VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Rámec pro přístup ke vzdáleným zdrojům s podporou samodokumentujícího se API

Framework for Access to Remote Resources with Self-documented API Support

2017 Bc. Jakub Skokan

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jakub Skokan**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Rámec pro přístup ke vzdáleným zdrojům s podporou samodokumentujícího se API
Framework for Access to Remote Resources with Self-documented API Support

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je vytvořit rámec pro vzdálený přístup ke zdrojům, který umožní klientovi získat informace o nabízených zdrojích a operacích, které je možné s nimi provádět. Na základě těchto informací pak může klient s danými zdroji pracovat.

Protokoly budou podporovat:
1. Verzování zdrojů (jejich popis a poskytovanou podobu).
2. Dokumentaci objektů, akcí, parametrů, validátorů, ukázek použití, významů parametrů.
3. Autentizační metody.
4. Postranní kanál pro metadata (např. počty objektů, volba eager/lazy načítání asociovaných objektů).
5. Volitelný blokující/neblokující mód volání akcí.
6. Asociaci mezi objekty (n:1) a možnost načítání asociovaných objektů (volby eager a lazy).
7. Podporu formátování dat pomocí JSON.
8. Přenos dat.
9. Výsledné API bude následovat REST sémantiku.

Práce bude obsahovat:
1. Popis protokolu pro přenos dat.
2. Popis dokumentačního protokolu.
3. Implementaci serverové části rámce, která bude:
a) implementovat specifikované protokoly,
b) generovat online dokumentace vytvořeného API,
c) umožňovat napojení poskytovaných zdrojů na ORM rámec,
d) umožňovat řízení oprávnění přístupu k objektům (filtrování parametrů).
4. Implementaci klientské části rámce, která bude:
a) implementovat specifikované protokoly v různých programovacích jazycích,
b) obsahovat přehlednou dokumentaci každého klienta,
c) obsahovat administrační rozhraní pro procházení objektů jakéhokoliv API, které implementuje specifikovaný protokol.
5. Testování se zaměřením na porovnání výkonu API využívající referenční implementaci protokolu a jiných řešení.

Seznam doporučené odborné literatury:

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Gang of Four): Návrh programů pomocí vzorů. Grada. Praha 2003. ISBN 8024703025
[2] DARWIN, Ian F. Java cookbook. 2nd ed. Sebastopol, CA: O'Reilly, c2004, xxiv, 829 p. ISBN 05-960-0701-9. Dostupné z: http://it-ebooks.info/book/2249/

Dále podle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. David Ježek, Ph.D.**

Datum zadání: 01.09.2015

Datum odevzdání: 28.04.2017

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 24. dubna 2017 .....

Rád bych na tomto místě poděkoval všem, kteří mi s prací pomohli, zejména kolegům ze spolku vpsFree.cz a vedoucímu práce za jejich podporu.

Abstrakt

Tato práce se zabývá návrhem protokolu pro dokumentování serverových API s využitím protokolu *HTTP* a architektury *REST*. Součástí projektu je referenční implementace protokolu ve formě frameworku sloužícího k tvorbě *API* serveru, který je pomocí reflexe a doménového jazyka schopen výsledné *API* sám zdokumentovat a tuto dokumentaci předat klientům ve strojově čitelné formě. Referenční implementace dále obsahuje klientské knihovny v několika programovacích jazycích. Nad těmito knihovnami je pak postaveno uživatelské rozhraní ve formě webové administrace, rozhraní v příkazové řádce a virtuálního souborového systému. Každý klient a nad ním postavená aplikace dokáže pracovat s jakýmkoli *API*, které implementuje definovaný protokol.

Klíčová slova: API, klient, server, framework, protokol, REST, HTTP, web, služba

Abstract

The goal of this thesis is to design a protocol to document server APIs with the use of the *HTTP* protocol and *REST* architecture. Included is a reference implementation of said protocol in the form of a framework that can be used to create server *API*. The framework uses reflection and a domain-specific language to document itself and is able to pass this documentation in a machine-readable form to clients. The reference implementation also contains client libraries in several programming languages. There are three user interfaces built on top of these libraries: a web administration, a command-line interface and a virtual file system. All clients and applications can be used with any *API* that implements the created protocol.

Key Words: API, client, server, framework, protocol, REST, HTTP, web, service

Obsah

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface
BDD	–	Behaviour-driven Development
CLI	–	Command Line Interface
DSL	–	Domain-specific language
FUSE	–	Filesystem in Userspace
HAL	–	Hypertext Application Language
JSON	–	JavaScript Object Notation
ORM	–	Object-relational mapper
REST	–	Representational state transfer
SPA	–	Single Page Application
URL	–	Unified Resource Location

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Jednou z možností poskytování portálů, informačních systémů nebo jiných služeb je architektura klient-server, na které je založen například dnešní web. Klient se serverem komunikuje přes počítačovou síť a musí si navzájem rozumět, tzn. používat nějaký protokol. Na webu je tímto protokolem *HTTP* [1]. Jak jsou data přenášeny a jak jsou strukturovány už závisí na kontraktu mezi klientem a serverem. Tato práce se zaobývá pasivními servery poskytujícími *API* a aktivní klienty. Server tedy čeká na požadavek od klienta přes dané *API* a zašle příslušnou odpověď.

Architektura klient-server nám např. umožňuje oddělit aplikační logiku od uživatelských rozhraní. Pokud aplikační logiku umístíme za server *API*, různé uživatelské rozhraní můžeme implementovat jako klienty využívající *API* serveru.

Náplní této práce je vytvořit protokol nad *HTTP*, který by byl schopen zobecnit komunikaci mezi klientem a serverem tak, aby bylo možné implementovat klienta, který bude schopen pomocí *API* komunikovat s jakýmkoli serverem implementující daný protokol. Současně přitom vyřešíme i problém udržování a generování dokumentace *API* serveru pro jeho uživatele.

V sekci 2 jsou podrobněji popsány problémy, které se tato práce snaží řešit a v sekci 3 následuje navržené řešení. Protokol, který byl v rámci této práce vytvořen, je specifikován v sekci 4. Referenční implementace serverové části je popsána v sekci 6 a jednotlivé klientské knihovny a aplikace potom v sekci 7. Výsledky výkonnostního testování jednotlivých implementací jsou znázorněny v sekci 8 a závěr a informace o použití v praxi v sekci 9.

2 Analýza problému

Při tvorbě *API* serveru musíme pokaždé opakovat několik kroků:

- implementace *API* serveru a vlastní aplikační logiky,
- dokumentace *API* pro uživatele,
- knihovny/aplikace pro práci s *API*.

Musíme se rozhodnout, jak bude vypadat rozhraní, přes které budou s *API* komunikovat uživatelé. Ať už zvolíme jakékoli rozhraní, máme definován jen způsob, jakým se přenáší data. Uživatelé však potřebují ještě informaci o tom, jaké data se přenášejí, jak jsou strukturovaná a jaký je jejich význam. K tomuto slouží dokumentace, kde popíšeme rozhraní pro přenos dat a možný obsah. Pokud je implementace *API* a dokumentace oddělená, dochází zde k duplicitám: struktura *API* je implementována a pak popsána v dokumentaci. Protože dokumentace není přímo svázaná se samotnou implementací *API*, může dojít ke vzniku nekompatibility mezi implementací a dokumentací.

Existence knihoven pro práci s *API* není vždy nezbytná a je možno tuto část přenechat uživatelům *API*, ale pro snadnost použití *API* je vhodné je mít k dispozici. Klientské knihovny jsou také odděleny od *API* serveru a je nutné je udržovat kompatibilní.

První problém, který se tato práce pokouší řešit, je specifikace protokolu pro popis struktury přenášených dat mezi *API* a jeho uživatelem přes definované rozhraní. S tím souvisí možnost vytvoření generických klientských knihoven a aplikací, které budou schopny pracovat s jakýmkoli serverovým *API*, které je schopno popsat svou strukturu pomocí zvoleného protokolu.

Druhý problém je udržování konzistence implementace *API* a jeho dokumentace, tzn. pokud dojde ke změně *API* serveru, která má vliv na funkčnost klientů, měla by se automaticky projevit i v dokumentaci, aniž by to znamenalo práci navíc.

Podle architektury *REST* mohou odpovědi ze serveru obsahovat odkazy na další relevantní zdroje a akce v *API*. Tento přístup rozvíjí např. *HAL* [5], který specifikuje umístění a formu odkazů v reprezentaci zdroje. Samodokumentováním v tomto stylu však klient nemá o *API* všechny informace hned, ale až když vyvolává akce. Cílem této práce je navrhnout protokol tak, aby byla klientům k dispozici kompletní dokumentace *API* ještě před tím, než s ním začnou pracovat. Díky tomu je pak možné generovat ucelené uživatelské rozhraní či zdrojové kódy klientských programů.

3 Navržené řešení

Architektura *REST* [4] definuje, jak by mělo vypadat *API* serveru. Už však neříká nic o tom, jak by měly vypadat přenášené data. *REST* tedy využijeme jako rozhraní pro komunikaci mezi *API* a jeho uživateli a přidáme možnost přenosu dokumentace struktury *API*.

Pro tento účel vznikl projekt *HaveAPI*. Cílem projektu je poskytnout způsob pro přenos dokumentace ve strojově čitelné formě ke klientům, kteří z ní vyčtou vše potřebné k tomu, aby mohli pracovat s jakýmkoli serverovým *API*, které umí poskytovat dokumentaci definovaným způsobem.

Definice protokolu pro formu a přenos dokumentace by samo o sobě nebylo užitečné, neboť by každý musel kromě samotného *API* implementovat také protokol pro přenos dokumentace a její vyhodnocení na straně klienta. Proto *HaveAPI* obsahuje referenční implementaci serverové i klientské části, které použití značně usnadňují.

HaveAPI se skládá z těchto částí:

- Definice struktury dokumentace a protokol pro její přenos
- Definice struktury a přenosu vlastních dat (volání akcí v *API*)
- Referenční implementace *API* serveru ve formě frameworku
- Referenční implementace klientů v různých programovacích jazycích ve formě knihoven
- Uživatelské rozhraní k *API* využívající klientské knihovny

Serverová část je implementována ve formě frameworku, tak aby pomocí něj šlo vytvářet různé *API* servery, které automaticky umí generovat dokumentaci a předávat ji klientům pomocí definovaného protokolu. Tím je tedy vyřešen druhý zmíněný problém, a sice aby ona poskytnutá dokumentace přímo odpovídala aktuální implementaci *API* serveru. To *HaveAPI* řeší tak, že dokumentaci generuje za běhu přímo ze zdrojového kódu *API* serveru.

Pokud zůstane protokol nezávislý na refenční implementaci, mohou vznikat i alternativní implementace serverové části a být vzájemně kompatibilní s existujícími klienty.

4 Specifikace protokolu

HaveAPI využívá komunikační protokol *HTTP* s architekturou *REST*.

Dokumentace se klientům předává pomocí *HTTP* metody *OPTIONS* formátovaná v *JSON* [6], jehož struktura je popsána níže.

4.1 Verzování protokolu

Verze protokolu udává podobu

- obálky (popsáno v sekci 4.2),
- struktury dokumentace,
- způsobu přenosu dokumentace.

Verze se skládá ze dvou čísel oddělených tečkou, `<MAJOR>.<MINOR>`. `MAJOR` se musí zvýšit pokaždé, když dojde ke zpětně nekompatibilní změně protokolu. `MINOR` se zvyšuje, pokud bylo do protokolu přidáno něco nového, ale zpětnou kompatibilitu to nenarušuje, tzn. na existující klienty to nemá vliv.

Verze protokolu je součástí obálky zaslané klientovi při vyžádání dokumentace.

Pokud se liší `MAJOR` verze *API* serveru a klienta, klient by měl skončit s chybou. Pokud se liší `MINOR`, klient může bezpečně fungovat, jen když podporuje nižší `MINOR`, než-li *API* server. V opačném případě končí s chybou.

4.2 Obálka

Každá odpověď z *API* serveru je zaslána v obálce, viz diagram 1.

Obsah obálky:

`version` verze protokolu, zasílá se jen spolu s dokumentací

`status` udává, zda byla akce úspěšná, nebo došlo k chybě

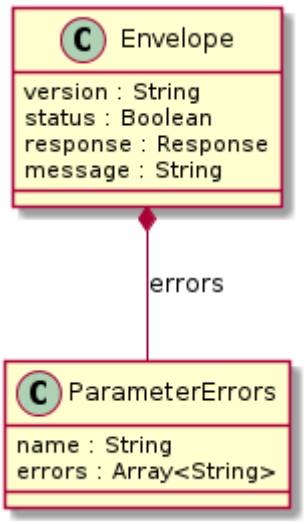
`response` vlastní odpověď (výstupní parametry akce)

`message` chybová hláška, dojde-li k chybě

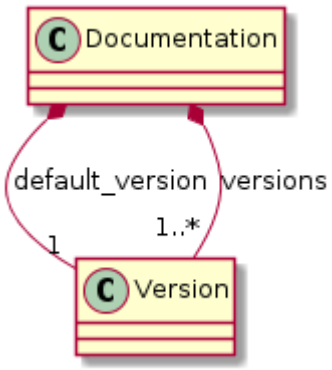
`errors` chybové hlášky pro vstupní parametry

4.3 Verze API

API server může v jednu chvíli podporovat více verzí *API*, které mohou obsahovat jiné autentizační metody, objekty, akce, parametry, apod. Všechny verze *API* však musí používat stejnou verzi protokolu.



Obrázek 1: Obálka



Obrázek 2: Dokumentace všech verzí *API*

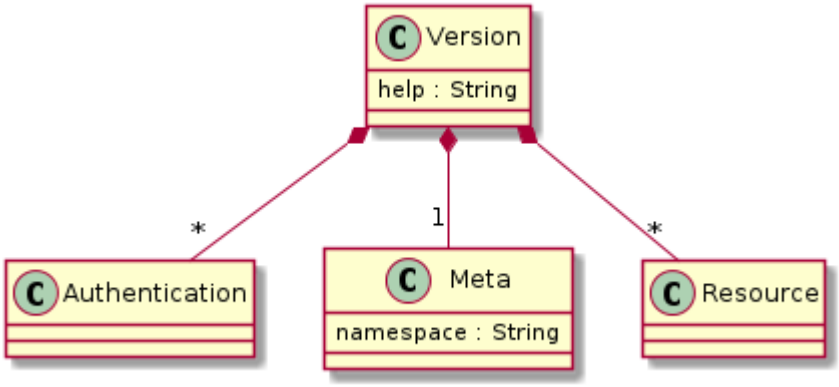
Klient si může vyžádat buď dokumentaci celého *API* (diagram 2), včetně všech jeho verzí, nebo dokumentaci jedné konkrétní verze *API* (diagram 3).

Níže popisované součásti dokumentace se týkají vždy jedné konkrétní verze API. Mohou se tedy vyskytovat vícekrát v různých verzích.

