VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

NÁSTROJ PRO DOKUMENTACI DYNAMICKY ROZŠIŘITELNÝCH DSL V RUBY

EXTENSIBLE DSL DOCUMENTATION TOOL IN RUBY

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **OLEH FEDORENKO**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. ZBYNĚK KŘIVKA, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Fedorenko Oleh**
Program: Informační technologie
Název: **Nástroj pro dokumentaci dynamicky rozšiřitelných DSL v Ruby**
Dynamic Extensible DSL Documentation Tool in Ruby
Kategorie: Překladače

Zadání:

1. Seznamte se s problematikou dokumentace dynamických jazyků a doménově specifických jazyků (DSL).
2. Seznamte se s již existujícími nástroji pro dokumentaci DSL se zaměřením na jazyk Ruby.
3. Seznamte se s nástrojem Apipie-rails pro dokumentaci RESTful API a dle konzultací navrhněte modifikaci nebo zcela nový nástroj umožňující dokumentování různých DSL.
4. Dle návrhu nástroj implementujte.
5. Své řešení otestujte v rámci projektu Foreman, zhodnoťte a diskutujte další možná rozšíření.

Literatura:

- M. Fowler: Domain-Specific Languages, Addison-Wesley Professional (2010)
- P. Perrotta: Metaprogramming Ruby 2: Program Like the Ruby Pros, Pragmatic Bookshelf (2014)
- S. McConnell: Code Complete, Microsoft Press (2004, 2nd edition)
- Apipie/apipie-rails: Ruby on Rails API documentation tool. Dostupné na https://github.com/Apipie/apipie-rails [cit. 2018-09-25]
- Foreman. Dostupné na https://www.theforeman.org/ [cit. 2018-09-25]

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Křivka Zbyněk, Ing., Ph.D.**
Konzultant: Nečas Ivan, Mgr., RedHatCZ
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 14. května 2020
Datum schválení: 16. října 2019

Abstrakt
Cílem práce je navrhnout a realizovat modifikaci stávajícího nebo úplně nový dokumentační nástroj, který poskytuje cestu k dokumentaci dynamicky rozšiřitelných DSL v Ruby. Řešení je založeno na již existujícím nástroji Apipie-rails pro dokumentaci RESTful API.

Abstract
The aim of this thesis is to design and implement a modification of existing or a completely new documentation tool which provides a way for documentation of dynamic extensible DSLs in Ruby. The solution is based on already existing tool Apipie-rails for RESTful API documentation.

Klíčová slova
Ruby, Dokumentace, Doménově specifický jazyk

Keywords
Ruby, Documentation, Domain Specific Language

Citace
FEDORENKO, Oleh. *Nástroj pro dokumentaci dynamicky rozšiřitelných DSL v Ruby.* Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Zbyněk Křivka, Ph.D.

Nástroj pro dokumentaci dynamicky rozšiřitelných DSL v Ruby

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Zbyňka Křivky, Ph.D. Další informace mi poskytli Mgr. Ivan Nečas a Mgr. Marek Hulán. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Oleh Fedorenko
28. května 2020

Poděkování

Zde bych chtěl poděkovat vedoucímu této práce Ing. Zbyňkovi Křivkovi, Ph.D. za konzultace, trpělivost a praktické rady potřebné pro psaní této práce. Zvlášť bych chtěl poděkovat Mgr. Ivanovi Nečasovi a Mgr. Markovi Hulánovi z firmy Red Hat Czech s.r.o., kteří mi poskytli teoretické informace, praktické rady a testovací prostředí.

Obsah

1 Úvod **2**

2 Dokumentace doménově specifických jazyků **3**
- 2.1 Doménově specifický jazyk . . . 3
- 2.2 Dokumentace . . . 4
  - 2.2.1 Komentáře . . . 5
  - 2.2.2 Kód . . . 5
- 2.3 Ruby . . . 6

3 Analýza a návrh **9**
- 3.1 Analýza nástroje Apipie-rails . . . 9
- 3.2 Specifikace požadavků . . . 12
- 3.3 Návrh nového nástroje . . . 13

4 Implementace **16**
- 4.1 Struktura nástroje . . . 16
- 4.2 Řešení implementačních problémů . . . 23

5 Aplikace nástroje **25**
- 5.1 Foreman . . . 25
  - 5.1.1 Šablony . . . 25
  - 5.1.2 Další možnosti integrace . . . 28

6 Závěr **29**

Literatura **30**

A Ukázky DSL nástroje Apipie-rails **31**

B Ukázky výsledné dokumentace Apipie-rails **35**

C Obsah přiloženého paměťového média **37**

Kapitola 1

Úvod

Hlavním cílem mojí práce je navrhnout a realizovat modifikaci stávajícího[1] nebo úplně nový dokumentační nástroj, který poskytuje cestu k dokumentaci dynamicky rozšiřitelných DSL v programovacím jazyce Ruby. Samotná aplikace musí poskytnout vývojářům doménově specifických jazyků lehký a srozumitelný způsob dokumentace jejich práce. Výsledek činnosti aplikace by měl usnadnit uživatelům použití těchto jazyků, a to poskytováním zcela přehledné dokumentace v několika formátech.

Dokumentace je vždy nutnou součástí libovolného vybavení buď programového, nebo technického. Bohužel se občas stává, že dokumentace určitých aplikací přestává být aktuální jen kvůli tomu, že není způsob, kterým by bylo možné dynamicky přidat pár drobných úprav. Za určitých okolností se může také stát, že je potřeba se starat o několik variant dokumentací (např. kódu), ze kterých každá další má pouze drobné změny. V takovém případě by mohlo dojít ke nutnosti ručně opakovat stejné nebo téměř stejné úpravy ve více verzích dokumentací. Nástroj, který umožní mít aktuální dokumentaci přímo v kódu, by mohl pomoci vyřešit daný problém.

Aplikace je vlastně knihovnou, kterou lze použít jak v čistě Ruby aplikacích, tak i v aplikacích napsaných pomocí frameworku[2] Ruby on Rails. Knihovna poskytuje vlastní doménově specifický jazyk pro popis jiných doménově specifických jazyků. Dokumentace je pak přístupná v kódu hlavního programu. Taktéž pomocí určitého příkazu lze vygenerovat dokumentaci v podobě HTML stránek.

Kapitola 2 vysvětluje pojmy jako interní a externí DSL, upřesňuje, na jakou dokumentaci je kladen důraz, a dává stručný přehled jazyka Ruby. Kapitola 3 je věnována analýze, specifikaci požadavků a návrhu nového nástroje. Kapitola 4 popisuje implementaci navrženého nástroje, přijímaná rozhodnutí při vyskytnutí se problémů s použitím v Ruby on Rails aplikacích. V kapitole 5 jsou ukázky aplikace nástroje ve velkém *open source* projektu.

[1] Stávající systém byl postaven na Apipie-rails, který je popsán v sekci 3.1.

[2] Framework (aplikační rámec) je softwarová struktura, která slouží jako podpora při programování a vývoji jiných softwarových projektů. Může obsahovat podpůrné programy, knihovny API, podporu pro návrhové vzory nebo doporučené postupy při vývoji.

Kapitola 2

Dokumentace doménově specifických jazyků

V této kapitole jsou vysvětleny základní pojmy potřebné pro pochopení problému, na jehož řešení je tato práce zaměřena.

Jak již bylo naznačeno, výsledkem práce by měl být nástroj schopný zdokumentovat dynamicky rozšiřitelný doménově specifický jazyk v Ruby. V první sekci této kapitoly se vysvětluje samotný pojem DSL a upřesňuje se, na jaký typ je hlavní záměr. V druhé sekci jsou vysvětlovány základní problémy při dokumentování kódu, zvlášť při dokumentování pomocí komentářů. Poslední sekce stručně popisuje zvláštnosti použitého programovacího jazyka, tj. Ruby.

2.1 Doménově specifický jazyk

Obecně doménově specifický jazyk (angl. *Domain Specific Language*, dále jenom DSL) je programovací jazyk mající pevně definovanou orientaci na určitou oblast, doménu[5]. Má omezenou výpočetní sílu na rozdíl od univerzálního programovacího jazyka.

K této definici patří čtyři klíčové prvky:

- **Programovací jazyk:** pomocí DSL lidé přikazují počítači (buď imperativně nebo deklarativně), aby něco udělal. Stejně jako u každého moderního programovacího jazyka je jeho struktura navržena tak, aby byla snadno pochopitelná lidmi a stále by měla být něčím, co lze počítačově zpracovat.

- **Jazyková povaha:** DSL je programovací jazyk a jako takový by měl mít smysl pro plynulost, kde expresivita přichází nejen z jednotlivých výrazů, ale také ze způsobu, jakým se mohou skládat dohromady.

- **Omezená expresivita:** univerzální programovací jazyk poskytuje spoustu možností: podpora různých dat, struktury kontroly a abstrakce. To vše je užitečné, ale kvůli tomu je těžší se učit a používat tento jazyk. DSL podporuje minimální množství funkcí potřebných pro podporu své domény. V DSL nelze vytvořit celý softwarový systém; spíše se DSL používá pro jeden konkrétní aspekt systému.

- **Zaměření na doménu:** omezený jazyk je užitečný pouze v případě, že má jasné zaměření na konkrétní doménu. Zaměření na doménu je to, co dělá omezený jazyk užitečným.

