytváření rešerše bylo stanovení kritérií pro požadavky a porovnání (sekce 6.1), následované nalezením vhodných nástrojů (sekce 6.3). Jednoduchým a naivním zdrojem nástrojů bylo zadání vhodných klíčových slov (*API management* (sekce 6.2), *API portal* apod.) do vyhledávání Googlu. Dalším zdrojem byl report od Forrester *The Forrester Wave™: API Management Solutions, Q3 2014* [71] srovnávající jednotlivé nástroje na trhu dle jejich funkcí, strategie a přítomnosti na trhu.

6.1 Kritéria pro srovnání nástrojů

Z požadavků na nástroj specifikovaných v user stories (sekce 2.3.2) jsem vybral následující kritéria pro srovnání nástrojů:

- **Architektura** – jakým způsobem nástroje sbírají data o aplikacích a sledování interakcí mezi konzumentem a producentem, detailněji rozebrané v podkapitole 6.1.1. (z nefunkčních požadavků 2.3.3)
- **Identifikace a autentizace** – jaké způsoby identifikace a autentizace nástroje umožňují. (z US2 5.1.2, US4 5.1.4)
- **Podpora pro dokumentaci aplikací a API** – jakým způsobem nástroje podporující dokumentaci a popis samostatných aplikací a jejich API (z US1 5.1.1, US2 5.1.2, US4 5.1.4, US6 5.1.6, US7 5.1.7)
- **Životní cyklus API** – možnost verzování jednotlivých API včetně oznamování vypnutí. (z US5 5.1.5)
- **Monitoring** – využití jednotlivých API, měření výkonnosti API a zpracování chybných odpovědí. (z US3 5.1.3, US8 5.1.8)
- **Vysoká dostupnost** – podpora vysoké dostupnosti[1]) nástroje. (z nefunkčních požadavků 2.3.3)
- **Vývoj a komunita** – pro open source nástroje je také podstatná živost projektu, zda je nástroj aktivně spravován, vyvíjen či využíván

[1]) high availability

6.1.1 Architektura

Architektura nástrojů lze rozdělit na dvě skupiny:

- **Gateway/Proxy** – nástroj je prostředníkem mezi producentem a konzumentem
- **Asynchronní** – nástroj přijímá data od producenta nebo konzumenta, ale nevstupuje do jejich vzájemné komunikace

Druhou osou rozdělení nástrojů je jejich umístění vůči producentově síti, organizaci a infrastruktuře:

- On Premise – řešení umístěné uvnitř vlastní infrastruktury provozované producentem či jeho organizací
- Cloud (SaaS) – řešení umístěné v internetu, běžící na externí službě provozované třetí stranou. Data jsou uložena v externím úložišti, což může
- Hybridní – kombinace předešlých řešení, obvykle část běží na vlastní infrastruktuře a druhá část využívá externí službu

Gateway/Proxy

U Gateway/Proxy řešení je nástroj prostředníkem mezi konzumentem a producentem.
Výhody:

- jednoduché pro implementaci – změní se endpoint na gateway/proxy
- možnost měření doby odpovědi
- možnost úpravy logiky na gateway/proxy – například změna URI na API bez dopadu na klienta. Současně se jedná o riziko přesunu moc velké logiky na gateway.

Nevýhody:

- single point of failure - pokud nefunguje gateway/proxy, pak nefunguje nic a producent nemá přímou možnost zajistit případnou opravu.
- přidání dalšího mezistupně mezi producenta a konzumenta – možné zpomalení a prostor pro výpadek

Asynchronní

U asynchronního řešení je nástroj mimo komunikaci producenta a konzumenta a přijímá od nich data asynchronně.
Výhody:

- výpadek nástroje nezpůsobí přerušení komunikace producenta s konzumentem
- lze implementovat do starých API bez nutnosti zásahu do klientů (pokud klienti nepoužívají identifikaci, tak)

Nevýhody:

- nutnost implementace na každém API dodatečné logiky nesouvisející přímo s funkčností API

6.1.2 Výběr nástroje

Doplněním kritérií pro výběr nástroje zmíním prezentaci, kde Kai Wähner [72][1]) uvádí několik otázek, které je vhodné si odpovědět před vybíráním vhodného nástroje:

- Které API funkce požadujete? (Gateway, Portal, Analytics)

[1]) slide 40

- Jak jednoduché má být nainstalovat nástoj? Je nástroj dost vyspělý a silný ve svých funkcích?
- Kolik specifických funkcí pro API je ihned k dispozici (pro implementaci, integraci, testování, deployment, logování, subscription, billing, dashboard atd.). Je možné rozšíření (konektory, bezpečnost, reportování atd.)?
- Je cílem vytvořit jen adresář existujících služeb, nebo vytvoření nové infrastruktury pro vytváření, řízení, nasazování a spravování služeb?
- Bude se používat REST nebo i jiné styly či protokoly (SOAP či JMS)?
- Je potřeba flexibilní konfigurace, možnosti routování a management pomocí různých bezpečnostních standardů (LDAP, SAML, OAuth, Kerberos, atd.)?
- Bude vyžadována vysoce škálovatelná architektura pro milióny požadavků (založené na event driven architektuře místo synchronních HTTP volání)?
- Jaké jsou požadavky na cache a omezení požadavků (throttling)?
- Je požadováno rozšíření portálu dle svých požadavků (v oblasti service managementu, developer portálu nebo analytice)?
- Bude využito i jiných služeb od stejného dodavatele (produkty pro integraci, mapování, transformaci, routování, bussiness procesy atd.)?
- Je požadováno nasazení nástroje jako on-premise řešení nebo v cloudu?

6.2 API Management

Provázanost aplikací a další požadavky zmíněné v analýze pro potřeby této práce jsou podmnožinou API Managementu. Hlavním cílem API managementu je usnadnění a zjednodušení vztahu mezi producentem API a konzumentem. Obvykle služby či nástroje API managementu poskytují následující funkce:

- Bezpečné vytváření přístupů k API a jejich kontrola - vytváření klíčů, uživatelů pro jednotlivé služby, jejich revokace atd.
- Monitoring provozu z jednotlivých aplikací - zobrazení provozu, chyb, nastavení limitu/throttlingu
- Dokumentace rozhraní API a jeho verzování - dokumentace, živé ukázky, generování kódu v různých jazycích
- Konverze vstupních a výstupních formátů - REST, XML, SOAP, atd.
- Rychlostní optimalizace - přidání cache hlaviček, CDN, atd.
- Ochrana API před zneužitím a útoky - validace, známé útoky

6.3 Nalezené nástroje a jejich zařazení

Nalezených nástrojů bylo mnoho a proto jsem je rozdělil do následujících skupin.

On-Premise Open Source nástroje:

- Apiaxle [73]
- Apiman [74]
- Kong [75]
- Tyk [76]

Cloud nástroje:

- Mashape [77]
- Apigee [78]

Poslední skupina jsou velké enterprise řešení. Ty obvykle mají vysoké ceny či jsou dostupné jen na smlouvu a velmi málo nabízejí trial verze, které bych mohl vyzkoušet a tudíž je nebudu dále popisovat.