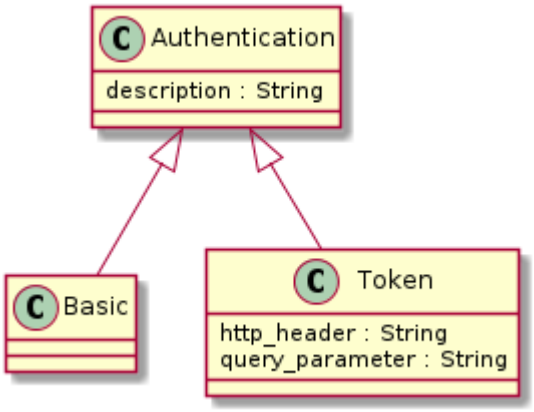
4.4 Autentizace

HaveAPI definuje dvě autentizační metody 4: *HTTP basic* [2] a pomocí tokenů. Autentizační metody mohou klientům předávat vlastní nastavení a mohou definovat zdroje v *API*. Pomocí těchto prostředků je možné definovat vlastní autentizační metody, je však nutno zajistit podporu jak na serveru, tak u klientů.

Jedna verze *API* může používat jednu, více, nebo žádnou autentizační metodu. Klient si může zvolit autentizační metodu, která mu vyhovuje.



Obrázek 3: Dokumentace konkrétní verze *API*



Obrázek 4: Autentizační metody

4.4.1 HTTP basic

Při použití této metody se uživatelské jméno a heslo musí zaslat s každým požadavkem na *API*. Tato metoda je vhodná na jednorázové akce, nehodí se na dlouhodobé použití, neboť je potřeba si pamatovat heslo.

4.4.2 Tokeny

Autentizace pomocí tokenů (viz diagram 5) funguje tak, že si nejprve klient pomocí jména a hesla vyžádá token. Po přidělení tokenu může klient zapomenout jméno i heslo a dále se autentizuje jen pomocí tokenu.

Tato autentizační metoda definuje zdroj `Token` (diagram 5), který má tři akce:

`Request` vyžádání tokenu (diagram 6)

`Renew` prodloužení platnosti tokenu

`Revoke` zrušení platnosti tokenu

Tokeny mohou mít různou dobu platnosti:

`fixed` pevně daná doba platnosti

`renewable_manual` platnost tokenu lze manuálně prodloužit

`renewable_auto` platnost tokenu se prodlužuje automaticky s každým požadavkem

`permanent` token je platný navždy, resp. dokud není zrušen

Typ tokenu a dobu platnosti si volí klient při žádosti o vytvoření tokenu.

Vstupní parametr `interval` z akce *Request* je počet sekund, po který je token validní od jeho vytvoření. V případě prodloužení tokenu se opět k aktuálnímu času přičte `interval` sekund.

4.5 Zdroje

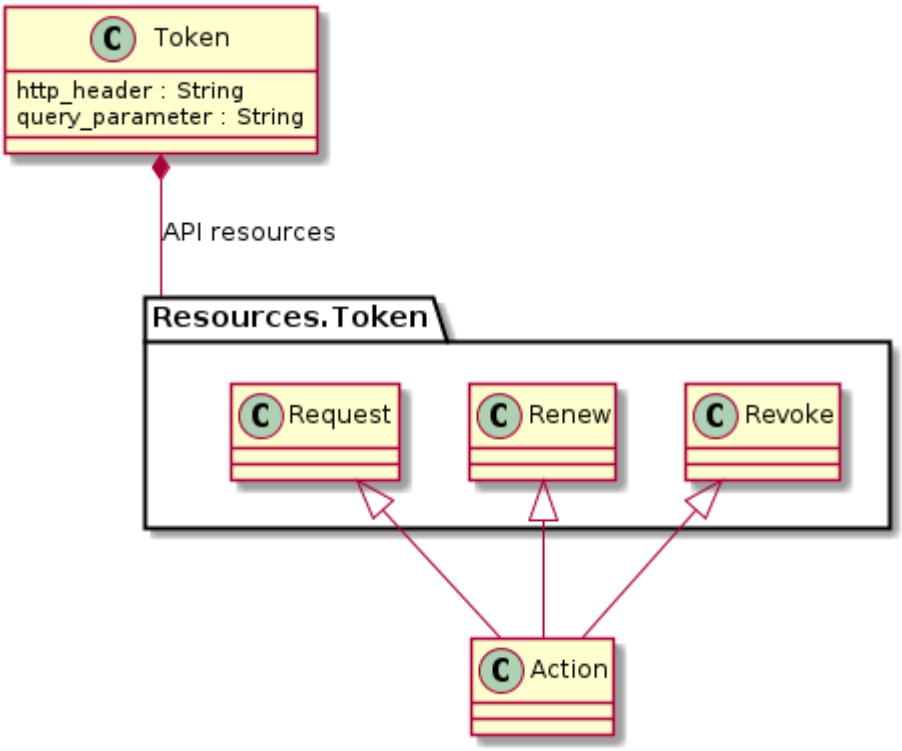
Zdroj odpovída pojmu *resource* definovaném v architektuře *REST*. Každý zdroj je identifikován svým jménem, které je poté součástí *URL*. Zdroje mohou být zanořené.

4.6 Akce

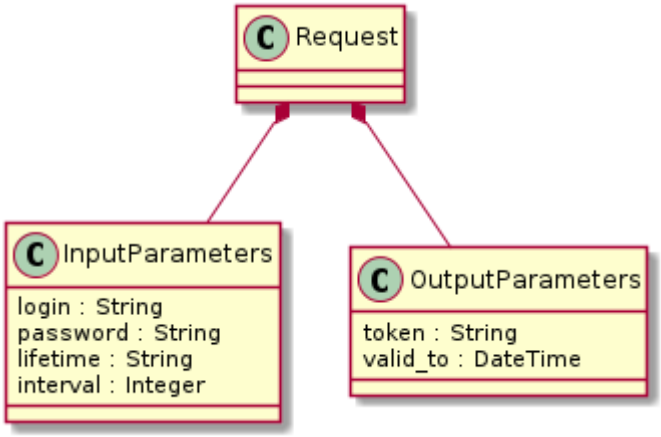
Každý zdroj může obsahovat různé akce, pomocí kterých s ním manipulujeme. Akce jsou identifikovány jménem, které je součástí *URL*. Standardní akce jsou popsány v tabulce 1.

Každý zdroj by měl mít definovány minimálně akce *Index* a *Show*. Mimo tyto akce je možné definovat další akce s libovolným názvem. Tvůrce *API* si může zvolit na jakou *HTTP* metodu má akce reagovat.

Parametry akce:



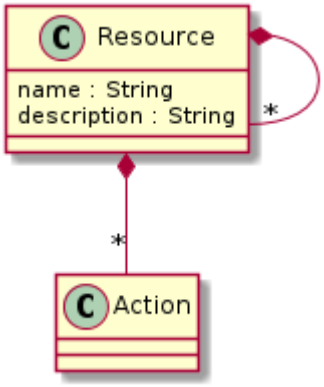
Obrázek 5: Autentizace pomocí tokenů



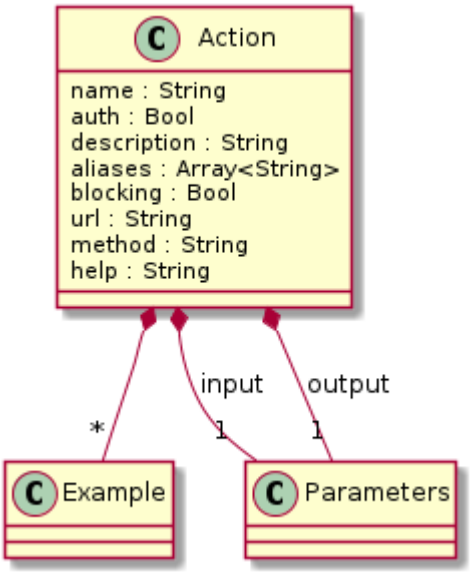
Obrázek 6: Vyžádání tokenu

Akce	*HTTP* požadavek	Popis
Index	`GET /resources`	vrací seznam objektů, které zdroj obsahuje
Show	`GET /resources/123`	vrací jeden konkrétní objekt
Create	`POST /resources`	vytvoření nového objektu
Update	`PUT /resources/123`	změna existujícího objektu
Delete	`DELETE /resources/123`	vymazání existujícího objektu

Tabulka 1: Standardní akce



Obrázek 7: Zdroj



Obrázek 8: Akce

`auth` udává, zda musí být uživatel k vyvolání akce autentizován

`description` popisek, který je součásti dokumentace

`aliases` seznam alternativních názvů akce

`blocking` udává, zda je akce blokující, viz sekce 4.13

`url` cesta pro vyvolání akce

`method` *HTTP* metoda

`help` cesta, na které lze získat dokumentaci konkrétní akce

Parametry *url* a *help* neobsahují plnohodnotnou *URL*, ale pouze lokální cestu ke zdroji a akci v API serveru.

4.6.1 URL parametry

URL akce se skládá z verze API, názvu všech nadřazených zdrojů a názvu akce. Dále mohou být součástí *URL* parametry, které slouží k identifikaci konkrétních objektů. Například, seznam uživatelů bychom získali pomocí `GET /users` (akce *Index*), pokud ale chceme získat jednoho konkrétního uživatele, použijeme akci *Show*, ale musíme přidat uživatelovo ID: `GET /users/1`.

URL takové akce je pak v dokumentaci jako `GET /users/:user_id`. Parametry v *URL* tedy začínají dvojtečkou a končí buď lomítkem, nebo koncem *URL*. Názvy *URL* parametrů si volí tvůrce API, pro klienty nejsou podstatné. Klienty zajímá jen kolik *URL* parametrů akce potřebuje, aby byli schopni je vyplnit a zahlásit chybu, když nějaký chybí.

Tvůrce *API* si tedy u každé akce může zvolit libovolnou *HTTP* metodu, *URL* a parametry v ní. Zvolené parametry jsou pak součástí dokumentace pro klienty.

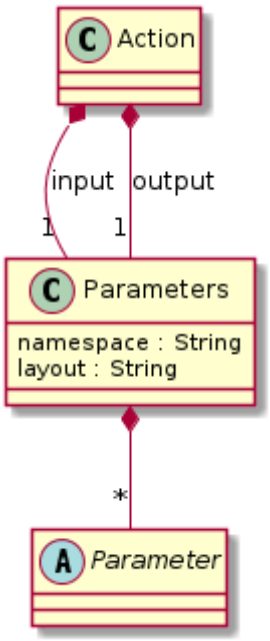
4.7 Formát vstupu/výstupu

HaveAPI definuje čtyři typy forem vstupu a výstupu:

- `object`
- `object_list`
- `hash`
- `hash_list`

`object` udává, že vstupem/výstupem je jeden objekt, který je popsán nějakými parametry. `object_list` je pak seznam takových objektů.

`hash` a `hash_list` fungují podobně. Rozdíl mezi `object` a `hash` je, že mezi parametry v `object` se mohou objevit asociace – odkaz na jiný zdroj v *API*. Klienti pak z parametrů



Obrázek 9: Vstupní/výstupní parametry

typu `object` vytvářejí instance objektů s pokročilou funcionalitou, jako např. předčasné načítání asociací. `hash` se prezentuje jen jako hash mapa – asociativní pole.

Formát vstupu/výstupu může být různý pro vstupní a výstupní parametry, viz digram 9.

4.8 Jmenný prostor

Výstupní parametry v daném formátu (`object`, `object_list`, `hash`, `hash_list`) jsou umístěny v tzv. jmenném prostoru. Jedná se jen o zanoření do dalšího objektu, kde výstupní parametry jsou přístupné pod klíčem s hodnotou `namespace`, viz diagram 9. Toto oddělení je nutné, neboť součástí odpovědi můžou být kromě výstupních parametrů např. metadata, viz sekce 4.12.

Hodnotou `namespace` je typicky název zdroje.

4.9 Vstupní/výstupní parametry

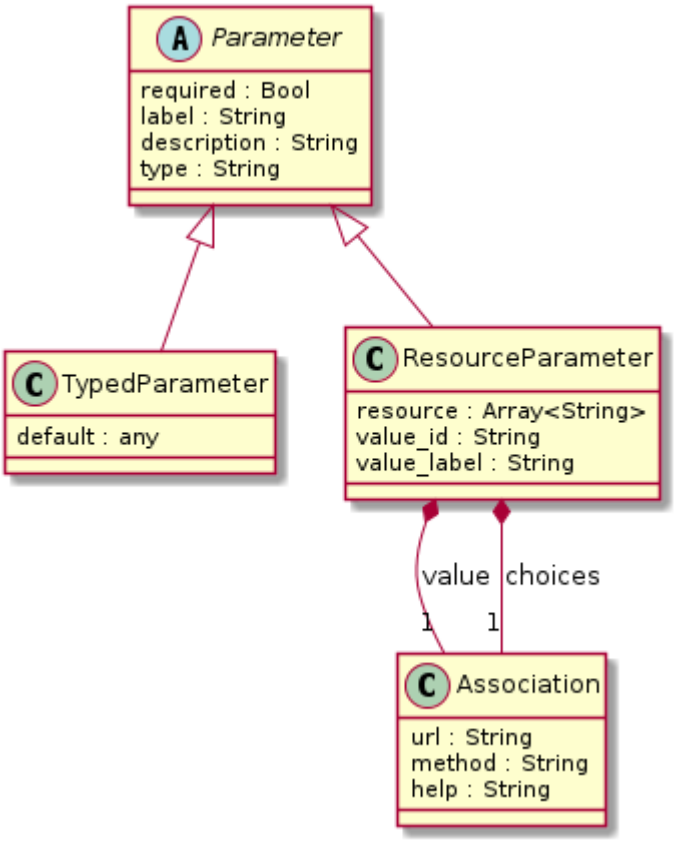
Parametry mohou být dvojího typu: hodnotové a asociace (diagram 10). Buď je tedy obsahem nějaká hodnota, jako řetězec, číslo, datum, apod., nebo odkaz na jiný zdroj v *API*.

4.9.1 Hodnotové

Parametry mají jasně specifikovaný datový typ:

`String` jednořádkový řetězec

`Text` víceřádkový řetězec



Obrázek 10: Parametry

`Boolean` pravda/nepravda

`Integer` celé číslo

`Float` desetinné číslo

`Datetime` datum a čas ve formátu *ISO 8601* [7]

`Custom` vlastní, blíže nespecifikovaný datový typ

Rozdíl mezi *String* a *Text* je především sémantický. V grafických aplikacích se pro zobrazení/editaci typu *String* se používá jednořádkové textové pole, kdežto pro *Text* víceřádkové textové pole.

Typ *Custom* může obsahovat cokoliv: číslo, řetězec, seznam, mapu, apod. Tvůrce *API* musí zajistit, aby byla zvolená struktura formátovátelná v *JSON*.

Vstupní hodnotové parametry mohou mít nastavenou výchozí hodnotu, která se použije, pokud jej klient nezašle. Tato hodnota je uložena pod klíčem `default`.

4.9.2 Asociace

HaveAPI podporuje jen asociace typu *n:1*. Asociace může vést na jakýkoli zdroj ve stejné verzi *API*, který má definovánu akci *Show*.