Doménově specifické jazyky je možné rozdělit do dvou hlavních kategorií:

- Vnější (angl. *external*) DSL
- Vnitřní (angl. *internal*) DSL

External

Vnější čili externí DSL je jazyk oddělený od hlavního jazyka aplikace, se kterou pracuje. Obvykle externí DSL má vlastní syntaxi, ale také je běžná i syntaxe jiného tzv. metajazyka (XML je častá volba). Skript napsaný v externím DSL bude obvykle analyzován kódem v hostitelské aplikaci pomocí technik analýzy textu. Příklady externích DSL zahrnují regulární výrazy, SQL, Awk a konfigurační soubory XML pro různé systémy.

Internal

Vnitřní čili interní DSL je zvláštní způsob použití univerzálního programovacího jazyka. Skript v interním DSL je platný kód v rámci svého univerzálního jazyka, ale používá pouze podmnožinu vlastností jazyka v určitém stylu, aby zvládl pouze jeden malý aspekt celého systému. Výsledek by měl vyvolávat pocit spíše nového jazyka než jeho hostitelského. Příkladem interního DSL je nástroj Apipie-rails[9], kterému se věnuje sekce 3.1.

Práce se zaměřuje na dokumentování a použití vnitřních DSL v programovacím jazyce Ruby.

2.2 Dokumentace

Dokumentace je nezbytná pro efektivní vývoj, implementaci, úpravu, provoz a využití jakéhokoli systému. Dokumentaci k softwaru lze rozdělit následovně:

- **Požadavky:** prohlášení, která identifikují atributy, schopnosti nebo vlastnosti systému. To je základem toho, co bylo nebo bude realizováno.
- **Architektura/Návrh:** přehled softwaru. Zahrnuje vztahy k prostředí a stavebním základům, které budou použity v návrhu softwarových komponent.
- **Technická dokumentace:** dokumentace kódu, algoritmy, popis rozhraní a API[1].
- **Uživatelská dokumentace:** příručky pro koncového uživatele, systémové administrátory a osazenstvo podpory.
- **Marketingová dokumentace:** jak prodávat produkt a analýza tržní poptávky.

V této práci je kladen důraz na technickou dokumentaci, poněvadž nástroj by měl umožňovat snadné dokumentování DSL, tj. zdrojového kódu.

V současné době převážná většina dokumentace zdrojového kódu je vytvořena generováním textu na základě komentářů, pomocí kterých se popisují třídy, chování metod apod. Táto práce zavádí trochu jiný způsob, a to popis rozhraní, tříd apod. pomocí kódu.

[1]Aplikační programové rozhraní (angl. *Application Programming Interface*) obecně označuje rozhraní, pomocí kterého aplikace přistupuje k operačnímu systému a dalším službám. API je definováno na úrovni zdrojového kódu a poskytuje úroveň abstrakce mezi aplikací a jádrem (nebo jinými privilegovanými obslužnými programy), aby byla zajištěna přenositelnost kódu. Jde o sbírku procedur, funkcí, tříd či protokolů nějaké knihovny, programu nebo jádra operačního systému, které může programátor využívat.

2.2.1 Komentáře

Komentáře jsou jedním z nejčastěji používaných způsobů, kterým se programátoři vyjadřují při psaní kódu. Z pohledu člověka je to docela užitečná informace, která by měla určitým způsobem zjednodušit práci s kódem, který byl napsán velmi dávno nebo rovnou jiným člověkem.

Z pohledu kompilátoru, interpretu nebo obecně počítače je komentář obyčejný text, který nemá žádný vliv na práci programu, a proto je zcela vynechán za jeho běhu z kódu programu.

Hlavními výhodami takového přístupu jsou:

- **Snadnost použití:** člověk píše komentář v přirozeném jazyce.
- **Snadnost zpracování:** sice komentář je text pro člověka, ale každopádně se musí nějak oddělit od zbytku kódu, a to různé jazyky umožňují podobným způsobem: tento text se uvádí mezi určitými značkami, což pak umožňuje snadné zpracování pomocí jiných nástrojů. Tentýž nástroj dokáže zpracovat komentáře v různých programech, napsaných v různých jazycích, pokud ovšem tyto jazyky mají stejný význam oněch značek.
- **Režie:** nepodstatná režie navíc za běhu programu, protože většinou komentář není součástí spustitelného kódu.

Nicméně jsou následující nevýhody:

- **Znovupoužitelnost:** pokud je nutno nějaké části použít opakovaně, je potřeba komentář psát znovu. V nejjednodušším případě dochází ke kopírování kusů komentářů, což zbytečně zvětšuje objem zdrojového souboru.
- **Aktuálnost:** tento způsob dokumentování se moc nehodí v případě dynamicky rozšiřitelných kusů kódu (např. metod, které se dají rozšířit pomocí redefinice v rámci jiné třídy). Kvůli redefinici komentář ztrácí svou aktuálnost, protože původně byl napsán pro jinou verzi kódu.
- **Proces zpracování:** Někdy může být nevýhodou nutnost použití zcela jiné nástroje pro generování dokumentace.

2.2.2 Kód

Dokumentování pomocí kódu se snaží vyřešit zmíněné nevýhody dokumentování pomocí komentářů.

Poněvadž dokumentační kód je spustitelnou součástí celého programu, podprogram zodpovědný za interpretaci tohoto kódu může mít představu o aktuálním stavu i jiných částí hlavního programu, což může určitým způsobem usnadnit zachování aktuálnosti výsledné dokumentace. Navíc kvůli začlenění onoho podprogramu do hlavního programu (např. v roli knihovny), se ztrácí nutnost použití jiné aplikace pro generování dokumentace, stačí jen zavolat podprogram zodpovědný za generování.

V případě dobrého návrhu tohoto podprogramu neboli knihovny se vyřeší i problém znovupoužitelnosti: definovat popis jenom jednou a pak používat tuto definici dle nutnosti v různých částech programu. Tento přístup navíc usnadňuje případné změny onoho popisu, kde stačí pouze jednou upravit původní definici a zbytek se upraví automaticky. Na základě

dokumentace by se automaticky také mohla provádět kontrola typů parametrů předávaných do metody, případně hodnoty při návratu z funkce. To by se hodilo zejména u dynamicky typovaných jazyků jako je Ruby. Například programovací jazyk Python již má v sobě tzv. anotace funkcí (angl. *Function Annotations*), pomocí nichž lze popsat typ přijímaných parametrů a návratové hodnoty[11].

Další výhodou dokumentace pomocí DSL je syntaktická kontrola. V komentářové dokumentaci si člověk nemusí všimnout, že se přepsal v názvu proměnné nebo použil špatné klíčové slovo např. `@parma` místo `@param`.

Hlavními nevýhodami takového přístupu jsou:

- **(Ne)Snadnost použití:** sice programátor, který používá dokumentační kód, je seznámen s programovacím jazykem, ve kterém píše, stále se ale musí naučit nové DSL nebo API, pomocí kterých by byl schopen napsat dokumentaci.
- **(Ne)Snadnost zpracování:** napsaný dokumentační kód není pouhý text, který lze snadno zpracovat pomocí nástrojů pro zpracování komentářů.
- **Režie:** dokumentační kód je na rozdíl od komentářů spustitelnou součástí kódu hlavního programu, a proto se určitým způsobem musí zpracovat, což zavádí nějakou režii za běhu programu navíc.

2.3 Ruby

Ruby je dynamický programovací jazyk s lakonickou, ale výraznou gramatikou a vestavěnými knihovnami s bohatým a výkonným API. Ruby čerpá inspiraci z takových programovacích jazyků jako jsou Lisp, Smalltalk a Perl, ale používá gramatiku, která se programátorům C a Java snadno učí. Ruby je čistě objektově orientovaný jazyk, ale je také vhodný pro procedurální a funkcionální programovací styly. Podporuje metaprogramování, které lze použít k vytváření interních DSL[7].

Ruby má bohatou API pro *reflexi*. Reflexe, nazývaná také *introspekce*, jednoduše znamená, že program může zkoumat svůj stav a strukturu. Navíc Ruby podporuje tzv. *intercession* neboli kauzální změny za běhu, což dovoluje vkládat nové metody do tříd za běhu, vytvářet aliasy pro existující metody a dokonce definovat metody na jednotlivých objektech. Program napsaný v Ruby může například získat seznam metod definovaných určitou třídou, dotazovat hodnotu pojmenované instanční proměnné uvnitř daného objektu nebo iterovat přes všechny za běhu existující instance třídy regulárních výrazů `Regexp`. API reflexe ve skutečnosti jde dál a dovoluje programu měnit jeho stav a strukturu. Program napsaný v Ruby může dynamicky nastavovat pojmenované proměnné na základě dynamicky vytvořeného jména, vyvolávat pojmenované metody a dokonce definovat nové třídy a nové metody.

Program v Ruby může být napsaný s použitím několika interních DSL najednou, z nichž každý pak může být libovolně rozšiřován buď pomocí nějakého zvláštního rozhraní určité knihovny, nebo rovnou pomocí nativního rozhraní[6]. Ukázky typických způsobů rozšíření DSL v Ruby jsou uvedené na výpisu 2.1. Níže je taky uveden seznam 2.3 s popisy použitých metod pro rozšíření.

```
module MyModule
  def number
```

```
    puts 0
  end
end

class MyClass
  def number
    puts 42
  end
end

MyClass.prepend(MyModule) # Overrides MyClass#number method

mc = MyClass.new
mc.nubmer #=> prints 0

class MyClass
  include MyModule # In this example does nothing,
                   # since MyClass#number was already defined
end

mc = MyClass.new
mc.nubmer #=> prints 42

class MyClass
  extend MyModule
end

mc = MyClass.new
mc.nubmer #=> prints 42
MyClass.number #=> prints 0
```

Výpis 2.1: Ukázky možností rozšíření v Ruby.

