ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	RESTful API v jazyce Python
Student:	Bc. Miroslav Hrončok
Vedoucí:	Ing. Jakub Jirůtka
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2016/17

Pokyny pro vypracování

Cílem této práce je vypracovat rešerši existujících open-source frameworků pro tvorbu RESTful API v jazyce Python.

1. Prostudujte existující frameworky pro tvorbu RESTful API v jazyce Python. Zaměřte se zejména na realizaci konceptu HATEOAS, například specifikace JSON API, JSON-LD, HAL či obdobné, a řízení přístupových práv. Stanovte kvalitativní kritéria a na základě nich prostudované frameworky porovnejte.

2. Na základě analýzy vyberte alespoň tři nejvhodnější řešení a implementujte v nich ukázkovou službu pro přístup k rozvrhovým datům Ústavu tělesné výchovy a sportu, ČVUT v Praze. Před implementací analyzujte strukturu dat v poskytnutých databázových pohledech. Navrhněte nad těmito daty RESTful API s podporou formátu JSON a vhodný model přístupových práv.

3. Na základě zkušeností s implementací ukázkových služeb vybraná řešení dále zhodnoťte. Mimo jiné proveďte měření času odezvy jednotlivých implementací.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 3. února 2016

České vysoké učení technické v Praze

Fakulta informačních technologií

Katedra softwarového inženýrství

Diplomová práce

RESTful API v jazyce Python

Bc. Miroslav Hrončok

Vedoucí práce: Ing. Jakub Jirůtka

10. května 2016

Poděkování

Na tomto místě bych rád poděkoval a vyslovil uznání všem lidem, kteří mi pomáhali při vzniku této práce. V prvé řadě Jakubovi Jirůtkovi, vedoucímu mé diplomové práce, za cenné podněty k textové části i implementacím, ale také za poskytnutí řešení k psaní textové části této práce v jazyce Markdown.

Dále také Slávkovi Kabrdovi, vedoucímu mé bakalářské práce, za to, že mě naučil, jak závěrečné práce psát.

Děkuji i Jiřímu Mlejnkovi za návrh, jak psát kapitolu *Implementace*.

Děkuji Magdě Friedjungové, Báře Havelkové, Báře Hrončokové a Katce Hrončokové za pomoc s hledáním chyb.

V neposlední řadě také děkuji desítkám autorů zkoumaných frameworků, bez jejichž úsilí by tato práce neměla důvod vzniknout.

Prohlášení

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

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

V Praze dne 10. května 2016 ……

České vysoké učení technické v Praze

Fakulta informačních technologií

Odkaz na tuto práci

HRONČOK, Miroslav. *RESTful API v jazyce Python*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016. Dostupný také z WWW: ⟨`https://github.com/hroncok/diplomka`⟩.

Abstrakt

Tato diplomová práce si klade za cíl prozkoumat a porovnat dostupné open-source frameworky pro tvorbu REST API v jazyce Python, s důrazem na princip HATEOAS. Součástí zkoumání je i implementace ukázkové služby pro rozvrhová data ÚTVS ČVUT ve vybraných frameworcích.

Klíčová slova web, API, REST, RESTful, Python, HATEOAS, open-source

Abstract

This master's thesis aims to study and compare available open source Python frameworks for creating REST APIs, with emphasis on the HATEOAS principle. A part of the study is the implementation of a demo service for ÚTVS ČVUT timetable data in selected frameworks.

Keywords web, API, REST, RESTful, Python, HATEOAS, open-source

Obsah

Seznam ukázek kódu

Seznam tabulek

Seznam obrázků

Úvod

Jak říká Zen of Python [121], měl by být jeden – a nejlépe pouze jeden – zřejmý způsob, jak to udělat. V případě webových RESTful API to ale v Pythonu neplatí, existuje celá řada open-source knihoven a frameworků, které umožňují RESTful API vytvářet. Některé fungují pouze společně s frameworky na tvorbu webových aplikací, jako například s Flaskem nebo Djangem, jiné fungují samostatně.