Vlastnosti asociací (diagram 10):

`resource` cesta k asociovanému zdroji v *API*

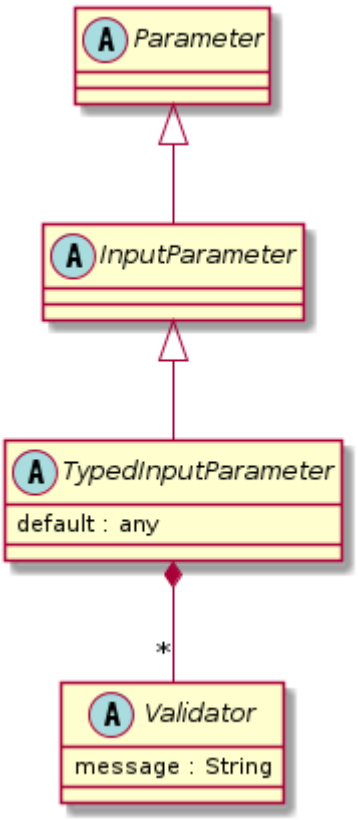
`value_id` název výstupního parametru z akce *Show* asociovaného zdroje, který obsahuje ID objektu ze zdroje; výchozí hodnota je `id`

`value_label` název výstupního parametru z akce *Show* asociovaného zdroje, který obsahuje popisek objektu ze zdroje; výchozí hodnota je `label`

`value` odkaz na dokumentaci akce *Show* asociovaného zdroje

`choices` odkaz na dokumentaci akce *Index* asociovaného zdroje

Při volání akce je hodnotou vstupního asociačního parametru ID asociovaného objektu ze zdroje. V odpovědi ze serveru je výstupní asociační parametr reprezentován objektem, který obsahuje výstupní parametry akce *Show* asociovaného zdroje. Objekt může být zaslán jen neúplně, tzn. obsahuje jen atributy vyplývající z vlastností `value_id` a `value_label` a lokálních metadata parametry (viz sekce 4.12), které obsahují informace potřebné ke stažení ostatních parametrů asociovaného objektu v případě potřeby.



Obrázek 11: Validátory hodnotových vstupních parametrů

4.10 Validátory vstupních parametrů

Hodnotové vstupní parametry mohou mít nastaven jeden nebo více validátorů (diagram 11). Validátory kromě validátoru přítomnosti jsou vyhodnoceny jen pokud byl parametr zadán.

Všechny validátory mají jednu společnou vlastnost `message` – zpráva, která se uživateli zobrazí, pokud zadaná hodnota neprojde validátorem. Výskyty `%{value}` v `message` jsou nahrazeny za aktuální hodnotu parametru.

4.10.1 Validátor přítomnosti

Validátor přítomnosti kontroluje, že je vstupní parametr zadaný a volitelně neprázdný.

Parametry:

`empty` boolean; pokud je `true` a hodnotou je řetězec, musí obsahovat alespoň jeden viditelný znak

4.10.2 Validátor konkrétní hodnoty

Tento validátor přijímá jen jednu konkrétní zadanou hodnotu.

Parametry:

`value` příjímaná hodnota

4.10.3 Validátor potvrzení

Validátor potvrzení kontroluje, zda dva parametry (ne)mají stejnou hodnotu. Používá se např. pro zadání hesla nebo e-mailové adresy, viz digram 12.

Parametry:

`equal` boolean; rozhoduje o tom, zda se mají dané parametry shodovat, nebo ne

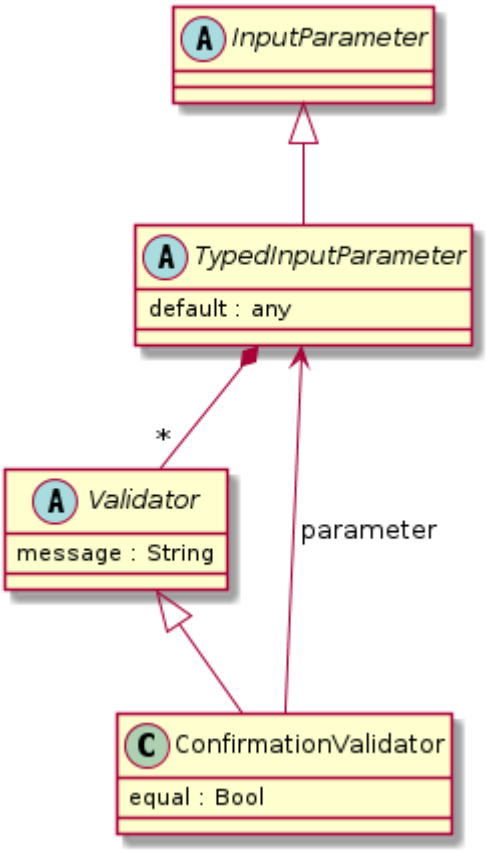
`parameter` název druhého vstupního parametru k ověření

4.10.4 Validátor povolených hodnot

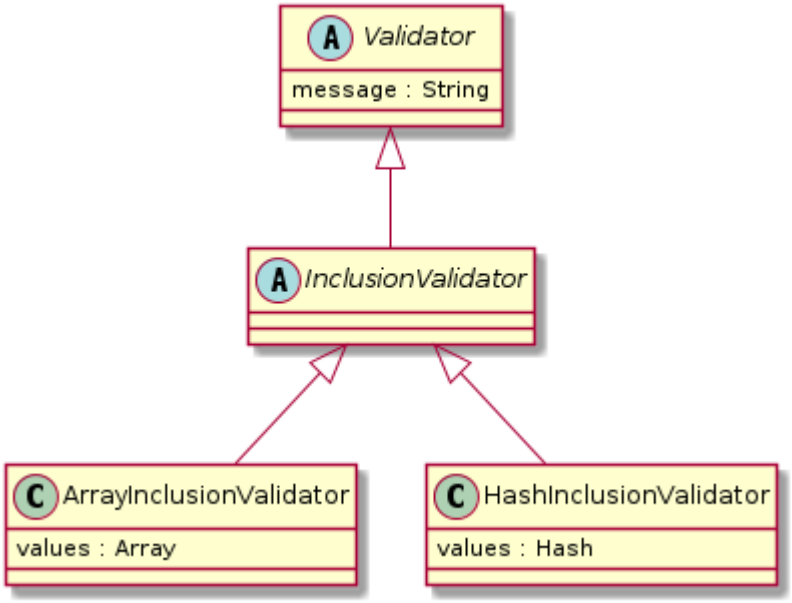
Validátor povolených hodnot přijímá jen hodnoty z definovaného seznamu. Může se vyskytovat ve dvou podobách (diagram 13): seznam hodnot je předán buď jako seznam nebo mapa (asociativní pole). Pokud jsou hodnoty v podobě mapy, klíče z mapy jsou přijímané hodnoty a hodnoty z mapy popisky zobrazené uživatelům.

Parametry:

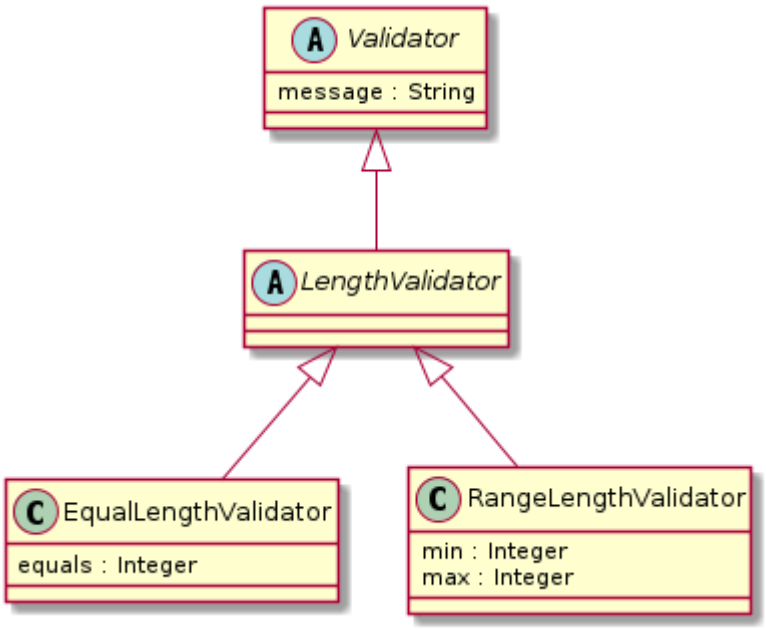
`values` seznam nebo mapa přijatelných hodnot



Obrázek 12: Validátor potvrzení



Obrázek 13: Validátor povolených hodnot



Obrázek 14: Validátor délky řetězce

4.10.5 Validátor zakázaných hodnot

Tento validátor příjme všechny hodnoty, které nejsou v seznamu zakázaných hodnot.

Parametry:

`values` seznam zakázaných hodnot

4.10.6 Validátor vlastního formátu

Pomocí tohoto validátoru můžeme formát přijatelných hodnot specifikovat pomocí regulárního výrazu.

Parametry:

`rx` regulární výraz

`match` boolean; rozhoduje o tom, zda má hodnota regulárnímu výrazu vyhovovat, nebo ne

`description` popis požadovaného formátu zobrazený uživatelům

4.10.7 Validátor délky řetězce

Validátor kontrolující délku řetězce. Může vyžadovat přesnou délku, nebo kontrolovat jestli je délka v určitém intervalu, viz diagram 14.

Parametry:

`equals` konkrétní délka řetězce

`min` minimální délka řetězce

`max` maximální délka řetězce

Pokud je nastaven parametr `equals`, je vyžadována přesná délka řetězce. Není-li `equals` nastaven, kontroluje se, zda délka spadá do intervalu zadaným `min` a `max`, včetně. Pokud jedno z `min` nebo `max` není specifikováno, kontroluje se jen polootevřený interval.

4.10.8 Validátor čísel

Validátorem čísel lze kontrolovat celá čísla, desetinná čísla i formát řetězce.

Parametry:

`min` minimální hodnota, včetně

`max` maximální hodnota, včetně

`step` číslo musí být v definovaném kroku, ověřuje se jako $(n - min) \bmod step \neq 0$, kde n je kontrolovaná hodnota a min je minimální hodnota, nebo 0

`mod` číslo musí být dělitelné n, ověřuje se jako $n \bmod mod = 0$, kde n je kontrolovaná hodnota

`odd` boolean; číslo musí být liché

`even` boolean; číslo musí být sudé

4.10.9 Vlastní validátor

Pokud nějaký validační mechanismus nelze vyjádřit pomocí dokumentovaných validátorů, ale chceme o něm uživatele *API* informovat, použije se tento validátor. Obsahuje jen textový popisek toho, co se interně v *API* validuje, jaké jsou požadavky na hodnotu parametru, apod.

4.11 Ukázky použití

Každá akce může mít ukázky použití (diagram 15). Ukázky demonstrují konkrétní použití dané akce, tj. příklady vstupních parametrů a výstupních dat, chybové stavy, apod.

Parametry:

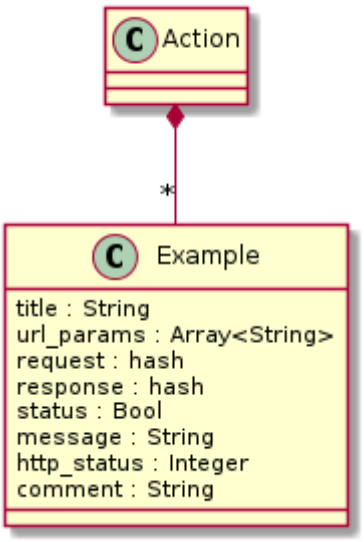
`title` titulek

`url_params` parametry v *URL*

`request` ukázka vstupních parametrů

`response` ukázka výstupních parametrů

`status` status z obálky (viz sekce 4.2)



Obrázek 15: Ukázky použití akcí

`message` vysvětlující zpráva z obálky

`errors` ukázkové chyby vstupních parametrů

`http_status` stavový kód *HTTP* odpovědi serveru

`comment` textový popisek ukázky

4.12 Metadata

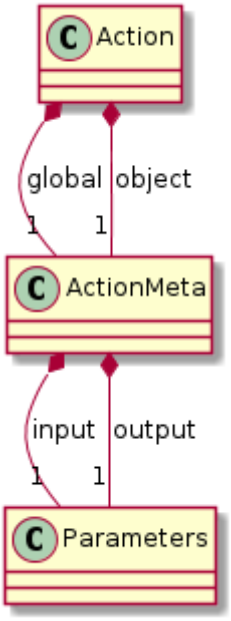
Metadata tvoří postranní kanál, přes který lze přenášet pomocné informace. *API* musí být plně funkční, aniž by klient musel umět pracovat s metadaty. Každá akce může mít vstupní a výstupní metadata. Metadata se předávají pomocí parametrů, které jsou popsány stejně jako standardní vstupní a výstupní parametry akce, viz diagram 16. Tvůrce *API* má možnost vytvořit si své vlastní metadata parametry.

Metadata jsou dvojího druhu: globální a lokální. Globální metadata obsahují informace týkající se celého požadavku/odpovědi, resp. všech zasílaných dat. Lokální metadata se týkají jen jednoho konkrétního objektu. Pokud tedy akce vrací parametry ve formátu `object_list` nebo `hash_list`, má jedny globální metadata a lokální metadata pro každý přenášený objekt.

Metadata jsou uloženy pod klíčem, který je k dispozici v popisu verze *API* z diagramu 3. Výchozí hodnota je `_meta`.

HaveAPI definuje význam metadata parametrů popsaných v tabulce 2.