Vybrané enterprise nástroje:

- Mashery [79]
- Informatica iPaaS [80]
- CA API Management [81]
- TIBCO Api Exchange Gateway [82]
- IBM Api Management [83]
- Axway Api Management [84]

6.4 On Premise Open Source API management nástroje

V této sekci se budu věnovat nástrojům, které jsou vydané pod open source licencí a jako on-premise řešení. Jednotlivé nástroje vyzkouším a krátce popíši jejich funkčnosti. V případě, že by některý nástroj vyhovoval některým požadavkům, tak by byla možnost ho integrovat do požadovaného nástroje.

6.4.1 Apiaxle

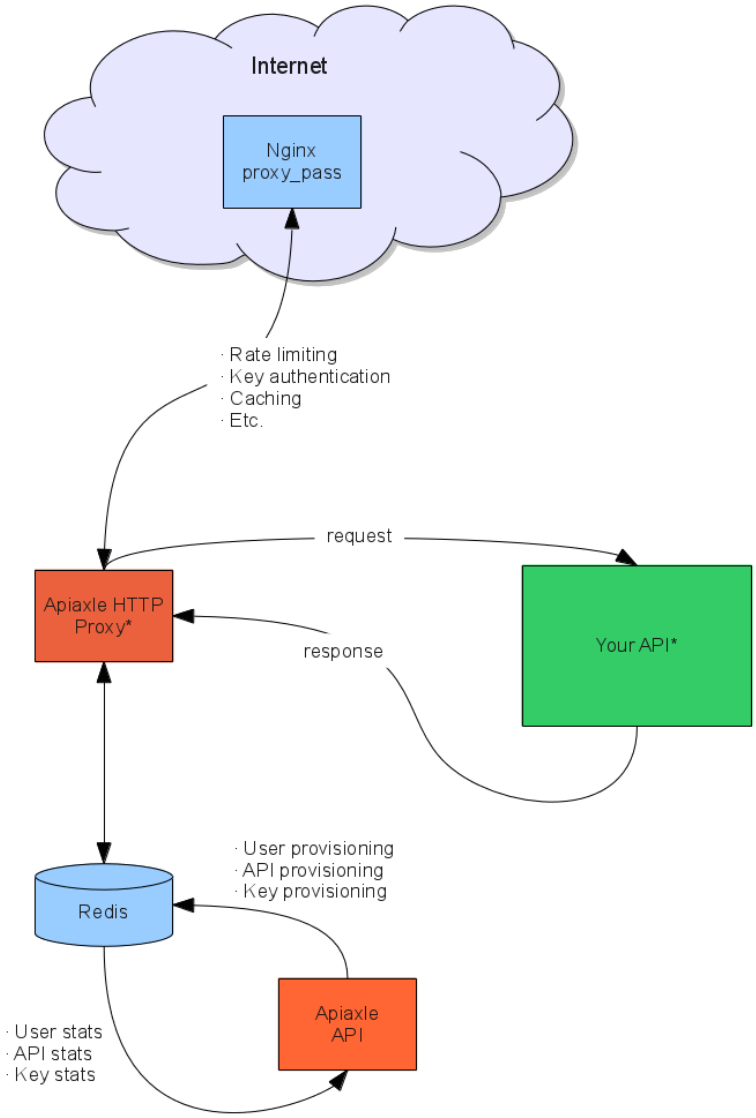
Apiaxle [73] je nástroj skládající se ze tří částí – proxy, API pro management uživatelů, klíčů, endpointů a REPL[1]) pro správu z konzole.

Architektura nástroje zobrazená na obrázku 6.1. Hlavní částí nástroje je HTTP proxy, která leží mezi mezi internetem, resp. jiným HTTP proxy (např. nginx, apache apod.), a vlastním API. Proxy komunikuje s Redis úložištěm, kde jsou data o uživatelích, klíčích a endpointech. Tato data v Redisu vytváří a konzumuje Apiaxle API, se kterým je možné komunikovat pomocí konzole. Konzole v praxi jen posílá a zpracovává požadavky na Apiaxle API a má integrovanou nápovědu. Pro toto API je detailní dokumentace, díky které existují knihovny v nodejs, Go a PHP.

Hlavní funkcionality ApiAxle jsou:

- HTTP proxy
- REST API pro management
- Autentizace jen pomocí API klíčů
- Podepisování zpráv pomocí HMAC
- Možnost přístupu k API bez klíče
- Sdružení jednotlivých klíčů do skupin (keyring)
- Analytika API, ale bez grafického rozhraní
- REPL konzole pro ovládání API
- Zdrojový kód v Javascriptu a nodejs

[1]) read–eval–print loop



Obrázek 6.1. Architektura Apiaxle: Hlavní částí nástroje je HTTP proxy, která leží mezi mezi internetem, resp. jiným HTTP proxy (např. nginx, apache apod.), a vlastním API. Proxy komunikuje s Redis úložištěm, kde jsou data o uživatelích, klíčích a endpointech. Tato data v Redisu vytváří a konzumuje Apiaxle API, se kterým je možné komunikovat pomocí konzole. zdroj: [85]

Apiaxle je relativně jednoduchý nástroj, který obsahuje jen základní funkčnosti pro API proxy a ukládá jednodušší data pro analytiku, které avšak chybí grafické rozhraní.

Bohužel vývoj open source části produktu se téměř zastavil od roku 2014, kdy byl získán [86] firmou Exicon, která vyvíjí jeho další placené části pod názvem Exicon ApiAxle [87].

Mezi placené části v Exicon ApiAxle patří Developer portál, který poskytuje webové rozhraní k vytváření uživatelů a jejich klíčů, interaktivní dokumentaci a další funkcionality postavené nad ApiAxle API a integrované do cloud platformy Exicon.

6.4.2 Apiman

Apiman [74] je nový projekt v portfoliu JBoss od firmy RedHat, který umožňuje API management. První verze ostrá verze 1.0.0. vyšla relativně nedávno na konci prosince 2014 [88]. Apiman se skládá ze dvou hlavních částí – API Manager a API Gateway.

API Manager je REST API a grafické rozhraní pro API management. Základním konceptem je *organizace*, která zastřešuje jednotlivé uživatele. Jednotlivý uživatelé mohou mít různá práva v rámci organizace. Přidání API je zde umožněno pomocí služby (Service), která má název, detaily o endpointu a další informace. Na jednotlivé služby je možné aplikovat různé politiky, které se použijí při každém požadavku na danou službu. Mezi hlavní politiky patří autentizace, omezení počtu dotazů (rate limiting) a filtrování podle IP. Pro každou službu může být definovány plány (kolekce politik), podle který je konzumenti konzumují. Jednotlivé konzumenty jsou představeny jako aplikace, které konzumují služby (API) podle přiřazeného plánu a vytvářejí tak společný kontrakt.

API Gateway následně jen aplikuje konfiguraci z API Manageru tak, že na každý požadavek použije všechny politiky.

Hlavní funkcionality:

- REST API pro management organizací, služeb, plánů a aplikací.
- Asynchronní jádro, které může běžet i pod Vert.x[1])
- Jen Basic autentizace
- Rate Limiting
- Filtrování dle IP adresy
- Import služeb z WADL
- Běží na prostředí Wildfly[2])

Nástroj bude mít určitě silnou budoucnost zvláště při využití s dalšími JBoss aplikacemi. Aktuálně je ale stále ještě v raném stádiu s omezenými funkcemi, které ale kvalitně fungují. Velmi také chybí lepší analytika a metriky spojené s využitím API, proto není tento nástroj využitelný pro účely této práce.