- **Prepend** - Přidá konstanty, třídní proměnné a metody instancím třídy `MyClass` dostupné v modulu `MyModule`. Pokud nějaká určitá metoda byla definována v `MyClass` a v `MyModule`, metoda z `MyModule` překryje metodu v `MyClass`.
- **Include** - Přidá konstanty, metody a proměnné definované v modulu `MyModule` instancím třídy `MyClass`. Pokud dojde ke kolizím, tak při hledání se použije definice konstanty, metody nebo proměnné ze třídy `MyClass`.
- **Extend** - Přidá konstanty a metody definované v modulu `MyModule` do metatřídy třídy `MyClass`. Jednoduše, přidá třídní metody.

Zmíněné API reflexe jazyka Ruby — společně s jeho obecně dynamickou povahou, řídicími strukturami bloků a iterátorů a jeho syntaxi s volitelnými závorkami — dělá Ruby ideálním jazykem pro *metaprogramování* [2]. To se obzvlášť hodí při vytváření DSL nebo nástrojů s nimi pracujících. Díky zmíněným schopnostem Ruby k metaprogramování napsané

[2] Metaprogramování je psaní kódu, který manipuluje jazykovými konstrukcemi za běhu programu[8].

v něm DSL může být rozšířené nebo jakkoli změněné i za běhu programu. To ovšem může vést k tomu, že napsaná dokumentace k tomuto DSL po jeho změně ztratí aktuálnost. Dokonce může nastat případ, kdy není jiná možnost než dokumentovat metody DSL pomocí kódu.

Uvažujme jednoduchou situaci, kdy chceme, aby se metody generovaly dynamicky na základě nějakého seznamu od uživatele. To se dá jednoduše zvládnout pomocí nativních prostředků jazyka Ruby. Navíc chceme, aby uživatel měl k dispozici seznam těchto metod někde v dokumentaci. Kvůli tomu, že tyto metody neexistují ve skutečnosti (nejsou explicitně definované v kódu), není možné je popsat pomocí komentářů, protože dopředu nevíme ani jména metod. Právě proto je vhodné používat dokumentování pomocí kódu.

Kapitola 3

Analýza a návrh

První sekce této kapitoly se zabývá analýzou již existujícího nástroje `Apipie-rails` napsaného v Ruby pro dokumentaci RESTful API[3]. Další sekce popisuje motivaci pro vznik nového nástroje a specifikuje na něj požadavky. V poslední sekci je popsán návrh splňující uvedené požadavky.

3.1 Analýza nástroje Apipie-rails

`Apipie-rails` je DSL pro dokumentaci RESTful API v Ruby On Rails[2] aplikacích. Nástroj se poskytuje jako externí knihovna (tzv. `gem`), a proto se snadno začleňuje do kódu aplikace. Namísto tradičního použití komentářů `Apipie-rails` umožňuje popisovat kód pomocí kódu, což ovšem přináší výhody zmíněné v sekci 2.2.2. Zejména:

- Není třeba se učit další syntaxi (předpokládá se znalost Ruby), ale pouze malý DSL.
- Možnost opětovného použití dokumentace pro jiné účely (např. validace).
- Jednodušší rozšiřování a údržba (není nutnost zpracovávat komentářové řetězce).
- Možnost opětovného využití dalších zdrojů pro účely dokumentace.

Dokumentace může mít několik verzi a je k dispozici jako HTML stránky v rámci aplikace pod uživatelem zvolenou cestou (např. `https://fqdn/apipie`). Nástroj je agnostický ke značkovacím jazykům a dokonce poskytuje rozhraní pro opětovné použití dokumentace v JSON. Jednoduchý příklad toho, jak vypadá `Apipie-rails` DSL v kódu, je vidět na výpisu 3.1, jehož rozšíření je vidět na výpisu 3.2. Na výpisu 3.1 je definice třídy `UsersController`, která je zároveň popsaná pomocí `resource_description`, s jedinou metodou `create`, která se vyvolává, když na server přijde požadavek na vytvoření nového uživatele. Tato metoda je zadokumentována pomocí `Apipie-rails`, jehož DSL je vidět před samotnou definicí metody. Dále na výpisu 3.2 je vidět rozšíření modulu `OauthConcern` o modul `Apipie::DSL::Concern`, který přidá metodu `update_api`, pomocí níž lze zaktualizovat dokumentaci přijímaných parametrů metodou `create`.

```
class UsersController < ApplicationController
  resource_description do
    formats [:json]
    api_versions 'v2'
```

```
  end

  api :POST, '/users' 'Create user'
  description 'Create user with specifed params'
  param :user, Hash, desc: 'User' do
    param :name, String, desc: 'First name of the user'
    param :surname, String, desc: 'Surname name of the user'
    param :age, Integer, desc: 'Age of the user'
  end
  def create
    # Action code
  end
end
```

Výpis 3.1: Apipie-rails příklad DSL.

```
module Concerns
  module OauthConcern
    extend Apipie::DSL::Concern

    update_api(:create, :update) do
      param :user, Hash do
        param :oauth, String,
          desc: 'oauth param'
      end
    end
  end
end

UsersController.send(:include, Concerns::OauthConcern)
```

Výpis 3.2: Apipie-rails příklad rozšíření DSL.

Dále následuje pro lepší představu krátké shrnutí syntaxe API a DSL.

Apipie-rails DSL

`Apipie-rails` umožňuje zdokumentovat zdroje (tzv. `resources`), koncové body (`endpoints`), parametry požadavků a odpovědí serveru.

Popis zdrojů

Popis lze provádět na úrovni řadiče, a to předáváním bloku kódu s popisovacími metodami, jako jsou `name`, `full_description` a `param`, do metody `resource_description`. Dědičnost je podporována, takže lze specifikovat společné parametry pro skupinu řadičů v jejich rodičovské třídě.

Příklad DSL pro popis zdrojů je vidět v přílohách na výpisu A.1.

Popis koncových bodů

Popis koncových bodů se provádí voláním určitých popisovacích metod před samotnou definicí příslušné metody, která se vyvolává při obsluze požadavku na serveru.

Příklad DSL pro popis koncových bodů je vidět v přílohách na výpisu A.2.

Popis parametrů

Popis parametrů požadavku se provádí voláním metody `param` s příslušnými parametry (např. název, popis či validátor). Pro popis vnořených parametrů lze použít `Hash` validátor ve spojení s blokem kódu popisujícím tyto parametry.

Příklad DSL pro popis parametrů je vidět na výpisu 3.3.

```
param :user, Hash, :desc => "User info" do
  param :username, String, :desc => "Username for login",
        :required => true
  param :password, String, :desc => "Password for login",
        :required => true
  param :membership, ["standard","premium"], :desc => "User membership"
  param :admin_override, String, :desc => "Not shown in documentation",
        :show => false
  param :ip_address, String, :desc => "IP address", :required => true,
        :missing_message => lambda { I18n.t("ip_address.required") }
end
def create
  #...
end
```

Výpis 3.3: Příklad DSL pro popis parametrů (převzato z [9]).

Stejné parametry se často vyskytují ve více akcích. Obvykle většina parametrů pro akce `create` a `update` je sdílena mezi nimi. Tyto parametry lze extrahovat pomocí klíčových slov `def_param_group` a `param_group`. Definice znovupoužitelné skupiny parametrů je vzata v úvahu v rámci řadiče. Pokud je skupina definována v jiném řadiči, musí se odkazovat zadáním druhého argumentu popisné metody `param_group`.

Příklad DSL pro definici a použití skupiny parametrů je vidět v přílohách na výpisu A.3.

Popis odpovědí serveru

Odpověď serveru lze zadokumentovat přidáním příkazu `returns` k popisu metody čili koncového bodu. To je obzvláště užitečné, když se `Apipie` používá k automatickému generování strojově čitelných definic `Swagger`.

Příklad DSL pro popis odpovědí je vidět v přílohách na výpisu A.4.

Validace

Navíc nástroj je dostatečně konfigurovatelný a mimo jiné umožňuje nastavit dle vlastního uvážení cestu, ze které bude dokumentace dostupná, lokalizaci celé dokumentace či dokonce

autentizaci uživatele. Stojí za zmínku také možnost automatické validace parametrů příchozích na server požadavků. Pokud validace parametrů je zapnutá (výchozí stav), tak se provádí automaticky před voláním příslušné metody řadiče. Je-li hodnota nesprávná, vyvolá se výjimka, kterou lze zachytit a následně zpracovat. Výjimka obsahuje popis očekávaných hodnot parametrů. Každý parametr pak musí mít přidružený validátor (angl. *validator*). Existují některé základní validátory, ale aby uživatelé dosáhli složitějších výsledků, mohou poskytnout i své vlastní.

Základní validátory:

- **TypeValidator:** Kontroluje typ parametru. Z důvodu jednoduchosti je podporováno pouze několik výchozích typů. Lze přidat vlastní validátor.
- **RegexpValidator:** Kontroluje hodnotu parametru proti danému regulárnímu výrazu.
- **EnumValidator:** Kontroluje, zda je hodnota parametru v daném výčtu zahrnutá.
- **ProcValidator:** Kontroluje hodnotu parametru předaným kusem kódu či lambda funkcí.
- **HashValidator:** Při zadávání bloku s popisem vnořených hodnot kontroluje tyto parametry objektu asociativního pole.
- **ArrayValidator:** Kontroluje, zda je parametr pole.
- **DecimalValidator:** Kontroluje, zda je parametr desetinné číslo.
- **NestedValidator:** Pomocí bloku s popisem vnořených hodnot lze popisovat vnořené parametry.