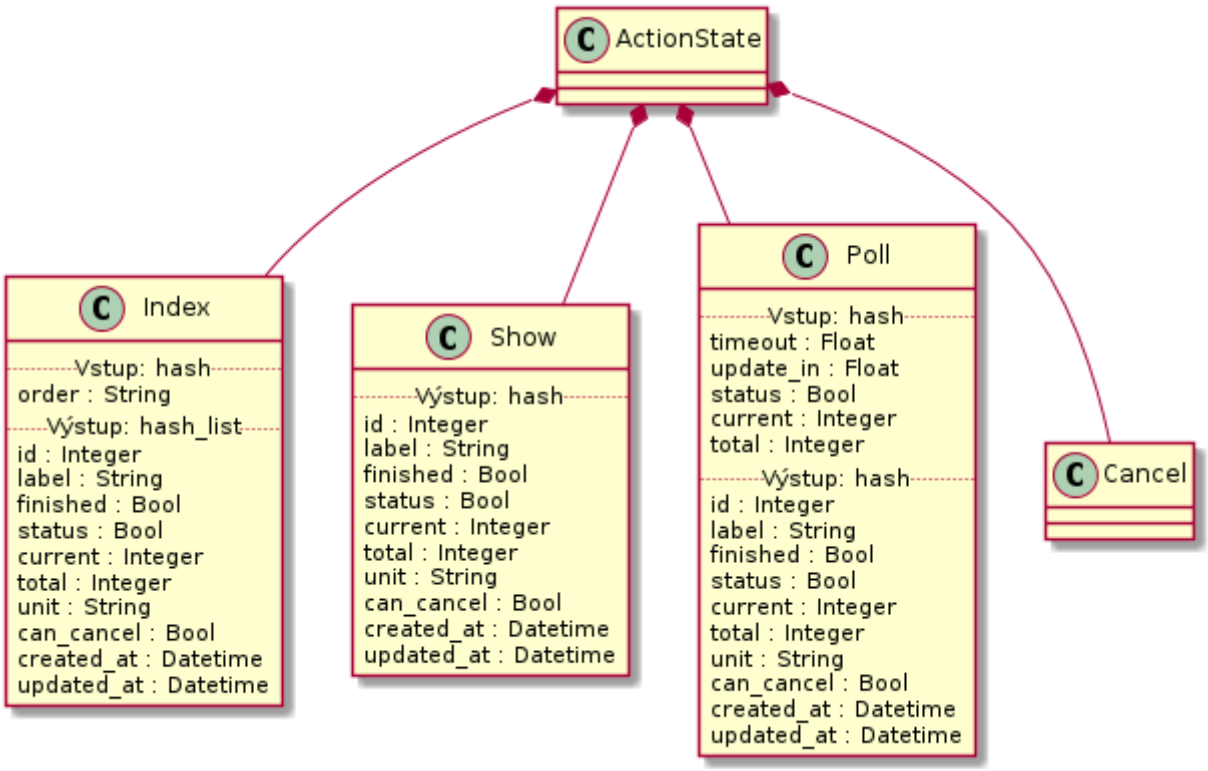
Metadata jsou nezbytné, pokud pracujeme s asociacemi mezi zdroji. Pokud není řečeno parametrem `includes` jinak, asociovaný zdroj se nezasílá celý, ale jen jeho ID a popisek, viz sekce 4.9.2. Aby byl klient schopen objekt načíst a nemusel kvůli tomu stahovat dokumentaci akce



Obrázek 16: Metadata

Akce	Směr	Umístění	Název	Datový typ	Popis
Index	Vstup	Globální	`count`	`Bool`	Žádost o zjištění celkového počtu položek
Index	Výstup	Globální	`total_count`	`Integer`	Celkový počet položek
-	Vstup	Globální	`includes`	`Custom`	Seznam asociovaných zdrojů, které mají být staženy
-	Výstup	Globální	`action_state_id`	`Integer`	Identifikátor blokující operace (viz sekce 4.13)
-	Výstup	Lokální	`url_params`	`Custom`	Seznam *URL* parametrů potřebných k nalezení zdroje
-	Výstup	Lokální	`resolved`	`Bool`	Určuje, zda je asociovaný zdroj plně stažen, nebo ne

Tabulka 2: Definované metadata parametry



Obrázek 17: Zdroj *action_state*

Show každého asociovaného zdroje, potřebuje znát *URL* parametry. Ty jsou předány v lokálním metadata parametru `url_params`.

4.13 Blokující mód

Akce mohou v *API* vyvolat dlouhotrvající operace libovolné délky. *HaveAPI* definuje rozhraní pro takové akce. Klient má k dispozici informaci o tom, že akce může trvat delší dobu a může se rozhodnout, jestli bude sledovat její průběh, čekat na dokončení, nebo pokračovat ve své činnosti. Tyto akce budeme nadále označovat jako akce blokující.

Blokující akce se poznají tak, že mají parametr `blocking` nastaven na `true` (viz digram 8). Chovají se stejně jako akce neblokující – mají vstupní a výstupní parametry, server na požadavek odpoví ihned a vrátí výstupní parametry. Pokud je součástí globálních výstupních metadat parametr `action_state_id`, operace je dlouhotrvající. Parametr `action_state_id` obsahuje ID operace, pomocí kterého může klient kontrolovat stav operace, případně ji přerušit.

API které chtějí blokující akce využívat, musí definovat zdroj `action_state`. Akce a jejich parametry jsou znázorněny na diagramu 17.

Popis parametrů:

`id` ID operace

`label` popisek operace pro zobrazení uživateli

`finished` indikuje dokončení operace

`status` indikuje status akce – dosavadní úspěch, nebo selhání

`current` číslo vyjadřující průběh akce

`total` pokud je `current` stejné jako `total`, operace by měla být dokončena

`unit` jednotka čísel `current` a `total`

`can_cancel` indikuje přerušitelnost operace akcí *Cancel*

`created_at` datum spuštění operace

`updated_at` datum poslední změny stavu/průběhu operace

Akce *Index* má vstupní parametr `order`, pomocí kterého si klient může vybrat směr řazení operací – `newest` řadí akce sestupně podle data spuštění, `oldest` vzestupně. *Index* vrací jen akce, které ještě nejsou dokončené, tedy `finished` je `false`.

Akce *Poll* vrací stejné informace jako *Show*, avšak umožňuje tzv. *long-polling*, tzn. *API* server neodpoví hned, ale čeká, až nastane změna ve stavu akce, nebo vyprší čas. Vstupními parametry jsou:

`timeout` maximální počet sekund, po který bude *API* server čekat na změnu

`update_in` počet sekund, po kterém server odešle odpověď, došlo-li ke změně průběhu operace (změna parametrů `current` nebo `total`)

`status` hodnota, vůči které se testuje změna stavu operace (poslední pro klienta známý stav)

`current` hodnota, vůči které se testuje změna průběhu operace

`total` hodnota, vůči které se testuje změna průběhu operace

Pomocí akce *Cancel* lze přerušit operace, které oznámí `can_cancel` jako `true`. Akce *Cancel* je sama o sobě blokující, takže klient může sledovat průběh a stav přerušení operace.

4.14 Přenos dokumentace

API server zasílá klientům dokumentaci pomocí *HTTP* metody *OPTIONS*, viz tabulka 3. Dokumentace je přenášena v obálce, viz sekce 4.2.

Dokumentaci si je možné vyžádat také pro každou akci, stačí jako *HTTP* metodu použít *OPTIONS* a původní *HTTP* metodu akce předat jako *URL* parametr, tedy např. `OPTIONS /resources?method=GET`. Pokud se ptáme na dokumentaci konkrétního zdroje, jako např. `OPTIONS /resources/123?method=GET`, dokumentace může obsahovat informace specické pro daný zdroj, jako např. odkazy na dokumentaci asociovaných objektů (více v sekci 4.9.2).

Požadavek	Odpověd
`OPTIONS /`	Dokumentace celého *API*, všech jeho verzí
`OPTIONS /v1`	Dokumentace konkrétní verze *API*
`OPTIONS /?describe=versions`	Seznam verzí *API*
`OPTIONS /?describe=default`	Dokumentace výchozí verze *API*

Tabulka 3: Přístupové metody k strojově čitelné dokumentaci

4.15 Volání akcí

Po stažení dokumentace *API* má klient všechny informace k tomu, aby s daným *API* dokázal pracovat. Vstupní/výstupní data jsou formátovány v *JSON*. Pro znázornění je v ukázkách použit protokol *HTTP/1.1*, avšak na stejném principu lze využívat i *HTTP/2* [3].

Ukázka 1 obsahuje předpis pro *HTTP* požadavek pro vyvolání akce a ukázka 2 je pak odpověd *API* serveru, která je zaslána v obálce (viz sekce 4.2). Hodnoty v ostrých závorkách jsou dosazeny z dokumentace *API*. U vstupních a výstupních parametrů vždy záleží na příslušném formátu parametrů akce. V případě `object_list` a `hash_list` se jedná o pole objektů, pro `object` a `hash` je to objekt. Případná lokální metadata jsou součástí každého objektu, pod klíčem, jehož název je v dokumentaci.

```
<METODA> <CESTA> HTTP/1.1
Host: <HOST>
Content-Type: application/json
Accept: application/json

{
    "<JMENNÝ PROSTOR>": <VSTUPNÍ PARAMETRY>
}
```

Ukázka kódu 1: *HTTP* požadavek pro volání akce

```
Content-Type: application/json

{
    "status": true,
    "response": {
        "<JMENNÝ PROSTOR>": <VÝSTUPNÍ PARAMETRY>,
        "<METADATA>": <GLOBÁLNÍ VÝSTUPNÍ METADATA PARAMETRY>
    },
    "message": null,
    "errors: null
}
```

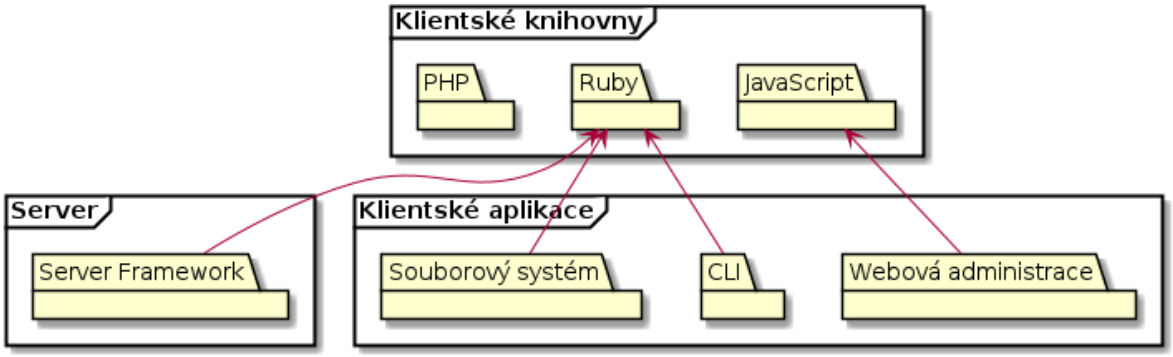
Ukázka kódu 2: *HTTP* odpověd na volání akce

5 Referenční implementace

Referenční implementace se skládá z následujících částí:

- Framework pro tvorbu *API* serverů
- Klientské knihovny
  - Knihovna v *Ruby* [8, 9]
  - Knihovna v *JavaScriptu* [10]
  - Knihovna v *PHP* [11]
- Uživatelské rozhraní
  - Rozhraní v příkazové řádce (*CLI*)
  - Virtuální souborový systém
  - Webová administrace

Implementace serverové části, jednotlivých klientů a aplikací jsou odděleny do samostatných projektů (diagram 18). Všechny části však používají stejné verze, které určují vzájemnou kompatibilitu. Referenční implementace je k dispozici pod svobodnou licencí *MIT*.



Obrázek 18: Moduly *HaveAPI*

6 Implementace serverové části

Aby bylo použití protokolu *HaveAPI* co nejjednodušší, je serverová část implementována ve formě frameworku v *Ruby*.

Zdroje v *API* jsou definovány jako třídy. Akce jednotlivých zdrojů jsou pak podtřídy daného zdroje. K popisu zdrojů a akcí se používá doménový jazyk implementovaný v rámci *Ruby*, dále jako *DSL*. Všechny zdroje jsou umístěny v modulu, který je následně předán frameworku. Při startu aplikace pak *HaveAPI* projde všechny třídy v modulu, vybere z nich ty, co se identifikují jako zdroje, a zaregistruje jejich akce v interním routeru založeném na knihovně *Sinatra* [12].

Při příchozím požadavku pak router podle *HTTP* metody a cesty vyhledá odpovídající akci. Pokud se klient táže na dokumentaci, *HaveAPI* mu ji samo předá. Pokud se jedná o volání akce, *HaveAPI* zkontroluje a ošetří vstupní parametry, předá kontrolu akci, zpracuje návratovou hodnotu, naformátuje a zašle odpověď klientovi (diagram 19).

6.1 Doménový jazyk

V rámci *DSL* je možné zdrojům a akcím nastavovat atributy, které se používají k tvorbě dokumentace a běhu *API* serveru. Každý atribut má přístupové metody, tzv. *getter* a *setter*. *getter* je metoda s názvem atributu, bez parametrů. *setter* je metoda se stejným názvem a jedním argumentem – novou hodnotou atributu.

DSL zdrojů a akcí dále definuje metody, kterým se předává spustitelný blok kódu (podobné anonymním funkcím z jiných jazyků). Tyto bloky kódu jsou spouštěny ve zvláštním kontextu, kde jsou pak k dispozici specifické metody, např. na registraci parametrů, autorizace, ukázek použití, apod.

Dostupné *DSL* atributy a metody jsou popsány níže.

6.2 Definice zdroje

Zdroje jsou potomkem třídy `HaveAPI::Resource` a obsahují atributy z tabulky 4.

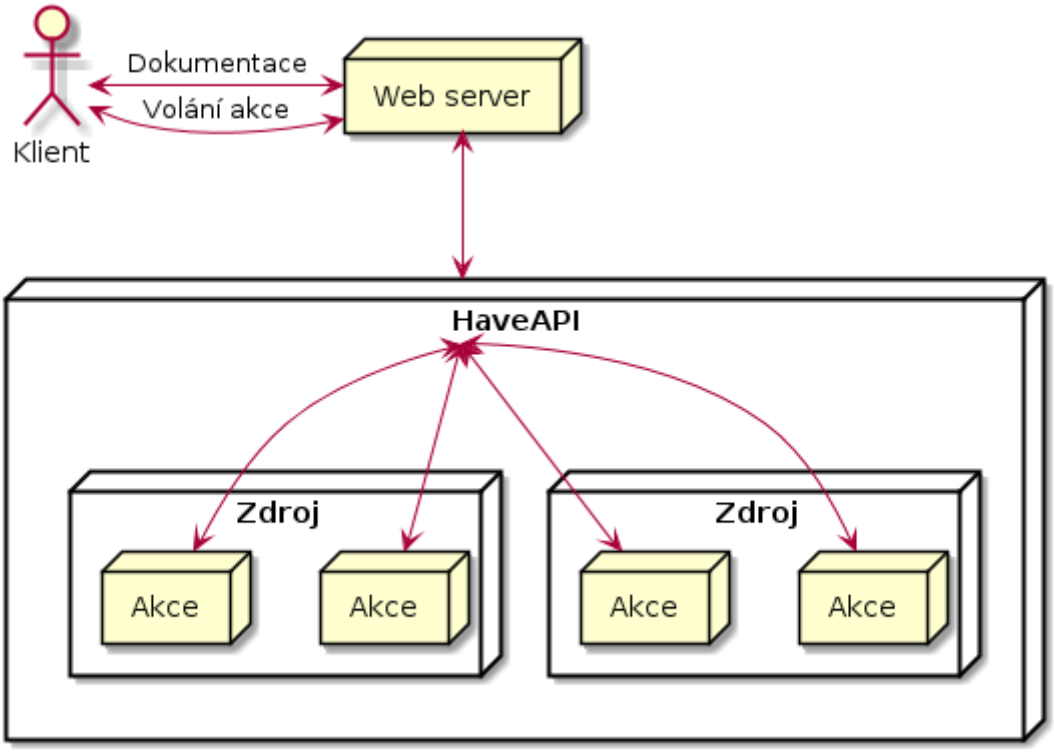
`version` udává, v kterých *API* verzích je tento zdroj dostupný. Pokud není nastaveno, použije se výchozí číslo verze (viz sekce 6.15).