6.4.3 Kong

Kong je jednoduchá API vrstva, která se chová jako Gateway, vytvořená společností Mashape, která provozuje stejnojmenný cloud systém. Momentálně je ale v beta verzi 0.1.1.

Kong je postavený nad nginx[3]) a napsaný v jazyce Lua. Obsahuje modulární architekturu, kdy jednotlivé funkčnosti jsou přidávány jako pluginy. Prozatím jsou dostupné pluginy jen pro Basic autentizaci, autentizaci pomocí API klíče, logování požadavků i odpovědí pomocí tcp, udp a do souboru a omezení počtu dotazů v závislosti na autentizaci nebo klientské IP adrese.

Jako úložiště používá Kong Apache Cassandra[4]).

- Modularita
- REST API pro konfiguraci systému
- Autentizace pomocí API klíčů a Basic autentizace
- Omezení počtu dotazů
- Logování požadavků a odpovědí

Kong je prozatím jednoduchý nástroj, který je modulární a rychlý a v budoucnu bude mít určitě více funkcí. Jelikož je přímo spjatý s firmou Mashape, tak integrován hlavně do jejich cloud nástroje, který budu popisovat v části 6.5.

[1]) `http://vertx.io/`
[2]) `http://wildfly.org`
[3]) `http://nginx.org/`
[4]) `http://cassandra.apache.org/`

6.4.4 Tyk

Tyk [76] je open source nástroj, který funguje jako API Gateway. Vyvíjí ho Martin Buhr z firmy Jively a je napsaný v jazyku Go. Soustředí na dvě hlavní komponenty – Gateway a Dashboard. API Gateway společně s Dashboardem poskytují tyto hlavní vlastnosti:

- Autentizace
  - API klíče
  - expirace klíčů
  - HTTP Basic authentication
  - OAuth 2.0
  - podepisování požadavků pomocí HMAC
- Autorizace pomocí Granular Access Control
- Možnost nastavení Quoty a Rate limit pro API
- Analytická část Dashboardu obsahuje data o přístupech k jednotlivým API a dokáže segmentovat podle jednotlivých konzumentů, chyb a endpointů i přes různé verze API.
- Prototypování API pomocí mocků – je možné importovat API Blueprint
- Monitorování výkonnosti jednotlivých API
- Umožňuje notifikace na důležité události
- Gateway umí transformovat požadavky pomocí jednoduchých šablon či middlewaru
- Obsahuje cache API odpovědí podle endpointu
- Umožňuje jednoduché začlenění do interní infrastruktury s vysokou dostupností:
  - je možné spustit více load—balancovaných instancí
  - konfigurace podporuje „hot-reload“, kdy je možné bezvýpadkově změnit konfiguraci
  - obsahuje API pro „health check“, který popisuje aktuální statistiky nástroje a umožňuje monitorování výkonnosti
- REST API pro komunikaci mezi Gateway, Dashboardem a případně dalšími aplikacemi

Tyk nástroj, který obsahuje zajímavé funkcionality. Při vyzkoušení ale trpěl chybami v uživatelském rozhraní, které není příliš přívětivé. To dost stráží celý dojem z nástroje. Jasnou nevýhodou je, že se jedná o projekt jednoho vývojáře a nemá úplně jasnou budoucnost.

6.5 Cloud API management nástroje

Cloud nástroje umožňují využívat služeb běžící na cizí infrastruktuře, ale mají nevýhodu ve svém externím centrálním bodě. Proto jsem prozkoumal jen dva nástroje pro inspiraci – Mashape a Apigee Edge.

6.5.1 Mashape

Mashape [77] je americký startup ze San Franciska, který poskytuje dle svých slov největší tržiště API. Snaží se o co největší spojení konzumentů a producentů, kdy dodávají podpůrné funkce v převážně oblastech managementu, monitoringu a dokumentace API. Další hlavní funkcí je možnost zpoplatnění veřejných API, která vytváří zajímavý model monetizace pro producenty. Technologicky Mashape funguje jako proxy mezi konzumentem a producentem.

V oblasti API managementu poskytuje Mashape následující služby:

- Kontrola přístupu k API (Access Control)
- Podpora několika druhů autentizace:
  - Query parametr
  - HTTP hlavička
  - HTTP Basic
  - OAuth 1.0a
  - OAuth 2
- Transformace požadavků a odpovědí - přidávání parametrů či hlaviček skrytých pro konzumenty
- Omezení(throttling) přístupu na API
- API centralizovaný tiketovací systém pro správu požadavků či problémů s API
- Rychlé přepínání testovacích a produkčních prostředí

Monitoring API:

- Sledování počtu dotazů na API a jejich odezvy
- Reporting užití dle konzumentů
- Zobrazování chybových odpovědí od klienta i serveru
- Logování odpovědí pro následnou analýzu problémů mezi API a klientem
- Posílání výstrah při překročení využití nebo při chybách

Dokumentace API:

- Dokumentace pro HTTP metody GET, POST, PUT, PATCH a DELETE
- Možnost vygenerování příkladu odpovědi pomocí reálného zavolání API
- Vygenerování ukázkového příkladu volání API pro cURL a jazyky Java, NodeJs, PHP, Python, Objective-C, Ruby, .NET pomocí knihovny *unirest*[1])
- Možnost dokumentace dodatečných parametrů do url (query string) a HTTP hlavičky - povinné, volitelné, konstantní s typem řetězec (string), číslo (number), logická hodnota (boolean)
- Dokumentace payloadu pro metody POST, PUT

API je možné definovat jako interní nebo veřejné. Veřejné jsou dohledatelné v katalogu a každý uživatel Mashape je může využít (API mohou být i placené). Pro interní API lze poslat přístupové údaje jen konkrétním vývojářům a přidělit jim práva.

Pro zabezpečení nejen interních API posílá Mashape HTTP hlavičku *X-Mashape-Proxy-Secret* s vygenerovaným klíčem dostupným v administračním rozhraní. Producent by tedy vždy měl tento klíč kontrolovat, aby nemohlo dojít k připojení na API na přímo mimo Mashape proxy. Toto bezpečnostní omezení je možné ještě podpořit kontrolou IP adresy, které Mashape využívá a zveřejňuje[2]).

Služba je v omezené funkčnosti poskytována zdarma - např. pro upozornění jsou posílány s 24 hodinovým zpožděním a omezenou podporou. Pro producenty API je navíc dle cenového plánu strhávána část výdělku z placených API (pro verzi zdarma je to 20%, pro placené až 10% z výdělku).