Příklady HTML stránek s výslednou dokumentací jsou v přílohách na obrázcích B.1 a B.2.

`Apipie-rails` se vydává pod licencí MIT.

3.2 Specifikace požadavků

Požadavky na výsledný nástroj lze rozdělit do tří skupin:

- Požadavky na zpracování vstupu
- Požadavky na strukturu
- Požadavky na výstup

Požadavky na zpracování vstupu

Požadavky na zpracování vstupu spočívají v poskytování možnosti zdokumentovat veškeré prvky libovolného DSL napsaného v Ruby stejným nebo podobným způsobem, jak to dělá `Apipie-rails` pro dokumentaci API. K těmto prvkům patří *moduly*, *třídy*, *metody*, *parametry* těchto metod a jejich *návratové hodnoty*.

Požadavky na strukturu

Požadavky na strukturu v sobě zahrnují definici API nebo DSL čili samotný způsob a prostředky pro dokumentování. `Apipie-rails` poskytuje spoustu dokumentačních metod, a to rozšířením uživatelských tříd a modulů, což může vést ke kolizím mezi poskytnutými a uživatelskými metodami. Jeden z nejočividnějších požadavků je zredukovat počet těchto kolizí a přitom nechat jednoduchost dokumentačního procesu. Dalším požadavkem je poskytnutí možnosti vyhnout se opakované definici popisu metod nebo parametrů se stejnými vlastnostmi čili umožnit přístup podle principu **DRY**[1]. Nástroj by měl umožňovat jednoduchý způsob rozšíření dokumentace a podporovat možnost konfigurace podobně jako `Apipie-rails`.

Požadavky na výstup

Požadavky na výstup definují formáty a způsoby zobrazení výsledné dokumentace. `Apipie-rails` začleňuje výslednou dokumentaci do Ruby On Rails aplikace a zpřístupňuje ji přes speciální URL uživatelovi. Kromě toho `Apipie-rails` umožňuje staticky generovat dokumentaci z příkazové řádky, a to ve formátech JSON a HTML s podporou lokalizace. Hlavním požadavkem tady je možnost generovat jak HTML, tak i JSON minimálně staticky. Výhodou by taky bylo zpřístupňovat dokumentaci uvnitř programu za jeho běhu, podpora lokalizace a možnost používat uživatelské šablony pro generování HTML stránek.

3.3 Návrh nového nástroje

Cílem této práce je navrhnout a realizovat modifikaci stávajícího nebo zcela nový dokumentační nástroj. Při zkoumání `Apipie-rails` jsem zjistil, že modifikace by nebyla nejlepším přístupem, neboť tento nástroj byl původně zaměřen na použití v kontextu Ruby On Rails aplikací. Nástroj používá vnitřní datové struktury frameworku Ruby On Rails jako jsou `ActiveSupport::OrderedHash` a `ActiveSupport::HashWithIndifferentAccess`, používá `Rails::Railtie` pro úpravu inicializačního procesu a obecně návrh orientuje své použití v rámci MVC[2] architektury. Modifikace tohoto nástroje, která by umožnila použití ve všech aplikacích napsaných v Ruby, by byla zbytečně velká a implementace by vedla spíše k přepsání celého kódu nástroje `Apipie-rails`. Proto jsem se rozhodl vytvořit nový nástroj `Apipie-dsl` pro obecnější použití.

Apipie-dsl DSL

Aby splnil požadavky na zpracování vstupu, `Apipie-dsl` musí umožňovat dokumentaci *modulů*, *tříd*, *metod*, jejich *parametrů* a *návratových hodnot*. Dále následuje stručný výčet na to navržených metod. Úplný přehled je popsán v příručce[3].

[1] Don't repeat yourself (česky „neopakuj se") – Každý kus znalostí musí mít jedno, jednoznačné a autoritativní zastoupení v systému.[10]

[2] Model-view-controller je softwarová architektura, která rozděluje datový model aplikace, uživatelské rozhraní a řídicí logiku do tří nezávislých komponent tak, že modifikace některé z nich má jen minimální vliv na ostatní.

[3] https://github.com/ofedoren/apipie-dsl/blob/master/README.md

Společné dokumentační metody

- **short (short_description):** Stručný popis třídy (je zobrazen jak na seznamu tříd, tak i na detailech tříd).
- **desc (description, full_description):** Úplný popis třídy (zobrazen pouze v detailech třídy).
- **dsl_version (dsl_versions):** Jaké verze existují pro danou třídu.
- **meta:** Vlastní metadata.
- **deprecated:** Booleovská hodnota označující, zda je prostředek označen jako zastaralý. (Výchozí hodnota je `false`)
- **show:** Booleovská hodnota označující, zda je třeba zobrazovat tyto informace ve výsledné dokumentaci.

Moduly a třídy

- **app_info:** Nastavuje popis informací o aplikaci.
- **property:** Slouží pro popis vlastností instance dané třídy.
- **define_param_group:** Definuje skupinu parametrů, na kterou se pak lze odkazovat při popisu metod nebo vnořených parametrů.

Metody

- **method:** Obecný popis metody (jméno a popis k čemu daná metoda slouží).
- **raises:** Popis možných výjimek.
- **returns:** Popis návratové hodnoty.
- **see:** Poskytuje odkaz na jinou metodu. Musí to být řetězec ve tvaru `název_třídy#jméno_metody`.
- **param_group:** Odkaz na dříve definovanou skupinu parametrů.

Parametry

- **param:** Popis parametru metody (jméno, validátor, slovní popis)
- **required:** Podobně `param`, ale pro povinné parametry.
- **optional:** Podobně `param`, ale pro volitelné parametry (lze ukázat výchozí hodnotu).
- **keyword:** Podobně `param`, ale pro pojmenované parametry.
- **block:** Podobně `param`, ale pro parametry typu `block`, `proc`.

Struktura

Protože tento nástroj je zaměřen na obecné použití (nikoliv pouze v Ruby On Rails aplikacích), je potřeba vyhnout se použití zvláštností zmíněného frameworku, ale přitom zachovat určitou kompatibilitu s `Apipie-rails`. Splnění požadavků na strukturu lze dosáhnout splněním následujících bodů:

- **Konfigurace** – Aplikace je konfigurovatelná voláním metody `ApipieDSL.configure` a předáním bloku kódu, ve kterém se vyvolávají konfigurační metody (např. pomocí `validate` lze vybrat způsob validace parametrů metod).
- **Rozšíření** – Poskytnutí dokumentačních metod uživateli musí být intuitivní a nesmí vést ke kolizím mezi nimi a metodami uživatele. Toho lze dosáhnout rozšířením uživatelských modulů nebo tříd pomocí příkazu `extend` vlastními třídami (např. `ApipieDSL::Class` nebo `ApipieDSL::Extension`), které přidají pouze pár (ideálně jednu) obalovacích metod se specifickým jménem (např. `apipie` nebo `apipie_update`), do kterých se předá blok kódu s dokumentačními metodami.
- **DRY** – Například některé metody mohou mít stejné parametry, což může vést ke zbytečnému popisu každého parametru několikrát. Aby se tomu dalo vyhnout, je vhodné umožnit předčasnou definici skupiny opakujících se parametrů metodou `define_param_group` a pak se na ni odkazovat metodou `param_group`.

Výstup aplikace

Pro splnění požadavků na výstup aplikace umožní získat výslednou dokumentaci několika způsoby:

- příkaz `ApipieDSL.docs` připraví dokumentaci pro použití uvnitř hlavní aplikace.
- příkaz `rake apipie_dsl:static` vygeneruje výslednou dokumentaci ve formátu HTML do složky uvedené uživatelem (výchozí složka je `docs/dsl`).
- příkaz `rake apipie_dsl:static_json` vygeneruje výslednou dokumentaci ve formátu JSON do složky uvedené uživatelem (výchozí složka je `docs/dsl`).

Kapitola 4

Implementace

V této kapitole jsou popsány hlavní části implementace navrženého nástroje a způsoby řešení problémů, které se vyskytly při použití knihovny v Ruby a v Ruby on Rails aplikacích.

4.1 Struktura nástroje

Nástroj je implementován jako samostatná knihovna, tzv. `gem`, který je veřejně dostupný ke stažení přes *RubyGems.org*[1].