Pomocí atributu `singular` lze vytvořit tzv. singleton zdroje. To jsou zdroje, které obsahují jen jeden objekt, tedy akce *Index* není potřeba, stačí akce *Show*.

Příklad viz ukázka 3.

```
class MyResource < HaveAPI::Resource
  version '1.0'
  desc 'Toto je můj první zdroj'
end
```

Ukázka kódu 3: Definicice zdroje



Obrázek 19: *HaveAPI* framework

Atribut	Typ	Výchozí hodnota	Popis
`version`	`String`	`nil`	Verze zdroje
`desc`	`String`	`nil`	Popis zdroje, zobrazuje se v dokumentaci
`model`	`Object`	`nil`	Volitelné propojení s modelem
`route`	`String`	Název zdroje v množném čísle	Cesta ke zdroji
`auth`	`Boolean`	`true`	Značí, zda zdroj vyžaduje autentizaci
`singular`	`Boolean`	`false`	Označuje tzv. singleton zdroj

Tabulka 4: Atributy zdroje

Atribut	Typ	Výchozí hodnota	Popis
`version`	`String`	`nil`	Verze akce
`desc`	`String`	`nil`	Popis akce, zobrazuje se v dokumentaci
`http_method`	`Symbol`	`:get`	*HTTP* metoda
`route`	`String`	Název akce	Cesta k akci
`resolve`	`Proc`	`nil`	Anonymní funkce, která k objektu přiřadí *URL* parametry
`auth`	`Boolean`	`true`	Značí zda akce vyžaduje autentizaci
`aliases`	`Array<Symbol>`	`[]`	Alternativní názvy akce
`blocking`	`Boolean`	`false`	Označuje blokující akce

Tabulka 5: Atributy akce

6.3 Definice akce

Akce jsou potomkem třídy `HaveAPI::Action`, jejíž atributy jsou popsány v tabulce 5.

Definice akce viz ukázka 4.

```
class MyResource < HaveAPI::Resource
  version '1.0'
  desc 'Toto je můj první zdroj'

  class Index < HaveAPI::Action
    desc 'Vrací seznam objektů'

    # Výsledná cesta bude `GET /my_resources`
    http_method :get
    route ''
  end

  class Show < HaveAPI::Action
    desc 'Vrací jeden konkrétní objekt'

    # Výsledná cesta bude `GET /my_resources/:my_resource_id`
    http_method :get
    route ':my_resource_id'
  end
end
```

Ukázka kódu 4: Definice akce

Tyto akce však zatím nic nedělají, nemají žádnou implementaci. Pro zpracování požadavku můžeme definovat několik metod (dle pořadí volání):

- `prepare`
- `pre_exec`
- `exec`

Metoda `prepare` se volá i pokud se klient dotazuje pouze na dokumentaci. Pokud vrátí objekt, dokumentace obsahuje konkrétní hodnoty pro daný objekt (např. *URL* parametry, hodnoty asociací, apod.). Jestliže nevrátí nic, přístup k tomuto objektu je zamítnut.

Metoda `pre_exec` je volána jen pokud se jedná o skutečné vyvolání akce, návratová hodnota se ignoruje.

Metoda `exec` pak slouží k vlastnímu zpracování akce a vrácení výsledku klientovi. Návratová hodnota je dále zpracována frameworkem, převedena do *JSON* a odeslána klientovi. Typ návratové hodnoty závísí na výstupním formátu akce (viz sekce 4.7): pro `hash` a `hash_list` je to `Hash` a `Array<Hash>`, pro `object` a `object_list` pak záleží na použitém modelu, viz sekce 6.13.

Aby mělo smysl definovat metody, musíme nejdřív definovat vstupní a výstupní parametry, viz následující sekce 6.4.

6.4 Vstupní/výstupní parametry

Součástí *DSL* akce jsou metody `input` a `output`, pomocí nichž se specifikují vstupní a výstupní parametry. Jediným argumentem těchto funkcí je typ formátu parametrů jako symbol (viz sekce 4.7).

Parametry se definují pomocí metod, jejichž název odpovídá datovému typu parametru: `integer`, `float`, `string`, `text`, `datetime`, `bool` a `resource`.

Prvním argumentem hodnotových parametrů (všechny kromě `resource`) je vždy název parametru, další argumenty jsou volitelné.

Prvním argumentem asociačního parametru (`resource`) je asociovaný zdroj, resp. jeho třída, další argumenty jsou taktéž volitelné.

Druhým argumentem jsou volitelné nastavení. Společné volby pro parametry všech typů jsou v tabulce 6, volby specifické pro hodnotové parametry v tabulce 7 a volby pro asociace v tabulce 8.

Protože akce jednoho zdroje mohou vyžadovat stejné parametry, je zde možnost definovat skupiny parametrů u zdroje a u vybraných akcí je využít. Skupiny parametrů se definují ve třídě zdroje pomocí *DSL* metody `params`, jejíž jediný argument je název skupiny parametrů. Této metodě předáme blok kódu, který definuje parametry. Skupin parametrů lze vytvořit libovolné množství a navzájem je propojovat. Skupinu parametrů lze využít pomocí metody `use` v kontextu bloku kódu pro metody `input`, `output` a `params`.

Název	Typ	Popis
`label`	`String`	Titulek
`desc`	`String`	Popisek
`db_name`	`Symbol`	Název parametru v modelu (je-li jiný)

Tabulka 6: Sdílené volby parametrů

Název	Typ	Popis
`default`	-	Výchozí hodnota
`fill`	`Boolean`	Udává, zda se má výchozí hodnota vyplnit
`clean`	`Proc`	Anonymní metoda k ošetření hodnoty
`load_validators`	`Boolean`	Rozhoduje o načítání validátorů z modelu

Tabulka 7: Volby hodnotových parametrů

K ošetřeným vstupním parametrům se z objektu akce přistupuje metodou `input`.

Příklad zdroje s akcí s parametry definovanými pomocí skupin a implementací metody `exec` viz ukázka 5.

Název	Typ	Popis
`choices`	`HaveAPI::Action`	Akce vracející seznam možných hodnot
`value_id`	`Symbol`	Název parametru obsahující identifikátor asociovaného zdroje
`value_label`	`Symbol`	Název parametru obsahující titulek asociovaného zdroje

Tabulka 8: Volby asociačních parametrů

```ruby
class MyResource < HaveAPI::Resource
  # Vytvoření skupiny parametrů pod názvem ‘common‘
  params(:common) do
    integer :id, label: 'ID'
    string :item, label: 'Item'
  end

  class Index < HaveAPI::Action
    http_method :get
    route ''
    desc "Vrací seznam, jehož maximální velikost je definována vstupním "+
         "parametrem ‘limit‘."

    input(:hash) do
      integer :limit, label: 'Limit', desc: 'Maximální počet položek',
              default: 10, fill: true
    end

    output(:hash_list) do
      # Použití skupiny parametrů
      use :common
    end

    def exec
      [
          {id: 1, item: 'prvni'},
          {id: 2, item: 'druhy'},
          {id: 3, item: 'treti'},
          # ...
      ][ 0 .. input[:limit] ]
    end
  end
end
```

Ukázka kódu 5: Implementace chování akce

6.5 Autorizace

DSL akce obsahuje metodu `authorize`, která řídí přístup k akci, případně lze filtrovat vstupní/-výstupní parametry. Autorizaci provádí funkce, která má jako argument objekt reprezentující

aktuálního autentizovaného uživatele (viz sekce 6.12) nebo `nil`. Tuto funkci lze ukončit zavoláním metody `allow` nebo `deny`. Pokud funkce explicitně nezavolá `allow`, přístup k akci je odepřen.

Filtrace vstupních/výstupních parametrů se provede pomocí metod `input`/`output` s argumentem `whitelist` nebo `blacklist`, což je seznam parametrů, které se mají povolit/zakázat. Díky těmto filtrům se metoda `exec` vůbec nemusí zabývat autorizací přístupu k jednotlivým vstupním a výstupním parametrům. Přístupné jsou vždy jen povolené vstupní parametry a na výstup stačí vždy vrátit všechny parametry s tím, že *HaveAPI* poté neautorizované parametry odstraní.

V ukázce 6 je přístup odmítnut zablokovaným uživatelům, plně povolen administrátorům, jinak je přístup povolen, ale parametr `secret` je odstraněn. Parametr `secret` je tedy zaslán jen administrátorům.

```
class Index < HaveAPI::Action
  output(:hash_list) do
    integer :id, label: 'ID'
    string :item, label: 'Item'
    string :secret, label: 'Secret'
  end

  authorize do |user|
    deny if user.blocked?
    allow if user.admin?
    output blacklist: [:secret]
    allow
  end
end
```

Ukázka kódu 6: Autorizace akce

6.6 Validátory

Validátory vstupních parametrů se nastavují jako volby při vytváření parametru. Seznam všech validátorů a způsobů jejich použití viz referenční dokumentace (sekce 6.19).

Obecně existují dvě formy použití validátoru: zkrácená a úplná. U zkrácené formy definujeme validátor jedinou hodnotou, tj. vždy to, co je hlavní náplní validátoru. Pro validátor formátu je to regulární výraz, pro validátor potvrzení je to jméno druhého parametry, apod.

V úplné formě validátoru předáváme více voleb, pomocí kterých můžeme více ovlivnit jeho nastavení, jako třeba chybovou hlášku.

Na ukázce 7 je příklad použití validátoru, který přijme jen dovolené hodnoty v krátké i úplné formě.

```
class Index < HaveAPI::Action
  input(:hash) do
    # Krátká forma
    string :order_by, choices: ['ascending', 'descending']

    # Úplná forma
    string :order_by, choices: {
      values: ['ascending', 'descending'],
      message: "'%{value}' není dovolená hodnota"
    }
  end
end
```

Ukázka kódu 7: Validátor vstupního parametru

6.7 Metadata

Metadata parametry se definují stejně jako vstupní/výstupní parametry, jen jsou zanořeny v metadata bloku, viz ukázka 8.

Z instance akce se k vstupním metadata parametrům přistupuje pomocí metody `meta`. Globální výstupní metadata parametry lze nastavit pomocí metody `set_meta`, viz referenční dokumentace (sekce 6.19).

```ruby
class Index < HaveAPI::Action
  # Globální metadata
  meta(:global) do
    input(:hash) do
      # Vstupní Parametry
    end

    output(:hash) do
      # Výstupní Parametry
    end
  end

  # Lokální metadata
  meta(:object) do
    output(:hash) do
      # Výstupní parametry
    end
  end
end
```

Ukázka kódu 8: Definice metadata parametrů

6.8 Ukázky použití

Ukázky použití akcí se definují pomocí *DSL* metody `example`, viz ukázka 9. Kromě zmíněných voleb je možné vložit *URL* parametry, *HTTP* status serveru, chybovou zprávu či chyby týkající se parametrů, viz referenční dokumentace (sekce 6.19).

```ruby
class Index < HaveAPI::Action
  http_method :get
  route ''

  input(:hash) do
    integer :limit, label: 'Limit', desc: 'Maximální počet položek',
            default: 10, fill: true
  end

  output(:hash_list) do
    integer :id, label: 'ID'
    string :item, label: 'Item'
  end

  example do
    # Vstupní parametry, není potřeba zmiňovat jmenný prostor
    request({
      limit: 2,
    })

    # Výstupní parametry
    response([
      {id: 1, item: 'prvni'},
      {id: 2, item: 'druhy'},
    ])

    # Popis ukázky
    comment 'Výpis prvních dvou položek'
  end
end
```

Ukázka kódu 9: Ukázky použití akce

6.9 Šablony akcí

Aby se u často používaných akcí jako *Index*, *Show*, *Create*, *Update* a *Delete* nemusela pokaždé specifikovat *HTTP* metoda a cesta, existují pomocné třídy, které toto nastavení vyřeší za nás.

Tyto třídy se nachází v modulu `HaveAPI::Actions::Default`, viz ukázka 10 a referenční dokumentace (sekce 6.19).

```ruby
class Index < HaveAPI::Actions::Default::Index
  desc "Vrací seznam, jehož maximální velikost je definována vstupním "+
       "parametrem ‘limit‘."

  input(:hash) do
    integer :limit, label: 'Limit', desc: 'Maximální počet položek',
            default: 10, fill: true
  end

  output(:hash_list) do
    integer :id, label: 'ID'
    string :item, label: 'Item'
  end

  def exec
    [
      {id: 1, item: 'prvni'},
      {id: 2, item: 'druhy'},
      {id: 3, item: 'treti'},
      # ...
    ][ 0 .. input[:limit] ]
  end
end
```

Ukázka kódu 10: Přednastavený předek akce

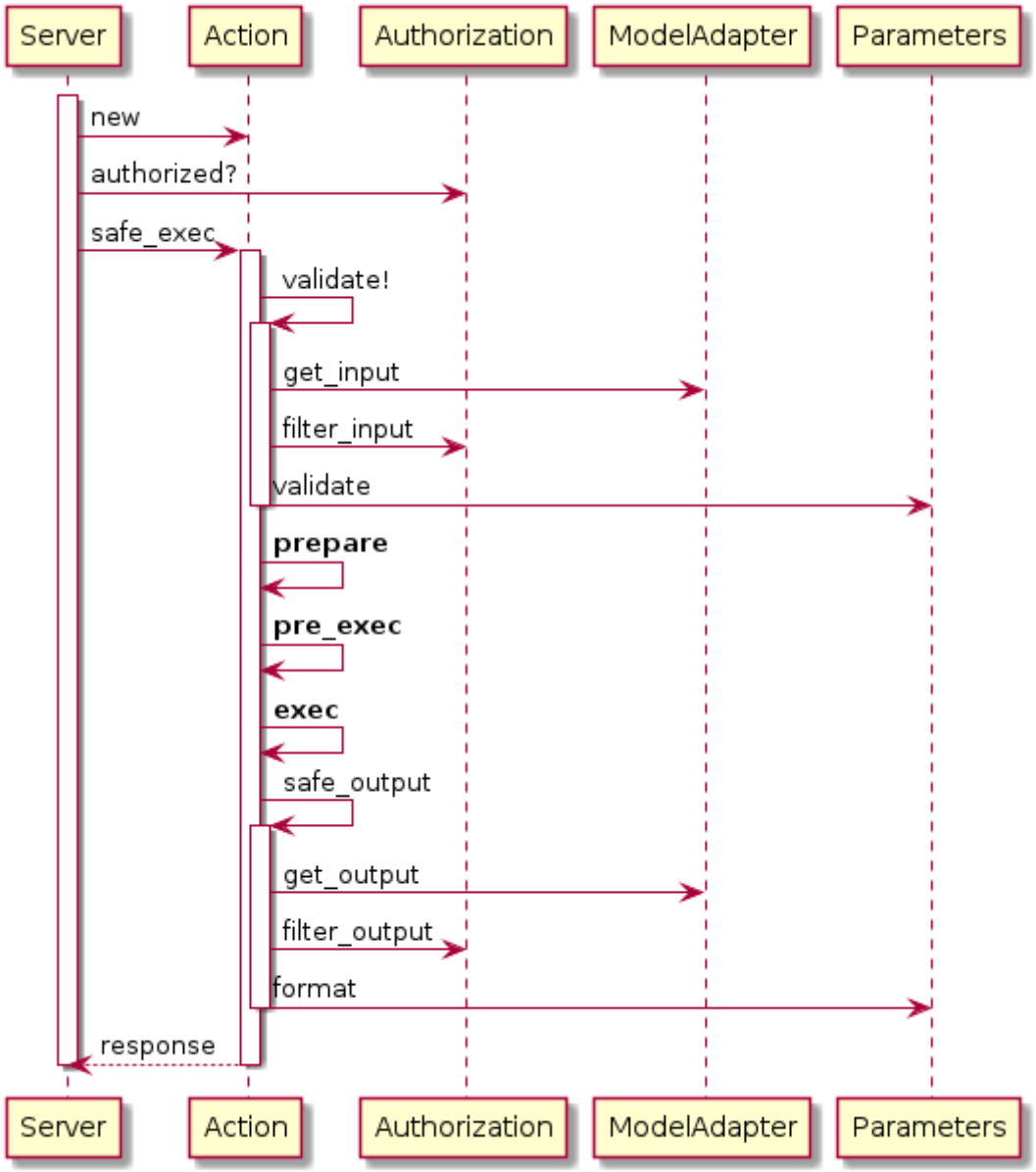
6.10 Blokující akce

Jestliže chce *API* využívat blokující akce, musí implementovat rozhraní `HaveAPI::ActionState` (dokumentace viz sekce 6.19). Pomocí tohoto rozhraní *HaveAPI* získává informace o běžících blokujících akcích a předává je klientům.