[1]) `http://unirest.io/`
[2]) `http://docs.mashape.com/firewall`

6.5.2 Apigee Edge

Apigee Edge [78] je cloud nástroj pro API management, fungující jako proxy/gateway pro API. Mezi jeho hlavní funkce patří možnost aplikovat politiky[1]) na požadavky i odpovědi. Politiky můžou být synchronní (autentizace, transformace) nebo asynchronní (započítání dotazu do kvóty). Tyto politiky se dají rozdělit do kategorií bezpečnostních a traffic managementu:

Bezpečnostní politiky:

- Autentizace pomocí API klíče (parametr v url nebo HTTP hlavička)
- Autentizace pomocí HTTP Basic
- Autentizace pomocí OAuth v1.0a nebo v2.0
- Autentizace pomocí LDAP (jen pro nástroj Apigee Edge on-premises)
- Access control – povolení či zakázání přístupu k API ze specifikovaných IP adres
- Assign message – úprava požadavků či odpovědí - přidání, odebrání či konverze aktuálních hlaviček, parametrů apod.
- Ochrana proti JSON útokům – validace vstupního JSON na počet elementů, jejich zanoření a omezení délky obsahu
- Ochrana proti XML útokům – validace platného XML dokumentu, validace proti XML schema, omezení počtu elementů, jejich zanoření,
- Validace pomocí regulárního výrazu – umožňuje validovat jednotlivé části požadavků i odpovědí (url parametry, hlavičky, payload, atd.) pomocí regulárních výrazů

Traffic management:

- Cache politika – cachování přímo na proxy, podpora pro uložení, vyhledání a invalidaci cache
- Kvóta na počet dotazů na API za zvolené období (hodina, den, týden, měsíc)
- Kvóta na omezení nárazově velkého množství dotazů[2]) - omezení maximálního počtu dotazů na sekundu či minutu
- Limit na současný počet připojení[3]) k serveru producenta

Apigee také podporuje rozšíření API služeb pomocí programovacích jazyků (JavaScript, Java, Python, Node.js). To umožňuje psát přímo kód v platformě Apigee Edge a tak upravovat chování a funkčnost již hotových API. Také je možné vytvářet díky této službě prototypy budoucího API, např. pomocí frameworků Express[4]) či Argo[5]).

Produkt Apigee Edge je poskytován zdarma pod názvem Apigee Developer [89]. Verze zdarma obsahuje stejné funkcionality jako placená verze, ale bez SLA[6]) a několika dodatečných komponent. Také verze zdarma je dostupná jen v cloudu (placená může být i on-premise) a obsahuje limity pro uložená data a také menší podporu. Placená verze naopak obsahuje možnost monetizace API.

6.6 Hodnocení

Pro porovnání nástrojů jsem vytvořil tabulku 6.1 hodnocenou dle kritérií vytyčených v sekci 6.1.

[1]) `http://apigee.com/docs/api-services/reference/reference-overview-policy`
[2]) Spike Arrest Policy
[3]) Concurrent Rate Limit policy
[4]) `http://expressjs.com/`
[5]) `https://github.com/argo/argo`
[6]) Service-Level Agreement

	Apiaxle	Apiman	Kong	Tyk	Mashape	Apigee
Architektura	Gateway On-Premise	Gateway On-Premise	Gateway On-Premise	Gateway On-Premise	Gateway Cloud	Gateway Cloud
Autentizace	++	++	++	+	++	++
Dokumentace	−−	+	−	+	++	++
Životní cyklus	−	+	−	+	++	++
Monitoring	−−	−−	+	+	++	++
Vysoká dostupnost	−−	+	++	++	++	++
Vývoj a komunita	−−	++	++	+	++	++

Tabulka 6.1. Rešerše nástrojů – přehled hodnocení vybraných nástrojů. Vysvětlení hodnocení: −− žádná nebo hodně slabá podpora, − nedostatečná podpora, + částečná uspokojující podpora, ++ velmi dobrá podpora

Výběr open source nástroje je často závislý na použité technologii, protože se vždy provozuje ve vlastní infrastruktuře. Při jeho instalaci, integraci a běhu je vhodné mít znalosti či lidské zdroje pro podporu dané platfomy nebo programovacího jazyka. Výhodou jsou jiné nástroje či aplikace běžící na obdobných technologiích.

Apiaxle využívá nodejs platformy a je napsaná v Javascriptu. Apiman běží na Wildfly a je napsaný v Javě, Tyk napsaný v Go a Kong v Lua. Všechny tyto nástroje jsou navrhnuty jako proxy a uzpůsobeny pro rychlé zpracování požadavku a provedení potřebných akcí.

Vybrat z těchto nástrojů v této skupině není úplně jednoduché – Apiaxle nemá aktivní vývoj, Tyk je projekt jen jednoho vývojáře, Kong je stále beta a Apiman má ještě málo funkcí. V aktuální době je dle mého názoru nejlepší volba Tyk, u kterého je nejvíce funkcí a nejlepší analytika. Slabší částí je horší uživatelské rozhraní. V budoucnu se bude nejspíš také zajímavý Apiman, který by měl mít podporu pro silný vývoj.

Cloud nástroje jsou zajímavé především pro poskytovatele veřejných API, pro které nabízejí nástroje pro nabízení svých API konzumentům s možností jejich monetizace. Pro obě strany kontraktu API pak nabízejí službu třetí strany, která dělá i technologického prostředníka a umožňuje nezávislý monitoring služeb.

Přestože tyto nástroje obsahují i podporu pro interní API, tak z pohledu požadovaného nástroje nepřináší mnoho výhod. Mezi nevýhody pro interní API patří nutnost jejich vystavení do internetu na adrese třetí služby, což může způsobovat větší latenci než přímé propojení konzumenta a producenta.

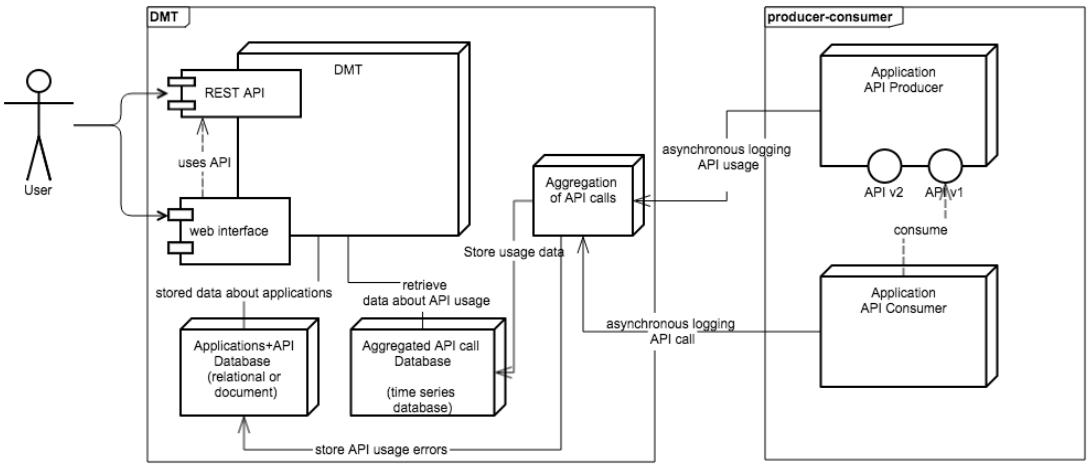
Pro nástroj v této práci nebyl žádný z uvedených nástrojů vhodný k integraci. Hlavním důvodem byla funkčnost všech uvedených nástrojů jako proxy/gateway, což neodpovídá nefunkčím požadavkům (měření konzumování API by nemělo být řešeno přes centrální bod).