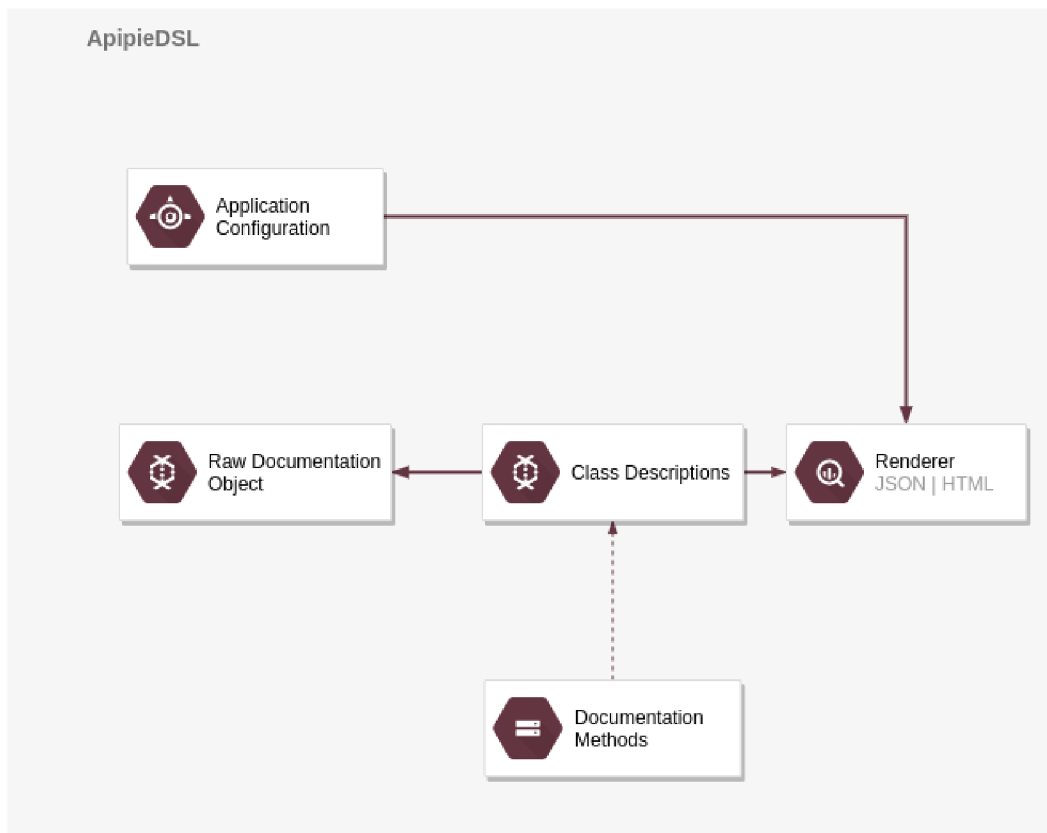
Hlavními prvky, ze kterých se skládá nástroj, jsou:

- **ApipieDSL::Configuration** – Třída, jejíž instance reprezentuje současnou konfiguraci celé aplikace.
- **ApipieDSL.class_descriptions** – Datová struktura typu `Hash`, ve které jsou uloženy popisy tříd, modulů a jejich metod.
- **ApipieDSL::Class** – Třída poskytující dokumentační metody. Úplný výčet všech dokumentačních metod je popsán v příručce[2].
- **ApipieDSL.docs** – Datová struktura typu `Hash`, ve které se nachází současná dokumentace.
- **Renderer** – Prvek v podobě tzv. `rake` úloh umožňující generování dokumentace v JSON nebo HTML podobě pomocí určitého příkazu z příkazové řádky.

Vizuálně struktura nástroje je zobrazena na schématu 4.1. Toto schéma zobrazuje závislosti mezi hlavními prvky. Čarkovaná šipka znamená přímou závislost mezi popisy tříd a dokumentačními metodami, tj. popisy jsou vytvářeny dokumentačními metodami. Obyčejná šipka znamená nepřímou závislost, např. popisy tříd se jenom používají pro získání výsledné dokumentace, a to buď pomocí tzv. vykreslovače, nebo jako pouhý objekt v kódu.

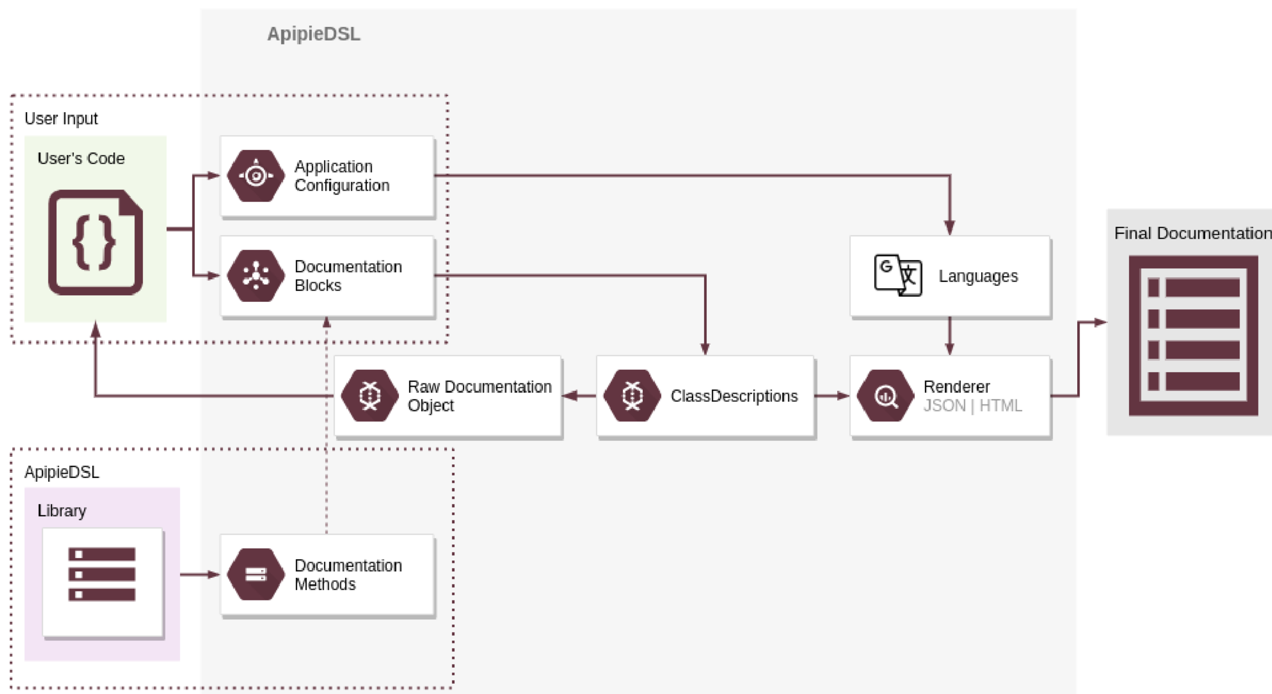
[1]https://rubygems.org/gems/apipie-dsl

[2]https://github.com/ofedoren/apipie-dsl/blob/master/README.md



Obrázek 4.1: Přehled struktury nástroje Apipie-dsl.

Stručný přehled struktury libovolné aplikace s použitím nástroje je zobrazen na schématu 4.2. Toto schéma zobrazuje závislosti mezi hlavními prvky nástroje stejným způsobem jako schéma 4.1, ale oproti předchozímu zobrazuje taky integraci nástroje do uživatelské aplikace. Po zapojení knihovny do uživatelské aplikace, uživatel má k dispozici ve svém kódu dva veřejně dostupné prvky — konfigurační třídu čili prvek *Application Configuration* a dokumentační metody čili prvek *Documentation Methods*. Nástroj získává od uživatele potřebnou konfiguraci pomocí prvního prvku a dokumentaci svého DSL přes tzv. dokumentační bloky kódu, ve kterých jsou použity dokumentační metody. Tato informace se pak použije pro generování dokumentace způsobem zvoleným uživatelem.



Obrázek 4.2: Přehled struktury aplikace s použitím nástroje Apipie-dsl.

Nástroj docela hodně využívá jednu z hlavních zvláštností Ruby — bloky kódu. Hlavním důvodem je snadnost návrhu vlastního DSL a to, že kód napsaný v podobném bloku se vykonává v jiném kontextu čili má jiný objekt, který odpovídá daným zprávám, což umožňuje vykonávat kód bez pro uživatele nepříjemných důsledků jako je kolize mezi různým kódem. Příklady zmíněného přístupu jsou popsány dále.

Konfigurace

Po instalaci uživatel musí zpočátku nakonfigurovat knihovnu dle svých potřeb. Toto lze provést pomocí speciální metody `ApipieDSL.configure` (viz příklad 4.1), která jako argument přijímá blok kódu, v němž se používají konfigurační metody poskytované třídou `ApipieDSL::Configuration`. Úplný seznam je popsán v příručce[3].

```
ApipieDSL.configure do |config|
  config.app_name = 'New App'
  config.dsl_classes_matchers = [
    "lib/my_app/**/*.rb",
  ]
  config.validate = true
end
```

Výpis 4.1: Příklad konfiguraci Apipie-dsl.

[3] https://github.com/ofedoren/apipie-dsl/blob/master/README.md

Dokumentace vlastního DSL

Kvůli požadavku na redukci kolizí mezi uživatelským kódem a kódem poskytovaným knihovnou jsem se rozhodl skrýt většinu metod, které mají docela generická jména (např. `name`), od uživatele a poskytovat dokumentační metody pouze se specifickým jménem jako `apipie` a `apipie_update`. Takže veškerý kód, který se vztahuje pouze k popisu, lze umístit jen jako argument výše zmíněných metod (viz příklad 4.2). Navíc uživatel nemusí zadávat jméno metody, kterou popisuje, ani jméno třídy, ve které se tato metoda byla definována, protože tahle jména jsou převzata automaticky, když interpret Ruby prochází definicí třídy. Stojí za zmínku, že se v tomto okamžiku mění definice popisované metody, aby obsahovala validaci předávaných argumentů, pokud samozřejmě tohle bylo nastaveno v konfiguraci. Toto se uskutečňuje díky vestavěné metodě `method_added`, která je v každé nativní třídě Ruby. Tato metoda se vyvolává jako zpětné volání, kdykoli je do příjemce přidána instanční metoda.

```ruby
class Example
  extend ApipieDSL::Class

  apipie :class, 'Example' do
    desc 'A very simple DSL class which demonstrates ApipieDSL
        capability'
    property :methods, 'Returns names of all defined methods.'
  end

  apipie :method, 'Creates HTML div tag for every image URI passed' do
    keyword :id, String, 'If passed, creates wrapper div with given id'
    keyword :class, String, 'If passed, apply given class for each div
        created'
    keyword :img_uris, Array, of: String, desc: 'URIs of images'
    returns String, desc: 'Returns HTML string with divs'
  end
  def img_divs(id: '', class: '', img_uris: [])
    # Code
  end
end
```

Výpis 4.2: Příklad popisu vlastního DSL pomocí Apipie-dsl.