Blokující akce se potom označují atributem `blocking true` a musí deklarovat metodu `state_id`, která vrací ID operace, viz sekce 4.13.

6.11 Volání akce

Na diagramu 20 je znázorněn průběh volání akce. Tvůrce *API* má pod kontrolou jen objekt `Action`. Metody, které tvůrce *API* může implementovat, jsou zvýrazněny. Z toho jediná metoda, která musí být implementována, je `exec`.



Obrázek 20: Průběh volání akce

6.12 Autentizace

HaveAPI poskytuje rozhraní pro autentizační metody, které je potřeba implementovat, pokud chce výsledné *API* autentizaci využívat. Obecně je potřeba definovat metodu, která pro uživatelské jméno a heslo vrátí objekt reprezentující autentizovaného uživatele, nebo `nil`, pokud uživatel neexistuje, nebo je špatné heslo.

HaveAPI postupně zkouší všechny dostupné autentizační metody, dokud některá neuspěje. Autentizační proces je možné přerušit zavoláním metody `deny`.

V ukázce 11 je možné vidět jednoduchou autentizaci pomocí *HTTP basic*. Jako objekt identifikující uživatele je použito jednoduše `true`. V reálné aplikaci bychom vraceli nějaký užitečnější objekt identifikující uživatele.

Implementované autentizační metody je nutné v *HaveAPI* zaregistrovat, viz sekce 6.15. Více informací o autentizačních metodách viz referenční dokumentace (sekce 6.19).

```
class Basic < HaveAPI::Authentication::Basic::Provider
  protected
  def find_user(request, username, password)
    return true if username == 'admin' && password == '1234'
  end
end
```

Ukázka kódu 11: Autentizace pomocí *HTTP basic*

6.13 Propojení s ORM

HaveAPI obsahuje podporu pro propojení s libovolnými *ORM* systémy. To je užitečné zejména při zpracování návratových hodnot metody `Action#exec`, která díky integraci s *ORM* může vrátit jen objekt reprezentující dotaz do databáze, apod. *ORM* adaptér je pak schopen tento objekt zpracovat, vytáhnout z něj výstupní parametry a předat dále k naformátování do *JSON* a odeslání klientovi.

Dále je možné optimalizovat přístup k databázi, např. můžeme složeným dotazem načítat asociace objektu, aby se nemusel pro každou asociaci vykonávat samostatný dotaz do databáze.

ORM adaptéry se registrují pomocí metody `HaveAPI::ModelAdaper.register`. Každý zdroj v *API* může využívat jiný *ORM* adaptér. Ten se vybírá dynamicky podle atributu `model` (viz sekce 6.2).

6.13.1 ActiveRecord

Jediný integrovaný *ORM* adaptér je pro *ActiveRecord* [13], který je součástí populárního frameworku *Ruby on Rails* [13]. Tento adaptér umožňuje:

- asociace mezi zdroji v *API*,

- přednačítání asociací, automaticky a pomocí metadata parametrů,
- načítání validátorů z modelu do vstupních parametrů akcí.

Za instalaci a načtení *ActiveRecord* zodpovídá tvůrce *API*. *HaveAPI* pak automaticky použije *ActiveRecord* adaptér pro zdroje, které nastaví atribut `model` na model z *ActiveRecord*.

6.14 Instalace

Server framework *HaveAPI* je distribuován pomocí *RubyGems.org* [19] a vyžaduje *Ruby* minimálně ve verzi *2.0*. Instaluje se pomocí příkazu `gem install haveapi`.

6.15 Inicializace a spuštění API serveru

Inicializace *API* sestává z:

- Vytvoření instance *API* serveru,
- předání modulu obsahující zdroje,
- vyběru aktivních verzí *API*,
- registrace autentizačních metod,
- registrace rout akcí pod zvolený prefix,
- spuštění web serveru.

```ruby
require 'haveapi'

# Modul se zdroji
module Resources ; end

# Modul s implementacemi rozhraní autentizačních metod
module Authentication ; end

# Výchozí verze API, použije se, pokud není verze explicitně uvedena
# u zdroje/akce
HaveAPI.implicit_version = '1.0'

# Vytvoření instance a předání modulu se zdroji
api = HaveAPI::Server.new(Resources)

# API bude obsahovat všechny nalezené verze
api.use_version(:all)

# Registrace autentizačních metod
api.auth_chain << Authentication::Basic

# Registrace rout akcí
api.mount('/')

# Spuštění web serveru, na standardní výstup se vypíše, na jakém portu server
# poslouchá
api.start!
```

Ukázka kódu 12: Inicializace *API* serveru

6.16 Online dokumentace

Po spuštění *API* je automaticky k dispozici online dokumentace (obrázek 21), naformátovaná pomocí *HTML*, která obsahuje všechny vybrané verze *API*, jejich zdroje, akce, parametry, apod. Ukázky použití akcí jsou zobrazeny v syntaxi klientů v různých programovacích jazycích.

Pokud *API* implementuje autentizaci, do dokumentace je možné se přihlásit. Autentizovaným uživatelům se zobrazují jen zdroje, akce a parametry, ke kterým mají autorizovaný přístup.

Jestliže *API* běží na veřejné adrese, je možné se k němu kliknutím tlačítka připojit pomocí *haveapi-webui* (sekce 7.7), které je hostované na serverech projektu *HaveAPI*.

Obrázek 21: Online dokumentace

Součástí online dokumentace je i popis protokolu *HaveAPI* a *JSON* schémata formátu dokumentace *API*. Tato dokumentace opět odpovídá verzi *HaveAPI*, kterou *API* právě využívá.

6.17 Nasazení

HaveAPI používá *Rack* [14], tj. rozhraní mezi web serverem a aplikacemi v *Ruby*. Pro vývoj se často používá např. *WEBrick*, což je *HTTP* server v *Ruby*. Pro nasazení v produkci pak existují další web servery, jako *Thin* [15] a *Unicorn* [16], které už dokáží využít a spravovat více vláken/procesů a jsou tak vhodnější k reálnému nasazení.

Pro aplikace používající *Rack* je důležitý soubor `config.ru`, pomocí kterého předáme naši aplikaci (*API*) web serveru, viz ukázka 13. Případně můžeme web server spustit příkazem `rackup` z adresáře, ve kterém se nachází soubor `config.ru`.

```ruby
require 'haveapi'

# Modul se zdroji
module Resources ; end

api = HaveAPI::Server.new(Resources)

# ... inicializace ...

# Registrace rout akcí
api.mount('/')

run api.app
```

Ukázka kódu 13: Předání aplikace web serveru

6.18 Testování API serveru

Framework je testován pomocí *RSpec* [18], což je nástroj pro behaviour-driven development (*BDD*), tzn. testování chování kódu. Framework poskytuje funkce v modulu `HaveAPI::Spec`, pomocí nichž může uživatel snadno testovat své *API*. Jsou zde metody pro inicializaci *API*, vybírání verzí, volání akcí a zpracování odpovědí, více viz referenční dokumentace v sekci 6.19.

6.19 Referenční dokumentace

Referenční dokumentace je generována ze zdrojových kódů pomocí nástroje *YARD* [17]. Je k dispozici na přiloženém CD, viz příloha A.

6.20 Šablony projektů a ukázky

K dispozici jsou ukázky implementace *API* serverů a šablony projektu, viz příloha A.

Šablony obsahují autorem doporučenou adresářovou strukturu a soubory potřebné k definici závislostí, spouštění testů a *API* serveru. Šablona `basic` obsahuje jen inicializaci *HaveAPI*, není zde žádný model. Šablona `activerecord` používá jako model ActiveRecord a obsahuje příkazy pro tvorbu databázových migrací, apod., stejně jako *Ruby on Rails*.

Důležité informace o každé šabloně jsou vždy v souboru `README.md`. Pro použití šablony stačí celý adresář zkopírovat.

7 Implementace klientů

Klienti jsou implementovaní v podobě knihoven v různých programovacích jazycích. Klient se umí připojit k *API*, stáhnout dokumentaci a poskytnout uživateli rozhraní k práci s *API*. Nad těmito knihovnami jsou potom postaveny složitější aplikace, jako rozhraní v příkazové řádce, webová administrace či virtuální souborový systém.

7.1 Pravidla pro implementaci klientů

Základem pro *HaveAPI* klienty jsou tyto tři pravidla:

- klient nemá o *API* žádné předpoklady a specifický kód pro nějaké konkrétní *API*,
- klient v *API* nevyžaduje žádné zdroje, které nejsou součástí protokolu,
- všechno, co o *API* klient ví, se dozví z jeho dokumentace.

Pokud není některé pravidlo splněno, klient nedokáže pracovat s jakýmkoli *API*, které je založeno na *HaveAPI* protokolu a tím ztrácí onu výhodu znovupoužitelnosti.

Klient by měl využívat přirozených prostředků programovacího jazyka (např. objekty v objektově orientovaných jazycích) a snažit se přiblížit stylu užití ostatních klientů, příklad viz ukázka 14.

```
// Vytvoření instance klienta
api = new HaveAPI.Client("https://your.api.tld")

// Autentizace
api.authenticate(<metoda>, <volby>)
api.authenticate("basic", {"user": "login", "password": "heslo"})

// Přístup k objektům a akcím
api.<zdroj>.<akce>( { <parametry> } )
api.user.create({"name": "Jméno", "password": "tajne"})
api.user.index()
api.zdroj.zanořený_zdroj.další_zdroj.akce()

// Nastavení URL parametrů akcí
api.user.car.show(1, 2)

// Objektový přístup
user = api.user.show(1)
print user.id
print user.name
user.delete()
```

Ukázka kódu 14: Ukázka užití klienta v pseudokódu

Klient nemusí nutně implementovat všechny funkce protokolu, aby byl použitelný. Mezi důležité funkce patří:

- stažení a porozumění dokumentaci *API*,
- práce se stromem zdrojů a akcí,
- vstupní/výstupní parametry,
- autentizační metody,
- objektový přístup ke zdrojům v *API* (pokud to jazyk umožňuje).

Zbývající funkce jsou volitelné:

- validace parametrů na straně klienta,
- metadata parametry,
- blokující akce.

7.2 Ruby

Klient v *Ruby* je distribuován pomocí *RubyGems.org* [19] a vyžaduje *Ruby* minimálně ve verzi *2.0*. Instaluje se příkazem `gem install haveapi-client`.

Tento klient implementuje všechny vlastnosti protokolu *HaveAPI*. Klient při inicializaci vyžaduje jen *URL* k *API* serveru. Stáhne dokumentaci výchozí nebo zvolené verze *API* a podle toho se nastaví. Klient za běhu vytváří metody, pomocí kterých se přistupuje ke zdrojům a akcím v daném *API*.

Příklad užití klienta s *API*, které má zdroj `todolist` s akcemi *Index* a *Show* můžeme vidět na ukázce 15.

```ruby
require 'haveapi/client'

api = HaveAPI::Client.new('https://your.api.tld')

# Po autentizaci je stažena dokumentace a klient je nastaven
api.authenticate(:basic, user: 'yourname', password: 'yourpassword')

# Výpis 10 položek
puts api.todolist.index(limit: 10)

# Výpis položky s ID 1
puts api.todolist.show(1)
```

Ukázka kódu 15: Použití *Ruby* klienta

7.3 PHP

Klient v *PHP* je distribuován pomocí programu *composer* [20] s repozitářem *Packagist.org* [21].

Tento klient implementuje všechny části protokolu, kromě validátorů parametrů na straně klienta a blokujících akcí. V použití blokujících akcí pomocí zdroje `action_state` však nijak nebrání, pouze to není integrováno.

Způsobem použití se podobá klientu v *Ruby*. Taktéž za běhu vytváří metody pro přístup ke zdrojům a akcím, ale jiným způsobem. Klient implementuje *magické* metody `__call`, `__get` a `__set`, které interpret *PHP* volá, když dojde k přístupu k neexistující metodě nebo atributu. Klient zkontroluje, zda se nejedná o zdroj, akci, či parametr a podle toho se zachová.

Druhou možností přístupu ke zdrojům a akcím je přes indexaci, protože klient implementuje rozhraní `ArrayAccess`.

Příklad použití se stejným *API* jako u *Ruby* klienta viz ukázka 16.

```php
<?php
$api = new \HaveAPI\Client('https://your.api.tld');

// Po autentizaci je stažena dokumentace a klient je nastaven
$api->authenticate('basic', [
    'user' => 'yourname', 'password' => 'yourpassword'
]);

// Výpis 10 položek, různé možnosti přístupu
var_dump( $api->todolist->index(['limit' => 10]) );
var_dump( $api['todolist']->index(['limit' => 10]) );
var_dump( $api['todolist']['index']->call(['limit' => 10]) );
var_dump( $api['todolist.index']->call(['limit' => 10]) );

// Výpis položky s ID 1
var_dump( $api->todolist->show(1) );
```

Ukázka kódu 16: Použití *PHP* klienta

7.4 JavaScript

Klient v *JavaScriptu* je distribuován buď jako samostatný soubor `haveapi-client.js`, přes *NPM* z *NodeJS* [22] nebo *bower* [23]. Funguje tedy ve webových prohlížečích i prostředí *NodeJS*.

JavaScript je od *Ruby* či *PHP* odlišný v tom, že na dokončení operace se nečeká – funkce neblokují – ale o výsledku se dozvídáme pomocí callbacku, tj. funkce, která je spuštěna při dokončení operace. Klient tohoto stylu využívá u všech operací, které komunikují se serverem.