Většina těchto nástrojů také řešila problém traffic managementu (omezení počtu požadavků, kvóty apod) a spolehlivé autorizace a autentizace, které nejsou pro požadované řešení tak podstatné, protože producenti budou vystavovat interní API pro interní konzumenty. Pro cílovou skupinu veřejných API jsou ale tyto funkce primární a *asynchronní* model by je jednoduše neimplementoval. Proto jsem se rozhodl navrhnout a implementovat vlastní nástroj.

Kapitola 7
Návrh

V této kapitole představím návrh aplikace dle předchozí analýzy v kapitole 5.
Na obrázku 7.1 jsou zobrazeny hlavní komponenty navrhovaného nástroje:

- **DMT** – document-monitoring tool – požadovaný nástroj, který se sestává z následujících komponent:
  - *Web interface* – webové rozhraní, které pro své data využívá REST API
  - *REST API* – přístup k datům o producentech, konzumentech, API
  - *Notification system* – notifikační systém pro oznamování událostí
- **Applications + API Database** – databáze obsahující data o aplikacích a jejich API, může být relační (např. PostgreSQL) nebo dokumentová (např. MongoDB)
- **Aggregated API call Database** – databáze agregovaných volání API dle času – time series database (např. Graphite nebo InfluxDb)
- **Aggregation of API calls** – komponenta, která asynchronně zpracovává data od producentů a konzumentů o využití daného API a distribuuje data do primární databáze (primárně informace o chybách při volání API) a do agregované databáze (informace o využití API producenta)



Obrázek 7.1. Návrh architektury nástroje

7.1 Návrh DMT

DMT (document-monitoring tool) je centrální komponenta, spojující celý nástroj a všechna data. Jak bylo zmíněno, bude se sestávat z následujících komponent:

- *REST API* – přístup k datům o producentech, konzumentech, API. Pomocí REST API bude aplikace umožňovat interakci. Jeho primárním konzumentem bude vlastní webové rozhraní, což zajistí jeho použitelnost a funkčnost[1]).
- *Web interface* – webové rozhraní, které pro své data využívá REST API. Webové rozhraní bude umožňovat pohledy na aplikaci tak, jak to bylo definováno v sekci 5.2. Mimo tyto statické pohledy na uložená data bude webové rozhraní umožňovat také přidání a editaci dat o jednotlivých aplikacích a datech.
- *Notification system* – notifikační systém pro oznamování událostí vyvolaných akcí v aplikaci (např. nastavení API jako *DEPRECATED*) nebo při výkyvu v užití API.

7.2 Návrh agregátoru

Komponenta agregátoru (na obrázku 7.1 označena jako *Aggregation of API calls*) má dle požadavků přijímat data od producentů a klientů asynchronně. To lze provést následujícími způsoby:

- Aplikace budou ukládat data o využití API do svých log souborů. Agregátor následně tyto log soubory přečte a zpracuje. Tento způsob má výhodu v tom, že asynchronní komunikace je řešena přes zapisování a čtení log souborů a je relativně jednoduchá na implementaci. Nevýhodou je, že každá aplikace nemusí využívat stejný log formát, což může přidávat komplexitu na straně komponenty agregátoru.
- Aplikace bude posílat data do agregátoru (buď přímo, nebo přes frontu zajišťující asynchronnost) pomocí následujících formátů:
  - REST API – pro přijetí dat bude existovat REST API, které zajistí pevnější kontrakt mezi odesílanými a požadovanými daty.
  - UDP protokol – např. pomocí GELF formátu[2]). UDP protokol zaručí, že případný výpadek agregátoru nebude mít vliv na komunikaci konzument producent.

Pro způsob odesílání dat do agregátoru jsem vybral možnost posílání přes REST API, kdy případná asynchronnost bude implementovaná přes jednoduché fronty.

7.3 Návrh získávání definice aplikací

Jak bylo zmíněno v sekci 7.1, tak DMT bude umožňovat přidání dat o aplikaci ručně před webové rozhraní. Tento způsob je vhodný pro přidávání nových API, ale s jejich vývojem se můžou stát tato data rychle neaktuální. Proto je vhodné, umístit tuto dokumentaci co nejblíže kódu:

- Ručně pomocí webového rozhraní
- Automaticky pomocí REST API – generování z kódu (anotace), meta-dokumentace umístěná v repozitáři s kódem

Nástroj bude podporovat editaci dat pomocí webového rozhraní, které bude využívat přímo REST API. Pro automatické zpracování pak bude možné vytvořit skript, který bude načítat data z jiného zdroje a pomocí REST API plnit aplikaci aktuálními daty.

[1]) dogfooding
[2]) https://www.graylog.org/resources/gelf-2/

Kapitola 8
Závěr

První část práce se hodně zabývala teoretickým popisem a úvodem do problematiky REST API i s přihlédnutím na často opomíjená hypermédia. Na základě těchto poznatků byla prezentována metodika pro návrh REST API. Druhá část práce se zaměřovala na samotný požadovaný systém pro sledování interakcí mezi aplikacemi.

8.1 Zhodnocení splnění cílů DP, doporučení dalšího pokračování práce

V práci byly vytyčeny následující cíle:

- **Vypracujte studii existujících nástrojů, které řízení provázanosti a dokumentace API podporují.**

  Rešerši existujících nástrojů jsem popsal v kapitole 6. Hlavní pozornost jsem věnoval open-source nástrojům zdarma, které by mohli být využitelné nebo integrovatelné do výsledného nástroje. Samotných nástrojů pro API management je velké množství, avšak všechny popisované využívaly architekturou gateway/proxy a nesplnili nefunkční požadavek o nepoužití centrálního bodu pro nástroj. Popis formátů a nástrojů pro podporu dokumentace API byl popsán v kapitole 3.5.
- **Vypracujte metodiku pro návrh a dokumentaci API - zejména strukturu a verzování.**

  Metodika pro návrh a dokumentaci API byla popsáná v kapitole 4. Metodika shrnuje podstatné části prezentované v kapitole 3, která popisovala základy architektonického stylu REST, popis sémantiky HTTP a popis formátů používaných pro REST API. Metodika je doplněna i o praktické rady.
- **Navrhněte systém pro monitorování a dokumentaci API v rámci celého vývojářského ekosystému.**

  V kapitole 5 byla provedena analýza zadaných user stories, ze kterých byly identifikovány základní úkoly pro jejich splnění. V kapitole 7 byl přestaven návrh, podle kterého vznikne budoucí nástroj.
- **Diskutujte a zvolte vhodné implementační prostředky - pro dílčí úkoly lze použít hotové moduly, pokud existují.**
- **Systém implementujte, nasaďte, zdokumentujte a na zvolených aplikacích a jejich API dokumentujte užitečnost vyvinutého systému.**

  Poslední dva body se bohužel v době odevzdání této práce nepodařilo splnit, a proto cílem v dalším pokračování práce je dodělat a implementovat navržený nástroj pro praktické využití.