Každá takhle popsaná metoda se automaticky uloží do `ApipieDSL.class_descriptions` jako objekt třídy `ApipieDSL::MethodDescription`.

Aktualizace dokumentace

Nástroj má prostředky pro rozšiřování neboli aktualizaci dokumentace. Jejich použití je lepší ukázat na příkladě mikroservisní architektury, kde samotná aplikace vystupuje v roli jádra a je pak rozšiřována pomocí několika zásuvných modulů přidávajících další funkcionalitu.

Představme si jednoduchou aplikaci, která umožňuje spouštění skriptů na počítači a poskytuje DSL, pomocí kterého lze modifikovat samotný proces jejich spouštění. Jednou z metod tohoto DSL je `run_script`. V základní verzi aplikace tato metoda přijímá pouze jeden

argument — `script` — soubor obsahující skript. Tato metoda je zadokumentována následujícím způsobem (viz výpis 4.3, pro výsledek viz obrázek 4.3).

```
class ExecutionDSL
  extend ApipieDSL::Class

  apipie :method, 'Runs given script on the host' do
    required :script, String, 'File containing the script, can be a
       string with the script or a path'
    returns one_of: [true, false], desc: 'On success returns true, false
        otherwise'
  end
  def run_script(script)
    # ...
  end
end
```

Výpis 4.3: Příklad dokumentace metody run_script.

All

My DSL Docs 1.0 / Execution::ExecutionDSL / run_script

run_script(script)

Runs given script on the host

Params

Param Name	Description
script Required	File containing the script, can be a string with the script or a path **Validations:** • Must be a String

Returns

On success returns true, false otherwise

Object	Details
one_of	[true, false]

Obrázek 4.3: Výsledek dokumentace metody run_script.

Teď si představme, že existuje plugin `Remote Execution` pro tuto aplikaci přidávající možnost spouštění skriptů na vzdáleném počítači. Po instalaci tento plugin přepíše metodu `run_script` a přidá zmíněnou funkcionalitu. Aby uživatel nemusel přepisovat své definice procesů, které používají danou metodu, plugin zachová zpětnou kompatibilitu a přidá volitelný parametr `ssh_key`. Navíc je potřeba dát uživateli vědět o změnách, a proto je potřeba taky aktualizovat dokumentaci této metody. Díky mému nástroji není nutno přepisovat celou dokumentaci, stačí přidat pouze související změny pomocí `apipie_update`. Aktualizace je zobrazena na výpisu 4.4 a výsledek na obrázku 4.4.

```
module RemoteExtension
  extend ApipieDSL::Extension

  apipie_update do
```

```
    desc 'With Remote Execution plugin installed requires path to SSH
      public key'
    optional :ssh_key, String, desc: 'Path to SSH key', default: ''
  end
  def run_script(script, ssh_key = '')
    # ...
  end
end
```

Výpis 4.4: Příklad aktualizace dokumentace metody run_script.

All

My DSL Docs 1.0 / Execution::ExecutionDSL / run_script

run_script(script, ssh_key = "")

Runs given script on the host

With Remote Execution plugin installed requires path to SSH public key

Params

Param Name	Description
script Required	File containing the script, can be a string with the script or a path **Validations:** • Must be a String
ssh_key Optional	Path to SSH key **Default value:** • "" **Validations:** • Must be a String

Returns

On success returns true, false otherwise

Object	Details
one_of	[true, false]

Obrázek 4.4: Výsledek aktualizace dokumentace metody run_script.

Sice je možné získat popisy všech tříd a metod voláním ApipieDSL.class_descriptions pro své účely, ale bez znalostí vnitřních tříd knihovny, je toto nepraktické. Pro získání reprezentace ve formátu JSON s více užitečnější dokumentací existuje metoda ApipieDSL.docs, jež je popsaná dále.

Získávání dokumentace

Jeden ze způsobů, jak získat dokumentaci je volání ApipieDSL.docs. Tato metoda přijímá maximálně 5 argumentů a v závislosti na předaných parametrech vrací dokumentaci buď celou, nebo jediné třídy, metody v podobě asociativního pole. Pro JSON podobu stačí zavolat to_json na dokumentačním objektu.

Uživatel má možnost získat dokumentaci kdekoli ve svém kódu pomocí volání této metody. To se může hodit za zvláštních podmínek, ale pro vývojáře jako uživatele jsou přehlednější HTML stránky. Proto je lepší generovat dokumentaci pomocí příkazu rake apipie_dsl:static, aby bylo možné používat takový příkaz, uživatel zpočátku musí přidat sadu příkazů z knihovny do souboru svých příkazů, do tzv. Rakefile. Jeden ze způsobů je uveden na výpisu 4.5.

```
require 'apipie-dsl'

# Configuration is required!
ApipieDSL.configure do |config|
  config.app_name = 'New App'
  config.dsl_classes_matchers = [
    "lib/my_app/**/*.rb",
  ]
  config.validate = true
end
spec = Gem::Specification.find_by_name('apipie-dsl')
rakefile = "#{spec.gem_dir}/lib/apipie_dsl/Rakefile"
load(rakefile)
```

Výpis 4.5: Přidání příkazů do Rakefile.

4.2 Řešení implementačních problémů

Během implementace jsem narazil na několik problémů souvisejících s podporou jak obyčejných aplikací v Ruby, tak i aplikací napsaných v Ruby on Rails. Moje knihovna se snaží být použitelná v obojím případě — využívat mocné vlastnosti frameworku, ale moc se nepřipoutávat k onému, aby se zredukoval počet závislostí.

Ruby

Pro jednoduchost generování HTML dokumentace jsem se rozhodl využít HTML šablony s použitím vestavěného Ruby[4]. Takový přístup vyžaduje implementaci dalšího prvku — tzv.

[4]Vestavěný Ruby (angl. *Embedded Ruby*) je šablonový systém, který vkládá Ruby do textového dokumentu. Často se používá ke vložení Ruby kódu do HTML dokumentu, což je podobné ASP, JSP a PHP

vykreslovače (angl. *renderer*). Protože původně aplikace byla zaměřena spíše na použití v Ruby on Rails aplikacích, rozhodl jsem se využít vykreslovač ze zmíněného frameworku. Díky tomu, že se vývojáři Ruby on Rails rozhodli rozdělit framework na dílčí součásti, dá se požadovaný vykreslovač připojit do libovolné aplikace mnohem jednodušeji, a to pouhým zařazením knihovny `actionview` mezi závislosti aplikace bez nutnosti instalace celého frameworku. Moje knihovna nemá balík `actionview` jako explicitní závislost kvůli redukci výchozích závislostí při instalaci. Aby bylo možné vygenerovat HTML stránky s dokumentací pomocí mého nástroje, uživatel si musí explicitně nainstalovat knihovnu `actionview`. Můj nástroj si automaticky načte potřebné třídy tohoto balíku, takže žádná další konfigurace není potřeba. Pokud tato závislost nebyla nainstalována, uživateli se zobrazí chyba. Mám v plánu tohle vyřešit v dalších verzích nástroje.

Ruby on Rails

V Rails aplikacích při použití nestandardní třídy jako validátoru se občas může vyskytnout chyba kruhové závislosti při automatickém načítání konstanty. Jedním z důvodů může být pořadí, ve kterém jsou soubory zpracovány. Pro eliminaci tohoto problému jsem přidal tzv. `LazyValidator`. Toto umožňuje pokračovat v dokumentaci metod, aniž by docházelo ke kruhové závislosti, protože k načítání konstanty dochází až při generování dokumentace, což se děje po kompletní inicializaci aplikace.

Během aplikace výsledku práce jsem získal zpětnou odezvu s požadavkem na zrychlení renderování dokumentace v Rails aplikacích. Tento požadavek vznikl hlavně kvůli velkému počtu dokumentovaných metod a nutnosti mít předgenerovanou verzi dokumentace, kterou lze použít jako cache. Aby bylo možné jednoduše vygenerovat tuto cache, rozhodl jsem přidat další příkaz `apipie_dsl:cache`, který lze spouštět z příkazové řádky. Tento příkaz se chová podobně příkazu `apipie_dsl:static`, ale je speciálně zaměřen na použití v rámci serverové aplikace pro rychlejší načtení souborů s dokumentací na straně serveru.

Byl taky nalezen potenciální problém, kdy volitelná knihovna obsahující jeden ze standardních značkovacích jazyků nebyla dostupná na systému používajícím můj nástroj. Když jsem přidával možnost používat několik vestavěných značkovacích jazyku, předpokládal jsem, že uživatel bude mít minimálně knihovnu `rdoc`, což není zaručeno. Tento problém mohl způsobit pád celé aplikace již v momentu inicializace `Apipie-dsl`. Pro zabránění tomuto problému jsem implementoval tzv. líné načítání (angl. *Lazy Loading*) značkovacího jazyka. Ke skutečnému vyhodnocení používaného jazyka dochází až v momentu skutečné potřeby, a to při generování výsledné dokumentace. Toto umožní uživateli definovat kdekoli ve svém kódu vlastní značkovací jazyk, který se musí pak použít.