V této diplomové práci se budu zabývat téměř dvacítkou těchto frameworků. V kapitole *Frameworky pro RESTful API* je zhodnotím z hlediska použitelnosti, množství nabízených funkcí, podpory standardů, množství závislostí, ale i z hlediska stavu a oblíbenosti projektu.

Abych mohl frameworky zkoumat více do hloubky a porovnat je i z hlediska rychlosti, navrhnu ukázkovou službu poskytující RESTful API pro přístup k rozvrhovým datům Ústavu tělesné výchovy a sportu ČVUT v Praze. Návrhem se budu zabývat v kapitole *Návrh API pro rozvrhová data ÚTVS ČVUT*.

Tuto službu pak implementuji ve čtyřech vybraných frameworcích na základě předchozího zkoumání a hodnocení; o tom prozradí více kapitola *Implementace*. Hotová řešení pak podrobím testování doby odezvy v kapitole *Měření odezvy*.

Kapitola 1

Frameworky pro RESTful API

V této kapitole představím osmnáct open-source frameworků pro tvorbu webových RESTful API v jazyce Python, které zhodnotím na základě mnou stanovených hodnotících kritérií.

1.1 Hodnotící kritéria

Než začnu zkoumání a hodnocení jednotlivých frameworků, je třeba si stanovit hodnotící kritéria, která mi umožní frameworky objektivně porovnávat a vybrat kandidáty pro kapitolu *Implementace*. Pokud to bude alespoň trochu možné, tak pro kritérium stanovím stupnici, na základě které bude možné frameworky mezi sebou porovnat.

1.1.1 Licence

Abychom vůbec mohli zvažovat použití některého frameworku, musíme zjistit, jestli nám to jeho licence umožňuje. V této práci se zabývám pouze open-source frameworky, takže bychom neměli narazit na zásadní problém. Typ licence ale může výrazně ovlivnit licenci díla, ve kterém framework použijeme, proto je dobré se touto otázkou zabývat.

Licence tedy rozdělím do skupin podle typu, pořadí typu v seznamu určuje stupnici od nejvolnější po nejstriktnější.

1. **Public domain** zahrnuje licence, které říkají, že si s frameworkem prakticky můžeme dělat, co chceme. Mezi takové řadím například Creative Commons CC0 [16] nebo WTFPL [54].
2. **Permisivní** licence jsou takové, které vyžadují například uvedení textu licence a jméno autora, ale neovlivňují licenci výsledného díla. Příkladem jsou licence MIT [115], BSD [113][114], ale i licence Pythonu [125].
3. **LGPL** je kategorie, která obsahuje GNU Lesser General Public License [37] a další podobné licence (například Mozilla Public License [100]), které v případě vhodného použití knihovny neovlivňují licenci díla. Pro potřeby použití frameworku se příliš neliší od předchozí skupiny, ale je třeba si dát pozor, jak framework použijeme; pokud bychom například kód z frameworku zkopírovali přímo do kódu našeho díla, mohli bychom výslednou licenci ovlivnit.
4. **Copyleft** licence jsou takové, které vyžadují, aby výsledné dílo v případě využití knihovny nebo frameworku převzalo jejich licenci [38]. Jako nejznámější exemplář jmenuji GNU General Public License [36].
5. **AGPL** je kategorie, která obsahuje GNU Affero General Public License [35] a případné další podobné licence, které navíc oproti předchozímu typu považují poskytování webové služby za distribuci díla a vyžadují tedy poskytnutí zdrojového kódu všem uživatelům služby.

1.1.2 Závislost na webovém frameworku

Některé frameworky fungují samostatně, jiné vyžadují konkrétní Python framework na tvorbu webových aplikací. Některé webové frameworky slouží čistě jako vrstva pro poskytovaní obsahu přes protokol HTTP, jiné striktně určují, jak bude webová aplikace vnitřně navržena. Škálu jsem tedy nastavil takto:

1. **Standalone** je kategorie pro frameworky, které lze pro RESTful API použít samostatně.
2. **Lightweight** je kategorie pro frameworky, které vyžadují webový microframework, který slouží pouze jako vrstva mezi Pythonem a HTTP. Takovými microframeworky jsou třeba Werkzeug [129], Flask [130], Pyramid [123] nebo Morepath [27].