I tento klient dynamicky vytváří atributy, pomocí kterých se přistupuje ke zdrojům a akcím. Dokonce jde tak daleko, že pokud k akci přistoupíme jako k aktributu, tj. bez kulatých závorek, dostaneme objekt reprezentující akci. Pokud ale akci zavoláme jako funkci, tzn. s kulatými závorkami, akce se rovnou spustí. Tohoto chování je docíleno pomocí dynamické změny prototypu akce, viz zdrojové kódy v příloze A. Klient v *JavaScriptu* implementuje všechny vlastnosti protokolu.

Příklad použití se stejným *API* jako u předchozích klientů viz ukázka 17.

```
var HaveAPI = require('haveapi-client');

var api = new HaveAPI.Client("http://your.api.tld");

api.authenticate('basic', {
        username: 'yourname',
        password: 'yourpassword'
    }, function(c, status) {
    // Autentizace dokončena, můžeme volat akce

    // Výpis 10 položek různými způsoby
    api.todolist.index({limit: 10}, function (c, reply) {
        console.log(reply);
    });

    api.todolist.index.invoke({limit: 10}, function (c, reply) {
        console.log(reply);
    });

    // Výpis jedné položky
    api.todolist.show(1, function (c, reply) {
        console.log(reply);
    });
});
```

Ukázka kódu 17: Použití *JavaScript* klienta v prostředí NodeJS

7.5 Rozhraní v příkazové řádce

Rozhraní v příkazové řádce, nebo-li *CLI*, je založeno na klientu v *Ruby*. *CLI* a knihovna v *Ruby* jsou ve stejném repozitáři a jsou distribuováni společně.

Přes *CLI* je možné zavolat jakoukoli akci, kromě těch, které mají jako vstupní formát `hash_list` nebo `object_list`. Název zdroje, akce a parametry se předávají pomocí argumentů a přepínačů. Výstup je v případě jedné položky formátován do řádků a pokud je položek více, tak do sloupců. Pomocí přepínačů můžeme využívat různé metody autentizace (i s možností uložit údaje na disk), řadit dle vybraných parametrů, vybírat si parametry k zobrazení či formáty datumů.

V případě blokujících akcí klient čeká, dokud akce není hotova a zobrazuje průběh vykonání akce. Toto čekání je možné bezpečně přerušit či změnit přepínačem.

CLI je přizpůsobitelné, např. je možné přidat speciální příkazy pro konkrétní *API*, které kombinují volání různých akcí, nebo obsahují jinou logiku. Uživatel by neměl přepisovat kód tohoto rozhraní, ale rozšířit ho ve svém programu, který má na *CLI* závislost. Tzn. vlastní příkazy nebudou v `haveapi-cli`, ale v `mojeapi-cli`.

Příklad práce s *CLI* viz ukázka 18.

```
# Autentizace, na heslo se program zeptá na standardním vstupu
$ haveapi-cli -u https://your.api.tld \
              --auth basic --username yourname \
              todolist index

# Uložení autentizačních údajů do souboru přepínačem '--save'.
# Aby nebylo na disku uloženo heslo v čistém textu, použijeme autentizaci
# přes tokeny.
$ haveapi-cli -u https://your.api.tld \
              --auth token --username yourname \
              --save \
              todolist index

# URL API serveru a autentizační údaje nyní nemusíme zadávat.
# Přepínače vstupních parametrů jsou od přepínačů klienta odděleny dvěmi
# pomlčkami (--).
$ haveapi-cli todolist index -- --limit 10

# Výpis jedné položky
$ haveapi-cli todolist show 1
```

Ukázka kódu 18: Použití rozhraní v příkazové řádce

7.6 Virtuální souborový systém

Virtuální souborový systém *haveapi-fs* umožňuje pracovat s *API* na úrovni adresářů a souborů. Zdroje a akce jsou adresáře, vstupní a výstupní parametry pak soubory, do kterých může uživatel zapisovat a číst. *haveapi-fs* je založen na klientu v *Ruby* a distribuován přes *RubyGems.org* [19]. Běží jen na systémech s *FUSE* [24] (Filesystem in Userspace) a *RFuse* [25], což bylo zatím testováno jen na *GNU/Linux*.

haveapi-fs dokáže zpřístupnit jen zdroje, které definují akce *Index* a *Show* mající výstupní parametr `id` identifikující jednotlivé objekty ve zdroji.

V každém adresáři najdeme soubory `help` v různých formátech: text, *Markdown*, *HTML* či manuálové stránky. Tyto soubory obsahují informace o adresáři, ve kterém se nachází.

```
$ haveapi-fs https://your.api.tld /mnt/your.api
User name: jmeno
Password: heslo
```

Obrázek 22: Připojení *API*

API připojíme do adresáře `/mnt/your.api` příkazem `haveapi-fs`, viz ukázka 22. Adresář *API* obsahuje seznam zdrojů a soubory s informacemi o souborovém systému (viz ukázka 19). Adresář zdroje (ukázka 20) potom seznam objektů z akce *Index*, filtry pomocí vstupních parametrů přes virtuální adresáře (např. adresář `by-limit/`). Adresář konkrétního objektu (ukázka 21) pak výstupní parametry akce *Show*.

Soubory `create.yml` a `edit.yml` jsou speciální soubory, pomocí kterých můžeme vyvolat akci `Create`, resp. `Update`. Tyto soubory obsahují vstupní parametry dané akce ve formátu *YAML* [26]. Po zapsání a uložení dojde k vykonání příslušné akce se zadanými vstupními parametry.

Zajímavý je ještě způsob, jakým se manuálně volají akce. Existují dvě možnosti: zapsáním jedničky do souboru `exec` v adresáři akce, nebo jej můžeme spustit. `exec` je totiž také spustitelný skript v *Ruby*. Pokud využijeme tohoto spustitelného souboru, operační systém vytvoří nový proces. Ten však potřebuje komunikovat s procesem souborového systému. Toho je docíleno pomocí virtuálního souboru `.remote_control` v adresáři *API*, který se chová podobně jako lokální soket. Spuštěný proces řekne souborovému systému, jakou akci má zavolat a pak čeká na odpověď.

haveapi-fs je taktéž rozšiřitelný na podobném principu jako *CLI*. Uživatel může nahradit a přidat libovolné komponenty, ze kterých je poskládáná stromová struktura reprezentující adresáře a soubory.

```
/mnt/your.api/
├── todolist/
├── help.{txt,md,html,man}
├── .client_version
├── .fs_version
├── .protocol_version
├── .reset
└── .unsaved
```

Ukázka kódu 19: Adresář *API*

```
/mnt/your.api/todolist/
├── 1/
├── 2/
├── 3/
├── actions/
├── by-limit/
├── create.yml
└── help.{txt,md,html,man}
```

Ukázka kódu 20: Adresář zdroje

```
/mnt/your.api/todolist/1/
├── actions/
├── id
├── title
├── text
├── edit.yml
├── help.{txt,md,html,man}
└── save
```

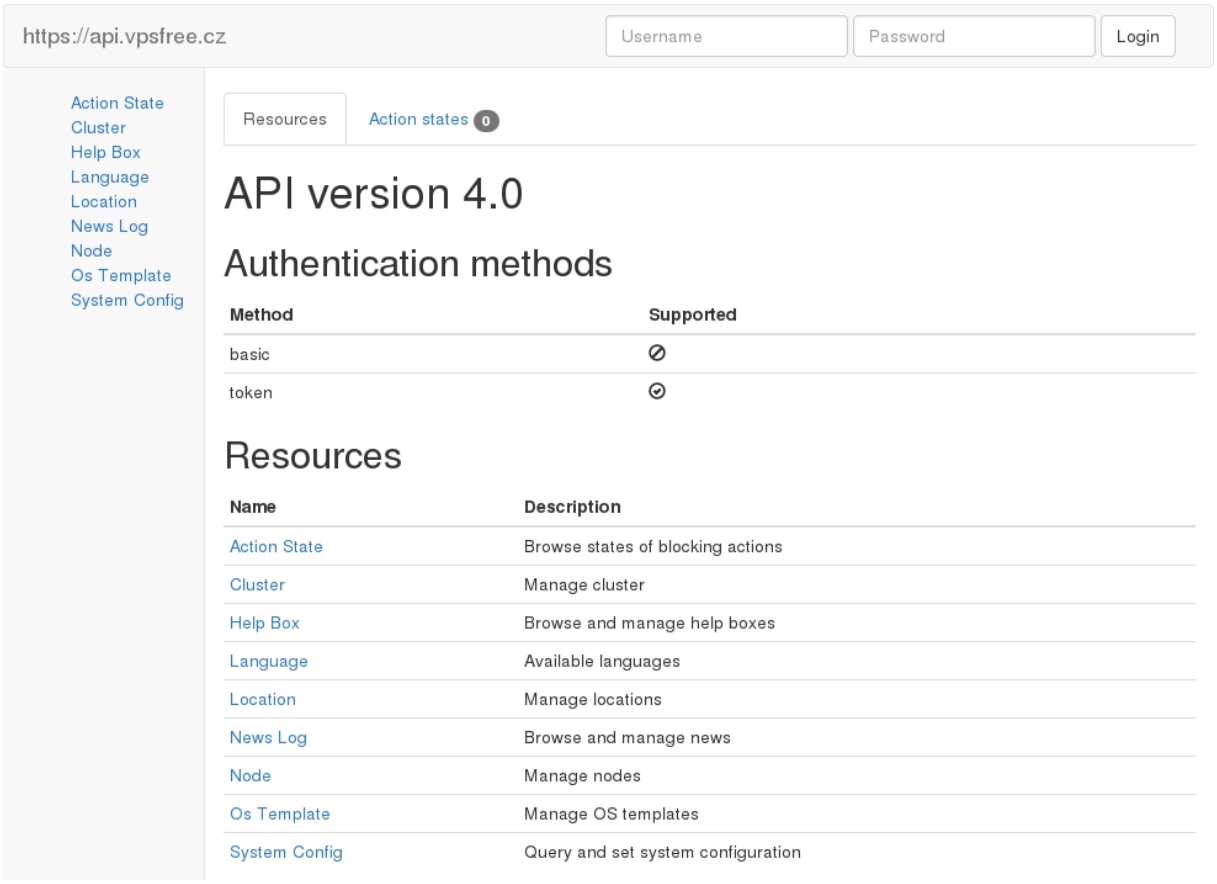
Ukázka kódu 21: Adresář objektu

7.7 Webová administrace

Webová administrace (dále *HaveAPI WebUI*) je tzn. jednostránková aplikace (*SPA*) v *JavaScriptu*, založena na klientu ve stejném jazyce a knihovně *React* [27] fungující plně na straně klienta ve webovém prohlížeči. Aplikaci je potřeba ze zdrojových kódů sestavit. Výslednou aplikaci lze ovlivnit konfiguračním souborem. Podle toho, jestli chceme aplikaci využívat lokálně nebo z web serveru můžeme zvolit např. způsob uchování historie. Sestavit ji lze pro konkrétní *API* (*URL API* je na pevno v kódu), nebo obecně pro jakékoli *API*.

HaveAPI WebUI podporuje pouze autentizační metodu *token*, neboť *HTTP basic* v prohlížeči není bezpečný. Stejně jako souborový systém dokáže procházet jen zdroje, které mají akce *Index* a *Show* s výstupním parametrem `id`.

Ukázka *HaveAPI WebUI* připojeného k *API* viz obrázek 23. V horní liště se zobrazuje formulář pro autentizaci, v levém postranním panelu pak seznam zdrojů v *API*, se kterými můžeme pracovat. V tabu *Resources* se zobrazuje přehled *API*, nebo zvolený zdroj/akce. Tab *Action states* slouží ke sledování blokujících akcí (viz sekce 4.13).



Obrázek 23: *HaveAPI WebUI* připojeno k `https://api.vpsfree.cz`

8 Výkonnostní testování

Cílem testování je porovnat výkon různých knihoven a aplikací implementující vytvořený protokol. Pro účely testování byl vytvořen jednoduchý *API* server, obsahující jeden zdroj se dvěma akcemi pro výpis seznamu objektů a výpis jednoho konkrétního objektu. Data jsou vygenerovány při startu serveru a zůstanou uloženy v paměti.

Použité verze programů:

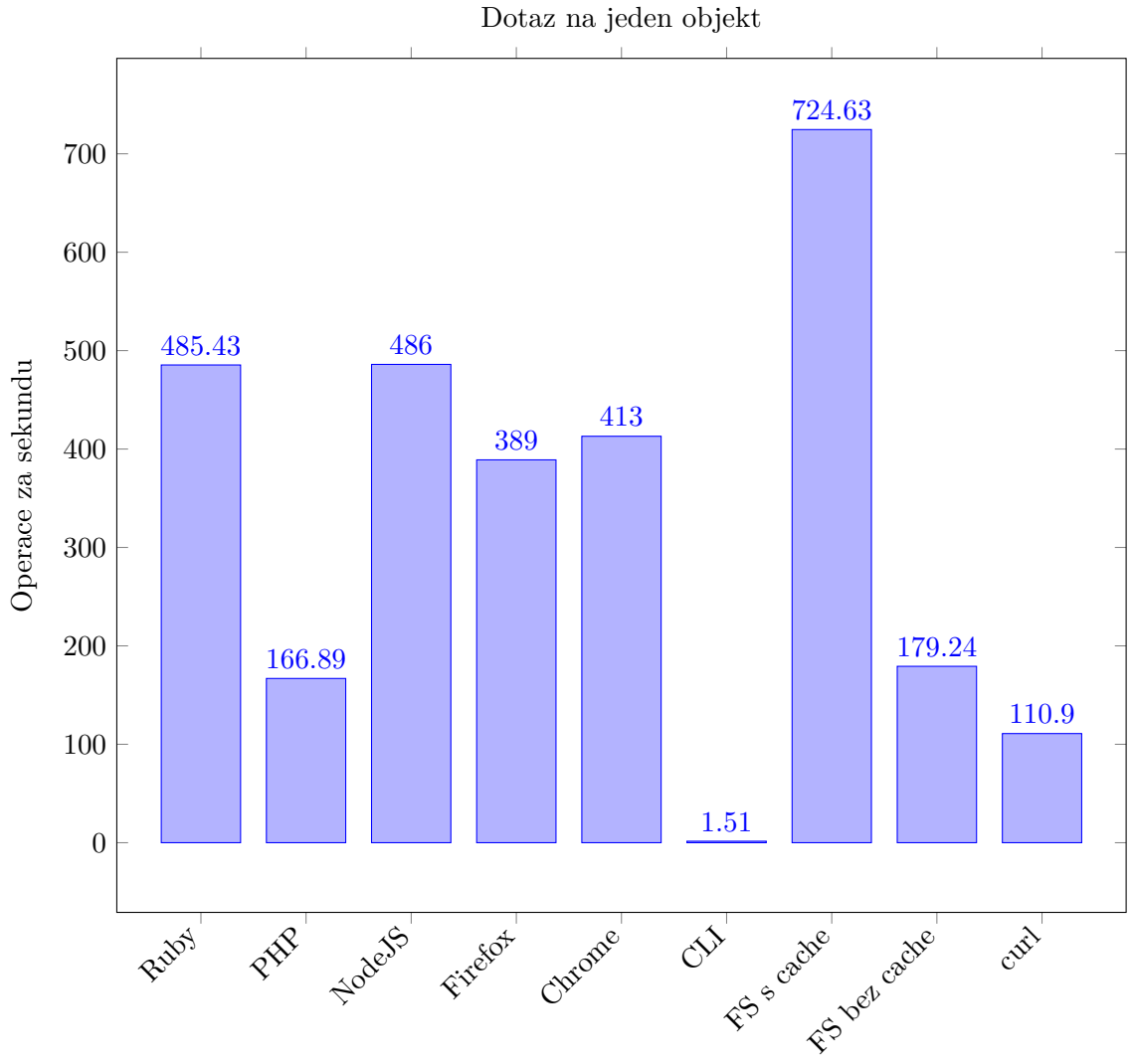
- *HaveAPI v0.9.0* s *Thin v1.7.0*
- *Ruby 2.1.9p490*
- *PHP 5.6.30-pl0-gentoo*
- *NodeJS v6.9.4*
- *Firefox 45.7*
- *Chrome 56*
- *curl 7.52.1*

Testy byly spuštěny postupně na stejném, jinak nezatíženém systému. Testovací *API* server a skripty testující jednotlivé klienty jsou k dispozici na CD, viz příloha A. Na grafech 24 a 25 můžeme vidět počty operací za sekundu, které zvládá klientská knihovna v *Ruby* a *PHP*, knihovna v *JavaScriptu* v různých běhových prostředích, rozhraní v příkazové řádce, virtuální souborový systém s a bez interní cache a program *curl*.