8.2 Shrnutí vlastního přínosu DP

Vlastní přínos DP spatřuji především v detailnějším prozkoumání bibliografie o REST API, které mi osvětlilo ještě více problematiku, včetně hypermédií. V teoretické části

jsem prozkoumal formáty pro popis a definici API (API Blueprint, RAML, Swagger) a také objevil média typy s podporou hypermédií (HAL, Siren), které budu využívat při tvorbě REST API. Při rešerši nástrojů pro API management jsem zjistil, že nástrojů okolo API ekosystému existuje obrovské množství. Pro tuto práci žádný nástroj nevyhovoval požadavkům, ale chtěl bych je vyzkoušet na jiné případy – např. Apigee nebo Mashape pro správu veřejných API.

Literatura

[1] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. Disertační práce, University of California, Irvine. 2000.

[2] Tim Berners-Lee. *WorldWideWeb: Proposal for a HyperText Project*. `http://www.w3.org/Proposal.html`. Citováno 15.4. 2015.

[3] Zuzana Šochová. *Proč píšeme User Story?* `http://soch.cz/blog/management/agile/proc-piseme-user-story/`. Citováno 28.3. 2015.

[4] Jiří Knesl. *Návod na User Stories*. `http://soch.cz/blog/management/agile/proc-piseme-user-story/`. Citováno 28.3. 2015.

[5] Michael Hüttermann. *DevOps for developers*. Apress, 2012.

[6] Daniel Jacobson, Greg Brail a Dan Woods. *APIs: A Strategy Guide*. O'Reilly Media, Inc., 2011. ISBN 1449308929, 9781449308926.

[7] Steven Willmott. *Public vs Private vs Internal APIs*. `http://www.3scale.net/2015/02/public-vs-private-vs-internal-apis/`. Citováno 15.4. 2015.

[8] Mark O'Neill. *API Axes - Categorizing APIs*. `http://www.soatothecloud.com/2015/02/this-week-there-has-been-great.html`. Citováno 15.4. 2015.

[9] *ProgrammableWeb API Directory*. `http://www.programmableweb.com/apis/directory`. Citováno 8.3. 2015.

[10] *Public APIs*. `http://www.publicapis.com/`. Citováno 8.3. 2015.

[11] *APIs.io*. `http://apis.io/`. Citováno 8.3. 2015.

[12] *APIs.json*. `http://apisjson.org/`. Citováno 8.3. 2015.

[13] Martin Lewis, James a Fowler. Microservices. *Online na* `http://martinfowler.com/articles/microservices.html`. 2014, Citováno 30.3. 2015.

[14] Gregory R Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys (CSUR)*. 1991, 23 (1), 49–90.

[15] *W3C: Architecture of the World Wide Web, Volume One*. `http://www.w3.org/TR/2004/REC-webarch-20041215/`. Citováno 30.3. 2015.

[16] "T. Berners-Lee, R. Fielding a L. Masinter". *"Uniform Resource Identifier (URI): Generic Syntax"*. "RFC 3986 (INTERNET STANDARD)". "Request for Comments". 2005. "Updated by RFCs 6874.

[17] Leonard Richardson, Mike Amundsen a Sam Ruby. *RESTful Web APIs*. O'Reilly Media, Inc., 2013.

[18] "R. Fielding a J. Reschke". *"Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content"*. "RFC 7231 (Proposed Standard)". "Request for Comments". 2014. `"http://www.ietf.org/rfc/rfc7231.txt"`.

[19] "L. Dusseault a J. Snell". *"PATCH Method for HTTP"*. "RFC 5789 (Proposed Standard)". "Request for Comments". 2010. `"http://www.ietf.org/rfc/rfc5789.txt"`.

[20] Roy T Fielding. REST APIs must be hypertext-driven. *Untangled musings of Roy T. Fielding, Online na http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven.*

[21] "M. Nottingham". *"Web Linking"*. "RFC 5988 (Proposed Standard)". "Request for Comments". 2010. `"http://www.ietf.org/rfc/rfc5988.txt"`.

[22] *Act Three: The Maturity Heuristic*. Prezentace na QCon v roce 2008, online na `http://www.crummy.com/writing/speaking/2008-QCon/act3.html`. Citováno 30.3. 2015.

[23] Martin Fowler. Richardson Maturity Model: steps toward the glory of REST. *Online na http://martinfowler.com/articles/richardsonMaturityModel.html*. 2010, Citováno 30.3. 2015.

[24] *APIs.json 0.14 Draft for Comment*. `http://apisjson.org/format/apisjson_0.14.txt`. Citováno 8.3. 2015.

[25] *Web Application Description Language*. `http://www.w3.org/Submission/2009/SUBM-wadl-20090831/`. Citováno 24.4. 2015.

[26] *API Blueprint*. `https://apiblueprint.org/`. Citováno 30.4. 2015.

[27] *Apiary*. `https://apiary.io/`. Citováno 30.4. 2015.

[28] *Markdown*. `http://daringfireball.net/projects/markdown/`. Citováno 30.4. 2015.

[29] *API Blueprint*. `https://github.com/apiaryio/api-blueprint/blob/master/API%20Blueprint%20Specification.md`. Citováno 10.5. 2015.

[30] *Dredd — HTTP API Validation Tool*. `https://github.com/apiaryio/dredd`. Citováno 10.5. 2015.

[31] *Drakov*. `https://www.npmjs.com/package/drakov`. Citováno 10.5. 2015.

[32] *APIMATIC*. `https://apimatic.io/`. Citováno 10.5. 2015.

[33] *Swagger*. `http://swagger.io/`. Citováno 11.3. 2015.

[34] *Swagger RESTful API Documentation Specification, Version 2.0*. `https://github.com/swagger-api/swagger-spec/blob/master/versions/2.0.md`. Citováno 11.5. 2015.

[35] *Swagger for Express and Node.js*. `https://www.npmjs.com/package/swagger-node-express`. Citováno 11.3. 2015.

[36] *Swagger generated server*.
https://github.com/swagger-api/swagger-codegen/tree/master/samples/server-generator/node. Citováno 11.3. 2015.

[37] *Swagger UI*.
https://github.com/swagger-api/swagger-ui. Citováno 11.3. 2015.

[38] *swagger-codegen*.
https://github.com/swagger-api/swagger-codegen. Citováno 11.3. 2015.

[39] *RESTful API Modeling Language*.
http://raml.org/. Citováno 12.3. 2015.

[40] *RAML IS TURNING 1.0!*
http://blog.raml.org/raml-is-turning-1-0/. Citováno 14.3. 2015.

[41] *RAML Version 0.8: RESTful API Modeling Language*.
https://github.com/raml-org/raml-spec/blob/master/raml-0.8.md. Citováno 19.5. 2015.

[42] *RAML Parser*.
https://github.com/raml-org/raml-js-parser. Citováno 14.3. 2015.

[43] *RAML Java Parser*.
https://github.com/raml-org/raml-java-parser. Citováno 14.3. 2015.

[44] *pyraml-parser*.
https://github.com/an2deg/pyraml-parser. Citováno 14.3. 2015.

[45] *RAML ruby*.
https://github.com/coub/raml_ruby. Citováno 14.3. 2015.

[46] *RAML Editor*.
https://github.com/mulesoft/api-designer. Citováno 14.3. 2015.

[47] *RAML Console*.
https://github.com/mulesoft/api-console. Citováno 14.3. 2015.