---

a dalším skriptovacím jazykům. Tento systém kombinuje kód Ruby a prostý text, aby poskytoval řízení toku a nahrazení proměnných, což usnadňuje údržbu.[4]

Kapitola 5

Aplikace nástroje

Aplikace mého nástroje čili knihovny byla provedena integrací do projektu s otevřeným zdrojovým kódem Foreman. Primárně je moje knihovna rozšiřována a upravována, aby byla integrace s projektem Foreman co nejlepší.

5.1 Foreman

Foreman je nástroj pro správu kompletního životního cyklu fyzických a virtuálních serverů[1]. Tento nástroj je zamýšlen jako webová aplikace napsaná v Ruby on Rails, kterou lze spouštět na jednom systému pro správu celých sítí serverů. Samotný Foreman lze rozšířovat pomocí velké sady zásuvných modulů (angl. *plugin*), čímž se přidávají různé funkce navíc.

5.1.1 Šablony

Jedním z hlavních prvků aplikace Foreman jsou šablony. Šablona je sada instrukcí pro popis nějakého konkrétního procesu. Procesem v daném případě může být vytvoření nového serveru, včetně jeho konfigurace, popis očekávaných zpráv od serveru a popis úloh na něm prováděných.

Šablony jsou soubory, v nichž se používá vestavěný Ruby, aby byl popis zmíněných procesů co nejobecnějším pro účely znovupoužití. Příklad takové šablony pro získání informací o stavech serverů je vidět na výpisu 5.1. Každá šablona může obsahovat libovolné množství metod, ale nemusí se skládat pouze z nich.

Kvůli tomu, že Foreman poskytuje docela velké množství těchto metod, je potřeba mít nějaký seznam nebo příručku, kde budou popsány všechny metody dohromady s příklady použití. Po dlouhou dobu Foreman měl pouhý seznam metod bez popisů a příkladů, ale díky mojí práci má projekt možnost mít přehlednou dokumentaci obsahující potřebné informace.

```
<%- report_headers 'Name', 'Global' -%>
<%- load_hosts(search: input('hosts'), includes: :host_statuses).
    each_record do |host| -%>
<%- report_row({
        'Name': host.name,
        'Global': host.global_status
      }.merge(all_host_statuses_hash(host))) -%>
<%- end -%>
<%= report_render -%>
```

Výpis 5.1: Příklad šablony.

Foreman poskytuje několik vestavěných šablon pro různé účely, ale uživatelé mají možnost je upravovat a psát své vlastní šablony, pokud je potřeba. Uživatelem může, ale nemusí být Ruby programátor či programátor vůbec. Proto je potřeba mít nějakou dokumentaci o metodách, které uživatel může používat.

Po integraci mé knihovny do projektu, výše uvedené metody — *report_headers* a *report_row* — na výpisu 5.1 lze popsat způsobem uvedeným na výpisu 5.2. Výsledek je pak vidět na obrázku 5.1.

```
apipie :method, 'Register minimal headers for the report' do
  desc "This is useful in case of possible empty reports. **report_row**
      macro will automatically update
    headers if needed."
  list :headers, desc: 'List of headers'
  returns Array, desc: 'Minimal registered headers'
  example "<%- report_headers 'id', 'name' -%>"
end
def report_headers(*headers)
  # code
end

apipie :method, 'Register a row of data for the report' do
  desc "For every record that should be part of the report, **report_row
     ** macro needs to be called.
    The only argument it accepts is a record definition. This is
      typically called in some **each** loop. Calling
    this at least once is important so we know what columns are to be
      rendered in this report.
    Calling this macro adds a record to the rendering queue."
  required :row_data, Hash, desc: 'Data in form of hash, keys are column
      names, values are values for this record'
  returns Array, desc: 'Currently registered report data'
  example "report_row(:name => 'host1.example.com', :ip =>
     '192.168.0.2')"
  example "<%- load_hosts.each_record do |host|\n report_row(:name =>
     host.name, :ip => host.ip)\nend -%>"
end
def report_row(row_data)
  # code
end
```

Výpis 5.2: Popis metod použitých v šabloně.

Za zmínku stojí i metoda metoda `load_hosts` ze šablony 5.1, která jako taková ve zdrojovém kódu programu neexistuje, ale je dynamicky vytvářena za běhu programu pomocí

vestavěné v Ruby metodě `define_method` přijímající jako argumenty jméno a blok kódu s definicí nové metody. Takovým způsobem se ale vytváří několik metod najednou. Moje knihovna s tím počítá a popis takového kódu je zobrazen na výpisu 5.3. Na tomto výpisu konstanta `LOADERS`, jež je pak iterována, mimo jiné obsahuje jména nově vytvářených metod, která jsou pak použitá v `define_method`. Tato jména se automaticky převezmou mojí knihovnou a pro každou z nově vytvářených metod se zavolá blok předaný metodě `apipie`. Kvůli tomu, že předávaný blok je kódem, programátor může napsat obecnou definici pro všechny metody a zároveň ji upravit tak, aby obsahovala různé proměnné v závislosti na určité generované metodě, což je výhodou oproti dokumentačním komentářům.

```
LOADERS.each do |name, model, permission|
  apipie :method, "Loads #{model} objects" do
    desc "This macro returns a collection of #{model.to_s.pluralize}
        matching search criteria.
      The collection is limited to what objects the current user is
          authorized to view by #{permission} permission. Also it's
          loaded in bulk
      of 1 000 records."
    param_group :load_resource_keywords
    returns array_of: model, desc: 'The collection that can be iterated
        over using each_record'
    example "<% #{name}.each_record do |#{model.to_s.downcase}| %>
<%= #{model.to_s.downcase}.id %>, <%= #{model.to_s.downcase}.#{model ==
    User ? 'login' : 'name'} %>
<% end %>", desc: "Prints #{model.to_s.downcase} id and #{model == User
    ? 'login' : 'name'} of each #{model.to_s.downcase}"
    example_for :load_hosts, "<% load_hosts(search: 'domain = example.
        com', includes: [ :interfaces ]).each_record do |host| %>
  <%= host.name %>, <%= host.interfaces.map { |i| i.mac }.join(', ') %>
<% end %>", desc: 'Prints host name and MACs of each host'
    example_for :load_users, "<% load_users(search: 'location = Europe',
         includes: :auth_source).each_record do |user| %>
  <%= user.login %>, <%= user.last_login_on %>, <%=
      user_auth_source_name(user) %>
<% end %>", desc: "Prints users in Europe, their login, when a user was
    logged on for the last time and the authentication source"
  end
  define_method name do |search: '', includes: nil, preload: nil, joins:
       nil, select: nil, batch: 1_000, limit: nil|
    load_resource(klass: model, search: search, permission: permission,
        includes: includes, preload: preload, joins: joins, select:
        select, batch: batch, limit: limit)
  end
end
```

Výpis 5.3: Popis dynamicky vytvořených metod.

All Basic Ruby Methods Provisioning Reports Jobs Additional Help

Foreman v1 / Report

Report

Macros specific for report rendering

Method

report_render(format: nil) >>>

Render a report for all rows defined

report_headers(*headers) >>>

Register minimal headers for the report

This is useful in case of possible empty reports. **report_row** macro will automatically update headers if needed.

Examples

```
<%- report_headers 'id', 'name' -%>
```

Params

Param Name	Description
headers Optional	List of headers **Default value:** • empty list **Validations:** • Must be a list of values

Returns

Minimal registered headers

Array **Details**

Obrázek 5.1: Výsledek popisu metod použitých v šabloně.

5.1.2 Další možnosti integrace

Jednou z dalších možností integrace do projektu Foreman je použití knihovny přímo v grafickém uživatelském rozhraní aplikace. Idea spočívá v poskytování nápovědy uživateli, zatímco on píše šablonu, a to pro validaci v reálném čase předávaných parametrů do tzv. maker, speciálných metod používaných v šablonách. Navíc by bylo dobré přidat syntaktickou kontrolu a automatické doplňování podobným způsobem, jak to dělají moderní textové editory a vývojová prostředí (např. Atom nebo Microsoft Visual Studio). Kvůli tomu, že dokumentace je součástí aplikace, uživatel bude schopen dostat nejaktuálnější dokumentaci ohledně verze použité aplikace, a to bez nutnosti se dívat na celou příručku.

Návrh takové integrace není cílem této práci, proto detaily návrhu nejsou její součástí.

Kapitola 6

Závěr

Cílem mojí práce bylo navrhnout a realizovat modifikaci stávajícího nebo úplně nový dokumentační nástroj, který by poskytoval způsob k dokumentaci dynamicky rozšiřitelných DSL v programovacím jazyce Ruby.

Na začátku práce jsem probral pojem DSL a možnosti dokumentací kódu dvěma různými způsoby — pomocí komentářů nebo kódu. Byly vysvětleny rozdíly, výhody a nevýhody těchto přístupů. Tato práce se zaměřila na druhý způsob neboli dokumentace pomocí kódu.

Dále byla provedena analýza stávajícího nástroje, pomocí níž jsem zjistil, že lepší by bylo realizovat nový dokumentační nástroj, což je výsledkem mojí práce. Nástroj je implementován jako knihovna napsaná v jazyce Ruby, kterou lze použít v jakékoli Ruby aplikaci. Samozřejmě použití v rámci Ruby on Rails aplikace je výhodnější a jednodušší, ale knihovnu je možné využít i v aplikacích napsaných bez dalších frameworků. Knihovna poskytuje prostředky jak pro dokumentaci základního DSL, tak i pro jeho rozšíření. Dokumentaci lze získat buď přímo v kódu, nebo vygenerovat v JSON a HTML podobě.