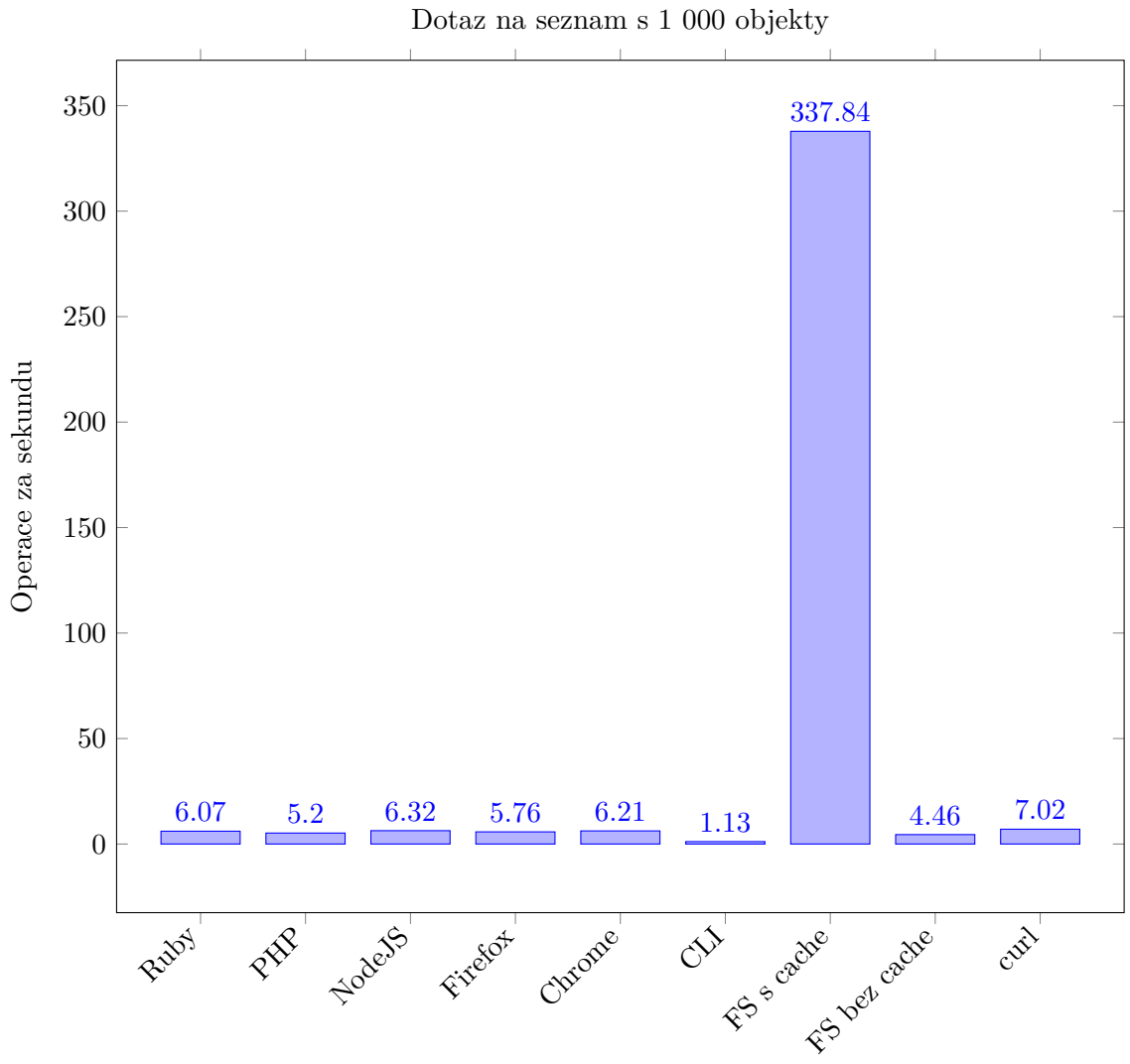
Špatné výsledky klienta v *CLI* jsou způsobeny stahováním dokumentace *API*. Jelikož není implementována žádná cache, do které by se stažená dokumentace ukládala, klient si ji musí při každém spuštění stáhnout znovu. Ostatní programy stahovaly dokumentaci *API* pouze jednou.

haveapi-fs jako jediný obsahuje cache pro stažené objekty a díky tomu dosahuje nejlepších výsledků. Do cache se ukládají všechny vytvořené entity reprezentující adresáře/soubory, resp. zdroje, akce, parametry z *API* a tím pádem všechny stažené data.

Program *curl* je příklad klasického *HTTP* klienta, který o *HaveAPI* protokolu nic neví. Na rozdíl od *CLI* z *HaveAPI* tedy nestahuje dokumentaci a *URL* akcí (a případné vstupní parametry) jsou zadány manuálně. Díky tomu dosahuje vyššího počtu operací za sekundu.



Obrázek 24: Operace za sekundu při dotazu na jeden objekt



Obrázek 25: Operace za sekundu při dotazu na seznam objektů

9 Závěr

HaveAPI je od rané fáze vývoje použito jako základ pro *API* projektu *vpsAdmin*, což je administrační rozhraní pro správu virtuálních serverů, souvisejících datových úložišť a sítě. *vpsAdmin* využívá celého ekosystému *HaveAPI*: webové rozhraní používá klientské knihovny v *PHP* a *JavaScriptu*, části systému využívají knihovnu v *Ruby* a rozhraní v příkazové řádce je rozšířeno o spefickou funkcionalitu, jako např. vzdálená konzole k virtuálnímu serveru nebo monitor datových přenosů. *vpsAdmin* je vyvíjen primárně mnou, tj. autorem této práce, v rámci spolku *vpsFree.cz* [28], jehož jsem členem, kde slouží jako centrální informační systém. V době odevzdání této práce se *vpsAdmin* ve *vpsFree.cz* [29] stará přibližně o 1 200 uživatelů a 1 600 virtuálních serverů.

Samozřejmě, výhody *HaveAPI* by se naplno projevily, kdyby došlo k jeho využití i v jiných projektech, případně začaly vznikat další implementace serveru či klientů, které by rozšířily portfolio podporovaných programovacích jazyků a prostředí. Pak teprve vynikne schopnost znovupoužití existujících knihoven a aplikací pro kterékoli *API*. V rámci této snahy jsme společně s kolegou z *vpsFree.cz* měli o *HaveAPI* přednášku na konferenci *InstallFest 2016*, kde jsem vysvětloval podstatu *HaveAPI* a Pavel Šnajdr se postaral o živou ukázku, jednoduché *API* ovládající piny na jednodeskovém počítači *Banana Pi M3*.

Vytvořený protokol je ze zkušenosti z provozu dostačující. To však neznamená, že by byl hotový. Z hlediska specifikace protokolu by se dala vylepšit situace s formátem vstupních a výstupních parametrů (viz sekce 4.7). Rozdělení na `hash` a `object` bylo zavedeno zejména proto, abychom mohli klientům nařídit, zda vrátit odpověď v nějakém objektu obalujícím zdroj v *API* (`object`), nebo v jednoduchém kontejneru (mapa, pole map; `hash`) a také kvůl asociacím mezi zdroji, které jsou napojeny na *ORM*. Asociace tak fungují jen mezi zdroji, jejichž akce *Index* a *Show* používají formát `object`. Reprezentace zdroje by spíše měla být ponechána na klientovi, případně na jeho uživateli a formát parametrů by neměl nic vypovídat o implementaci *API*, tj. jestli je použito *ORM*, nebo ne. Dále by bylo vhodné dodělat podporu pro přenos proudů dat ze serveru ke klientům.

Referenční implementace zatím nejvíc trpí chybějící podporou pro cache. Dokumentace *API* se stahuje při každém spuštění aplikace, ne jen když dojde ke změně. U programu typu rozhraní v příkazové řádce to má zásadní vliv na výkon (viz sekce 8). Pro správnou funkci bude muset být vyřešena integrace s modelem, aby *HaveAPI* vědělo např. jaká je maximální doba platnosti odpovědi, nebo jestli má být pro daný zdroj cache vůbec použita.

S vývojem protokolu a referenční implementace hodlám dále pokračovat.

Literatura

[1] FIELDING, R. T. et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616, RFC Editor, 1999. Dostupné z: http://www.rfc-editor.org/rfc/rfc2616.txt

[2] FRANKS, J. et al. *HTTP Authentication: Basic and Digest Access Authentication*. RFC 2617, RFC Editor, 1999. Dostupné z: http://www.rfc-editor.org/rfc/rfc2617.txt

[3] BELSHE, et al. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540, RFC Editor, 2015. Dostupné z: http://www.rfc-editor.org/rfc/rfc7540.txt

[4] FIELDING, Roy. *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irving, 2000. Disertace. Dostupné z: http://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm

[5] *HAL - Hypertext Application Language: A lean hypermedia* type [online]. Mike Kelly, 2011 [cit. 2017-04-15]. Dostupné z: http://stateless.co/hal_specification.html

[6] *Standard ECMA-404: The JSON Data Interchange Format*. 1. Geneva: Ecma International, 2013.

[7] *ISO 8601:2004*. 2004. Geneva: International Organization for Standardization, 2004.

[8] THE RUBY COMMUNITY. *Ruby Programming Language* [online]. 1999 [cit. 2017-03-22]. Dostupné z: https://www.ruby-lang.org/

[9] THOMAS, David, Chad FOWLER a Andrew HUNT. Programming Ruby 1.9: the pragmatic programmers' guide. 4. Pragmatic Bookshelf, 2013. Pragmatic programmers. ISBN 978-1-93778-549-9.

[10] *Standard ECMA-262: ECMAScript® 2016 Language Specification*. 1. Geneva: Ecma International, 2016.

[11] THE PHP GROUP. *PHP: Hypertext Preprocessor* [online]. 2001 [cit. 2017-03-22]. Dostupné z: http://www.php.net/

[12] MIZERANY, Blake. *Sinatra* [online]. [cit. 2017-03-22]. Dostupné z: http://www.sinatrarb.com

[13] *Ruby on Rails: A web-application framework that includes everything needed to create database-backed web applications according to the Model-View-Controller (MVC) pattern* [online]. [cit. 2017-03-22]. Dostupné z: http://rubyonrails.org

[14] *Rack: a Ruby Webserver Interface* [online]. Neukirchen, 2007 [cit. 2017-03-22]. Dostupné z: https://rack.github.io

[15] *Thin: A fast and very simple Ruby web server* [online]. Cournoyer [cit. 2017-03-22]. Dostupné z: http://code.macournoyer.com/thin/

[16] *Unicorn: Rack HTTP server for fast clients and Unix* [online]. [cit. 2017-03-22]. Dostupné z: https://bogomips.org/unicorn/

[17] *YARD: A Ruby Documentation Tool* [online]. Segal, 2007 [cit. 2017-03-22]. Dostupné z: http://yardoc.org

[18] *RSpec: Behaviour Driven Development for Ruby.* [online]. 2005 [cit. 2017-03-22]. Dostupné z: http://rspec.info

[19] *RubyGems.org* [online]. 2009 [cit. 2017-03-22]. Dostupné z: https://rubygems.org/

[20] *Composer: Dependency Manager for PHP* [online]. [cit. 2017-03-22]. Dostupné z: https://getcomposer.org

[21] *Packagist: The PHP Package Repository* [online]. 2011 [cit. 2017-03-22]. Dostupné z: https://packagist.org

[22] *Node.js* [online]. Node.js Foundation, 2017 [cit. 2017-03-22]. Dostupné z: https://nodejs.org

[23] *Bower: A package manager for the web* [online]. 2012 [cit. 2017-03-22]. Dostupné z: https://bower.io/

[24] *Libfuse: The reference implementation of the Linux FUSE (Filesystem in Userspace) interface* [online]. [cit. 2017-03-22]. Dostupné z: https://github.com/libfuse/libfuse

[25] *RFuse: Ruby FUSE bindings - write Filesystems in Ruby* [online]. [cit. 2017-03-22]. Dostupné z: https://github.com/lwoggardner/rfuse

[26] *YAML: YAML Ain't Markup Language* [online]. Evans [cit. 2017-03-29]. Dostupné z: http://yaml.org

[27] *React: A JavaScript library for building user interfaces* [online]. Facebook, 2017 [cit. 2017-03-22]. Dostupné z: https://facebook.github.io/react/

[28] *VpsFree.cz: Virtuální Privátní Servery svobodně* [online]. 2009 [cit. 2017-03-29]. Dostupné z: https://vpsfree.cz

[29] *VpsAdmin* [online]. 2008 [cit. 2017-03-29]. Dostupné z: https://vpsadmin.vpsfree.cz

A Příloha na CD

- `Priloha/`
  - `benchmarks/` ..... Skripty a server pro výkonnostní testování
    - `server/` ..... Testovací server
    - `README.md` ..... Instrukce pro spuštění testů
    - `test.*` ..... Jednotlivé testy
  - `doc/` ..... Referenční dokumentace v *HTML*
  - `haveapi/` ..... Git repozitáře se zdrojovými kódy
    - `haveapi.git` ..... Server framework
    - `haveapi-client.git` ..... Klient a *CLI* v *Ruby*
    - `haveapi-client-js.git` ..... Klient v *JavaScriptu*
    - `haveapi-client-php.git` ..... Klient v *PHP*
    - `haveapi-webui.git` ..... Webová administrace
    - `haveapi-fs.git` ..... Virtuální souborový systém
    - `haveapi-server-project-templates.git` ..... Šablony projektů
    - `haveapi-server-examples.git` ..... Ukázky implementace *API* serveru