[48] *RAML Console*.
http://apinotebook.com/. Citováno 14.3. 2015.

[49] Lane Kin. *API Design: Do You Swagger, Blueprint or RAML?*
http://apievangelist.com/2014/01/16/api-design-do-you-swagger-blueprint-or-raml/. Citováno 18.5. 2015.

[50] Mike Stowe. *RAML vs. Swagger vs. API Blueprint*.
http://www.mikestowe.com/2014/07/raml-vs-swagger-vs-api-blueprint.php. Citováno 18.5. 2015.

[51] Ole Lensmar. *Another API-Blueprint, RAML and Swagger Comparison*.
http://www.slideshare.net/SmartBear_Software/api-strat-2014metadataformatsshort. Citováno 18.5. 2015.

[52] Orlando Kalossakas. *Which API editor to use?*
https://medium.com/@orliesaurus/a-review-of-all-most-common-api-editors-6a720dc4f4e6. Citováno 18.5. 2015.

[53] *Mulesoft Anypoint Platform*.
https://www.mulesoft.com/platform/enterprise-integration. Citováno 14.5. 2015.

[54] "D. Crockford". *"The application/json Media Type for JavaScript Object Notation (JSON)"*. "RFC 4627 (Informational)". "Request for Comments". 2006. "http://www.ietf.org/rfc/rfc4627.txt". "Obsoleted by RFC 7159".

[55] *Home Documents for HTTP APIs*.
http://tools.ietf.org/html/draft-nottingham-json-home-03. Citováno 24.4. 2015.

[56] *JSend*.
http://labs.omniti.com/labs/jsend. Citováno 1.3. 2015.

[57] *NPM jsend*.
https://www.npmjs.com/package/jsend. Citováno 1.3. 2015.

[58] *python-jsend*.
ttps://pypi.python.org/pypi/python-jsend. Citováno 1.3. 2015.

[59] *jsend-rails*.
https://rubygems.org/gems/jsend-rails. Citováno 1.3. 2015.

[60] *Problem Details for HTTP APIs*.
https://tools.ietf.org/html/draft-nottingham-http-problem-07. Citováno 19.5. 2015.

[61] *HAL – Hypertext Application Language specification*.
http://stateless.co/hal_specification.html. Citováno 1.3. 2015.

[62] *Github HAL – Hypertext Application Language Libraries*.
https://github.com/mikekelly/hal_specification/wiki/Libraries. Citováno 1.3. 2015.

[63] *Siren: a hypermedia specification for representing entities*.
https://github.com/kevinswiber/siren. Citováno 21.5. 2015.

[64] "J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen a L. Stewart". *"HTTP Authentication: Basic and Digest Access Authentication"*. "RFC 2617 (Draft Standard)". "Request for Comments". 1999.
"http://www.ietf.org/rfc/rfc2617.txt". "Updated by RFC 7235".

[65] "H. Krawczyk, M. Bellare a R. Canetti". *"HMAC: Keyed-Hashing for Message Authentication"*. "RFC 2104 (Informational)". "Request for Comments". 1997.
"http://www.ietf.org/rfc/rfc2104.txt". "Updated by RFC 6151".

[66] *OAuth*.
http://oauth.net/. Citováno 31.5. 2015.

[67] "E. Hammer-Lahav". *"The OAuth 1.0 Protocol"*. "RFC 5849 (Informational)". "Request for Comments". 2010.
"http://www.ietf.org/rfc/rfc5849.txt". "Obsoleted by RFC 6749".

[68] "D. Hardt". *"The OAuth 2.0 Authorization Framework"*. "RFC 6749 (Proposed Standard)". "Request for Comments". 2012.
"http://www.ietf.org/rfc/rfc6749.txt".

[69] Prabath Siriwardena. *Advanced API Security: Securing APIs with OAuth 2.0, OpenID Connect, JWS, and JWE*. Apress, 2014.

[70] "R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach a T. Berners-Lee". *"Hypertext Transfer Protocol – HTTP/1.1"*. "RFC 2616 (Draft Standard)". "Request for Comments". 1999. "Obsoleted by RFCs 7230.

[71] Randy Heffner. *The Forrester Wave™: API Management Solutions, Q3 2014*.

[72] Kai Wähner. *A New Front for SOA: Open API and API Management as Game Changer*. 2014. Citováno 7.3. 2015.

[73] *Apiaxle*.
http://apiaxle.com/. Citováno 9.4. 2015.

[74] *Apiman*.
http://www.apiman.io/. Citováno 9.4. 2015.

[75] *Kong*.
http://getkong.org/. Citováno 10.4. 2015.

[76] *Tyk*.
https://tyk.io/. Citováno 9.4. 2015.

[77] *Mashape*.
https://www.mashape.com/. Citováno 7.3. 2015.

[78] *Apigee Edge*.
http://apigee.com/about/products/api-management. Citováno 7.3. 2015.

[79] *Mashery Intel Services*.
http://www.mashery.com/. Citováno 10.4. 2015.

[80] *Informatica iPaaS*.
https://www.informatica.com/products/integration-platform-as-a-service.html. Citováno 10.4. 2015.

[81] *CA API Management*.
http://www.ca.com/us/products/api-management/solutions/api-management-solutions.aspx. Citováno 10.4. 2015.

[82] *TIBCO Api Exchange Gateway*.
http://www.tibco.com/products/automation/application-development/soa-governance/api-gateway. Citováno 10.4. 2015.

[83] *IBM Api Management*.
https://apim.ibmcloud.com/. Citováno 10.4. 2015.

[84] *Axway Api Management*.
https://www.axway.com/en/enterprise-solutions/api-management. Citováno 10.4. 2015.

[85] *Apiaxle architecture*.
http://apiaxle.com/docs/architecture/. Citováno 9.4. 2015.

[86] *ApiAxle has been acquired*.
http://apiaxle.com/docs/acquisition-statement/. Citováno 9.4. 2015.

[87] *Exicon ApiAxle*.
http://www.exiconglobal.com/apiaxle/. Citováno 9.4. 2015.

[88] Markus Eisele. *Kickstart on API Management with JBoss Apiman 1.0*.
http://blog.eisele.net/2015/01/kickstart-on-api-management-with-jboss-apiman.html. Citováno 9.4. 2015.

[89] *Apigee Developer*.
http://apigee.com/docs/developer-vs-edge. Citováno 7.3. 2015.