Aplikace výsledného nástroje byla provedena v rámci projektu Foreman pro řešení problému souvisejícího s nedostatkem dokumentace maker používaných při psaní šablon. Knihovna se velice dobře zaintegrovala do projektu a teď je jednou ze součástí, kterou používají i další vývojáři. Navíc se zvažuje jiná možnost použití než výchozí (viz sekci 5.1.2), zejména pro vylepšení zpětné vazby, když uživatele definují vlastní šablony popsané v sekci 5.1.1.

Sice můj nástroj dokázal být použitelným pro účely dokumentace, má ale stále pár míst pro vylepšení. V budoucnu bych chtěl vylepšit podporu pro aktualizaci popisu tříd, přidat srozumitelný způsob popisu instančních proměnných a speciálních metod definovaných pomocí `attr_accessor`. Další prostor pro vylepšení skýtá zobrazení výsledné dokumentace v dalších formátech, např. jako PDF soubor. Taky zvažuji přidat lepší a rozsáhlejší možnosti přizpůsobení HTML stránek uživatelem.

Literatura

[1] *The Foreman* [online]. [cit. 2020-05-01]. Dostupné z: https://theforeman.org/.

[2] *Ruby On Rails* [online]. [cit. 2020-05-01]. Dostupné z: https://rubyonrails.org/.

[3] *What is REST* [online]. [cit. 2020-05-01]. Dostupné z: https://restfulapi.net/.

[4] Dave Thomas, A. H. a Fowler, C. *Programming Ruby: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 4th Edition, 2013. ISBN 9781937785499.

[5] Fowler, M. *Domain Specific Languages*. Addison-Wesley Professional, 2010. ISBN 9780132107549.

[6] Hetsch, L. *Ruby modules: Include vs Prepend vs Extend* [online]. [cit. 2020-05-01]. Dostupné z: https://medium.com/@leo_hetsch/ruby-modules-include-vs-prepend-vs-extend-f09837a5b073.

[7] Matsumoto, Y. a Flanagan, D. *The Ruby Programming Language*. O'Reilly Media, Inc., 2008. ISBN 9780596516178.

[8] Perrotta, P. *Metaprogramming Ruby 2*. Pragmatic Bookshelf, 2014. ISBN 9781941222751.

[9] Pokorný, P. a Nečas, I. *Apipie-rails* [online]. [cit. 2020-05-01]. Dostupné z: https://github.com/Apipie/apipie-rails/blob/master/README.rst.

[10] Thomas, D. a Hunt, A. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, 1999. ISBN 9780201616224.

[11] Winter, C. a Lownds, T. *PEP 3107 – Function Annotations* [online]. [cit. 2020-05-01]. Dostupné z: https://www.python.org/dev/peps/pep-3107.

Příloha A

Ukázky DSL nástroje Apipie-rails

```
resource_description do
  short 'Site members'
  formats ['json']
  param :id, Fixnum, :desc => "User ID", :required => false
  param :resource_param, Hash,
        :desc => 'Param description for all methods' do
    param :ausername, String,
          :desc => "Username for login", :required => true
    param :apassword, String,
          :desc => "Password for login", :required => true
  end
  api_version "development"
  error 404, "Missing"
  error 500, "Server crashed for some <%= reason %>",
        :meta => {:anything => "you can think of"}
  error :unprocessable_entity, "Could not save the entity."
  returns :code => 403 do
     property :reason, String, :desc => "Why this was forbidden"
  end
  meta :author => {:name => 'John', :surname => 'Doe'}
  deprecated false
  description <<-EOS
    == Long description
     Example resource for rest api documentation
     These can now be accessed in <tt>shared/header</tt> with:
       Headline: <%= headline %>
       First name: <%= person.first_name %>

     If you need to find out whether a certain local variable has been
     assigned a value in a particular render call, you need to use the
     following pattern:

     <% if local_assigns.has_key? :headline %>
        Headline: <%= headline %>
```

```
    <% end %>

    Testing using <tt>defined? headline</tt> will not work. This is an
    implementation restriction.
  EOS
end
```

Výpis A.1: Příklad DSL pro popis zdrojů (převzato z [9])

```
# The simplest case: just load the paths from routes.rb
api!
def index
end

# More complex example
api :GET, "/users/:id", "Show user profile"
show false
error :code => 401, :desc => "Unauthorized"
error :code => 404, :desc => "Not Found",
      :meta => {:anything => "you can think of"}
param :session, String, :desc => "user is logged in", :required => true
param :regexp_param, /^[0-9]* years/, :desc => "regexp param"
param :array_param, [100, "one", "two", 1, 2],
      :desc => "array validator"
param :boolean_param, [true, false],
      :desc => "array validator with boolean"
param :proc_param, lambda { |val|
  val == "param value" ? true : "The only good value is 'param value'."
}, :desc => "proc validator"
param :param_with_metadata, String, :desc => "",
      :meta => [:your, :custom, :metadata]
returns :code => 200, :desc => "a successful response" do
   property :value1, String, :desc => "A string value"
   property :value2, Integer, :desc => "An integer value"
   property :value3, Hash, :desc => "An object" do
     property :enum1, ['v1', 'v2'],
              :desc => "One of 2 possible string values"
   end
end
tags %w[profiles logins]
tags 'more', 'related', 'resources'
description "method description"
formats ['json', 'jsonp', 'xml']
meta :message => "Some very important info"
example " 'user': {...} "
see "users#showme", "link description"
see :link => "users#update", :desc => "another link description"
```

```
def show
  #...
end
```
Výpis A.2: Příklad DSL pro popis koncových bodů (převzato z [9])

```
# v1/users_controller.rb
def_param_group :address do
  param :street, String
  param :number, Integer
  param :zip, String
end

def_param_group :user do
  param :user, Hash do
    param :name, String, "Name of the user"
    param_group :address
  end
end

api :POST, "/users", "Create an user"
param_group :user
def create
  # ...
end

api :PUT, "/users/:id", "Update an user"
param_group :user
def update
  # ...
end

# v2/users_controller.rb
api :POST, "/users", "Create an user"
param_group :user, V1::UsersController
def create
  # ...
end
```
Výpis A.3: Příklad DSL pro definici a použití skupiny parametrů (převzato z [9])

```
# ------------------------------------------------
# Example of format #1 (reference to param-group):
# ------------------------------------------------
# the param_group :pet is defined here to describe the output
# returned by the method below.
```

```
def_param_group :pet do
  property :pet_name, String, :desc => "Name of pet"
  property :animal_type, ['dog','cat','iguana','kangaroo'],
           :desc => "Type of pet"
end

api :GET, "/pets/:id", "Get a pet record"
returns :pet, :desc => "The pet"
def show_detailed
  render JSON({:pet_name => "Skippie", :animal_type => "kangaroo"})
end

# -------------------------------------------------
# Example of format #2 (inline):
# -------------------------------------------------
api :GET, "/pets/:id/with-extra-details", "Get a detailed pet record"
returns :code => 200, :desc => "Detailed info about the pet" do
  param_group :pet
  property :num_legs, Integer, :desc => "How many legs the pet has"
end
def show
  render JSON({:pet_name => "Barkie", :animal_type => "iguana",
               :legs => 4})
end

# -------------------------------------------------
# Example of format #3 (array response):
# -------------------------------------------------
api :GET, "/pets", "Get all pet records"
returns :array_of => :pet, :code => 200, :desc => "All pets"
def index
  render JSON([ {:pet_name => "Skippie", :animal_type => "kangaroo"},
                {:pet_name => "Woofie", :animal_type => "cat"} ])
end
```

Výpis A.4: Příklad DSL pro popis odpovědí serveru (převzato z [9])

Příloha B

Ukázky výsledné dokumentace Apipie-rails

NV API 1.0 / Posts / create

POST /api/v1/comments

Create a post

Errors

404 Something went wrong! Please Try again!
404 Comment was empty! Please Try again!

Params

Param name	Description
featured optional	Value: Must be String
remote_url optional	Value: Must be String
images optional	Value: Must be an array of any type
remote_title optional	Value: Must be String
post optional	Value: Must be Hash
remote_image_url optional	Value: Must be String

Obrázek B.1: Příklad HTML stránky s dokumentací (převzato z [9])

NV API 1.0

A language agnostic RESTful API for NV applications. It's currently on v.1.0

Resources

Comments

Resource	Description
POST /api/v1/comments	Create a comment
DELETE /api/v1/comments/:id	Delete a comment

Users

Resource	Description
GET /api/v1/users/:id	Show specific user
GET /api/v1/users	Index all users
POST /api/v1/users/:id	Update specific user
POST /api/v1/users/:id/change_country	Change user country
GET /api/v1/users/:id/wishes	Get user wished items
GET /api/v1/users/:id/closet	Get user owned garments
GET /api/v1/users/:id/liked_garments	Get user liked garments
GET /api/v1/users/:id/liked_garments	Check if email exists, used in user registration

© 2015 Wazery

Obrázek B.2: Příklad HTML stránky s dokumentací (převzato z [9])

Příloha C

Obsah přiloženého paměťového média

- **/apipie-dsl/** - zdrojový kód implementovaného nástroje
- **/dsl-example/** - příklad použití implementovaného nástroje
- **/text/** - zdrojový kód textu práce
- **/pdf/** - text práce ve formátu PDF