Příloha A
Zadání práce

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Dokumentačně-monitorovací systém pro správu prostředí s mnoha aplikacemi
Student:	Bc. Jakub Šimon
Vedoucí:	Ing. Ondřej Mysliveček
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2015/16

Pokyny pro vypracování

Moderní SW aplikace jsou často velmi provázané pomocí API. Tato provázanost přináší i problémy a rizika v případě změn API některé z nich.
Cílem práce je návrh metodiky vývoje a dokumentace API v SW ekosystému tak, aby bylo možné jejich provázanost sledovat a řídit. Dále pak návrh, implementace a nasazení nástroje pro sledování provázanosti API aplikací v SW ekosystému.
1. Vypracujte studii existujících nástrojů, které řízení provázanosti a dokumentace API podporují.
2. Vypracujte metodiku pro návrh a dokumentaci API - zejména strukturu a verzování.
3. Navrhněte systém pro monitorování a dokumentaci API v rámci celého vývojářského ekosystému.
4. Diskutujte a zvolte vhodné implementační prostředky - pro dílčí úkoly lze použít hotové moduly, pokud existují.
5. Systém implementujte, nasaďte, zdokumentujte a na zvolených aplikacích a jejich API dokumentujte užitečnost vyvinutého systému.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof.Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 13. února 2015

Příloha B
Slovníček

API	Application Programming Interface
ČVUT	České vysoké učení technické v Praze
HA	High availability
HAL	Hypertext Application Language
HATEOAS	Hypertext As The Engine Of Application State
HMAC	Keyed-hash Message Authentication Code
HTTP	Hypertext Transfer Protocol
IANA	Internet Assigned Numbers Authority
JMS	Java Messaging Services
JSON	JavaScript Object Notation
RAML	RESTful API Modeling Language
REST	Representational State Transfer
RFC	Request for Comments
RMM	Richardson Maturity Model
SaaS	Software as a Service
SLA	Service-Level Agreement
SMTP	Simple Mail Transfer Protocol
WADL	Web Application Description Language
WSDL	Web Service Description Language
YAML	YAML Ain't Markup Language

Příloha C
User stories

User story C.1. As a developer I want to see APIs my apps provide that are no longer used by any consumers so that I can get rid of old useless code.
User story C.2. As a developer I want to clearly see the role of the system I'm applying my clumsy patches to in the application neighbourhood so that I can analyse impacts of my changes to those APIs easily in advance and maybe discuss the changes with impacted teams/developers
User story C.3. As a developer / DEVOP I want to know the performance of APIs my apps provide so that I can ensure required quality of service.
User story C.4. As a developer I want to be informed when any API my apps consume becomes deprecated so that I can rewrite the app to use supported versions (or solve this future problem differently when there is no supported version available).
User story C.5. As a developer I need to know how reliable the API I am about to consume is so that I can design my app for maximum reliability and uptime.
User story C.6. As a developer I need detailed descriptions of all available APIs my apps can use in one place, so that I know where to start when searching for information I need and when I'm just looking what I could possible use for new features (so that I won't invent a wheel again - I have invented lots of them already).
User story C.7. As an architect I want to see changes in usage intensity of APIs and get alerts when something behaves differently so that I can identify bugs / problems and plan system resources based on long-term trends.

Příloha D
Ukázky formátů

D.1 WADL

Ukázka WADL, zdroj [25]:

```
⟨?xml version="1.0"?⟩
⟨application ... xsi:schemaLocation="http://wadl.dev.java.net/2009/02 ... ⟩
  ⟨grammars⟩
    ⟨include href="NewsSearchResponse.xsd"/⟩
    ⟨include href="Error.xsd"/⟩
  ⟨/grammars⟩
  ⟨resources base="http://api.search.yahoo.com/NewsSearchService/V1/"⟩
    ⟨resource path="newsSearch"⟩
      ⟨method name="GET" id="search"⟩
        ⟨request⟩
          ⟨param name="appid" type="xsd:string" style="query" required="true"/⟩
          ⟨param name="query" type="xsd:string" style="query" required="true"/⟩
          ⟨param name="type" style="query" default="all"⟩
            ⟨option value="all"/⟩
            ⟨option value="any"/⟩
            ⟨option value="phrase"/⟩
          ⟨/param⟩
          ⟨param name="results" style="query" type="xsd:int" default="10"/⟩
          ⟨param name="start" style="query" type="xsd:int" default="1"/⟩
          ⟨param name="sort" style="query" default="rank"⟩
            ⟨option value="rank"/⟩
            ⟨option value="date"/⟩
          ⟨/param⟩
          ⟨param name="language" style="query" type="xsd:string"/⟩
        ⟨/request⟩
        ⟨response status="200"⟩
          ⟨representation mediaType="application/xml" element="yn:ResultSet"/⟩
        ⟨/response⟩
        ⟨response status="400"⟩
          ⟨representation mediaType="application/xml" element="ya:Error"/⟩
        ⟨/response⟩
      ⟨/method⟩
    ⟨/resource⟩
  ⟨/resources⟩
⟨/application⟩
```

D.2 JSON Home

Ukázka JSON Home, zdroj [55]

```
GET / HTTP/1.1
Host: example.org
Accept: application/json-home

HTTP/1.1 200 OK
Content-Type: application/json-home
Cache-Control: max-age=3600
Connection: close

{
  "resources": {
    "http://example.org/rel/widgets": {
      "href": "/widgets/"
    },
    "http://example.org/rel/widget": {
      "href-template": "/widgets/{widget_id}",
      "href-vars": {
        "widget_id": "http://example.org/param/widget"
      },
      "hints": {
        "allow": ["GET", "PUT", "DELETE", "PATCH"],
        "formats": {
          "application/json": {}
        },
        "accept-patch": ["application/json-patch"],
        "accept-post": ["application/xml"],
        "accept-ranges": ["bytes"]
      }
    }
  }
}
```

D.3 Problem Details for HTTP APIs

Ukázka Problem Details for HTTP APIs, zdroj [60]

```
HTTP/1.1 403 Forbidden
Content-Type: application/problem+json
Content-Language: en

{
 "type": "http://example.com/probs/out-of-credit",
 "title": "You do not have enough credit.",
 "detail": "Your current balance is 30, but that costs 50.",
 "instance": "http://example.net/account/12345/msgs/abc",
 "balance": 30,
 "accounts": ["http://example.net/account/12345",
              "http://example.net/account/67890"]
}
```

D.4 Siren

Ukázka Siren formátu, zdroj [63]

```
{
  "class": [ "order" ],
  "properties": {
      "orderNumber": 42,
      "itemCount": 3,
      "status": "pending"
  },
  "entities": [
    {
      "class": [ "items", "collection" ],
      "rel": [ "http://x.io/rels/order-items" ],
      "href": "http://api.x.io/orders/42/items"
    },
    {
      "class": [ "info", "customer" ],
      "rel": [ "http://x.io/rels/customer" ],
      "properties": {
        "customerId": "pj123",
        "name": "Peter Joseph"
      },
      "links": [
        { "rel": [ "self" ], "href": "http://api.x.io/customers/pj123" }
      ]
    }
  ],
  "actions": [
    {
      "name": "add-item",
      "title": "Add Item",
      "method": "POST",
      "href": "http://api.x.io/orders/42/items",
      "type": "application/x-www-form-urlencoded",
      "fields": [
        { "name": "orderNumber", "type": "hidden", "value": "42" },
        { "name": "productCode", "type": "text" },
        { "name": "quantity", "type": "number" }
      ]
    }
  ],
  "links": [
    { "rel": [ "self" ], "href": "http://api.x.io/orders/42" },
    { "rel": [ "previous" ], "href": "http://api.x.io/orders/41" },
    { "rel": [ "next" ], "href": "http://api.x.io/orders/43" }
  ]
}
```

Příloha E
Obsah přiloženého média

Přiložené médium obsahuje následující strukturu:

- **ctustyle-simonja3.pdf** – pdf soubor této práce
- **source** – adresář obsahující všechny zdrojové soubory textu této práce