3. **MVC** je kategorie pro frameworky typu *Model-view-controller*, především Django[1].

1.1.3 Velikost kódu včetně závislostí

Přestože dnes diskový prostor není tolik kritický jako dříve, čím víc kódu framework a jeho závislosti obsahují, tím více věcí se může zkomplikovat. Některé frameworky se označují za „lightweight“ a právě velikost kódové základny je jedním z faktorů, který vnímaní frameworku jako „lightweight“ může ovlivnit [132].

Měření budu provádět tak, že daný framework nainstaluji do prázdného *virtualenv*[2], a pak se podívám na jeho celkovou velikost (od té odečtu velikost „prázdného“ virtualenv) – ta bude určovat pořadí na stupnici.

1.1.4 Počet řádků kódu

Ještě důležitější než samotná velikost v MiB je počet řádek kódu – k velkosti mohou přispívat i jiné faktory, jako soubory s překlady, obrázky, šablonami apod. K měření použiji nástroj cloc [22], budu počítat pouze řádky v jazyce Python. Před měřením odstraním z modulů testy. Ve srovnávací tabulce budu uvádět jak počet řádků samotného frameworku, tak celého závislostního aparátu.

1.1.5 Počet závislostí

Kromě samotné velikosti je třeba zkoumat i kolik závislostí (přímých i nepřímých) daný framework vyžaduje. Každá závislost představuje riziko i zranitelnost [131]. Jelikož čtenáře může zajímat počet přímých i počet nepřímých závislostí, budu uvádět vždy obě čísla.

[1] Django samo sebe označuje jako MTV (*Model-template-view*) framework, prakticky se však jedná o MVC princip [55].

[2] Virtualenv je virtuální prostředí pro jazyk Python umožňující instalovat závislosti různých projektů do oddělených míst. [127]

1.1.6 Podpora Pythonu 3

Přestože Python 3 vyšel již v roce 2008 [124], některé knihovny třetích stran jej stále ještě nepodporují [58]. Bohužel je tedy třeba se zabývat i tím, jestli framework podporuje Python 3. Stejně tak může být pro někoho důležité, jestli framework podporuje Python 2, například kvůli tomu, že některé knihovny, které používá, Python 3 nepodporují.

Škálu jsem tedy stanovil takto:

1. podpora obou verzí Pythonu,
2. podpora pouze pro Python 3,
3. podpora pouze pro Python 2.

1.1.7 Oblíbenost

Čím více lidí a projektů daný framework využívá, tím větší je šance, že v případě problému najdeme hotové řešení. Oblíbenost je subjektivní pojem, a tak se špatně měří, využiji ale dva prvky, které o oblíbenosti mohou něco prozradit.

Většina zkoumaných frameworků má svůj kód zveřejněn na GitHubu, kde uživatelé mohou jednotlivé projekty zařadit mezi své oblíbené tím, že jím dají hvězdu (*star*) [48]. Počet těchto hvězd pak může poskytnout určitou vypovídající hodnotu.

Frameworky jsou zároveň distribuované přes *Python Package Index*, kde lze vidět počet stažení za poslední den, týden a měsíc [126]. Tyto informace jsou však často zkreslené kvůli různým automatickým nástrojům, které stahují všechny balíčky [1]. Budu uvádět jen hodnotu stažení za poslední měsíc, v době psaní tohoto textu.

1.1.8 Podpora HATEOAS

HATEOAS, tedy *Hypermedia as the Engine of Application State*[3], je jedním ze základních stavebních kamenů REST architektury [31]. Díky principu HATEOAS nemusí REST klient o poskytovaném API vědět příliš mnoho

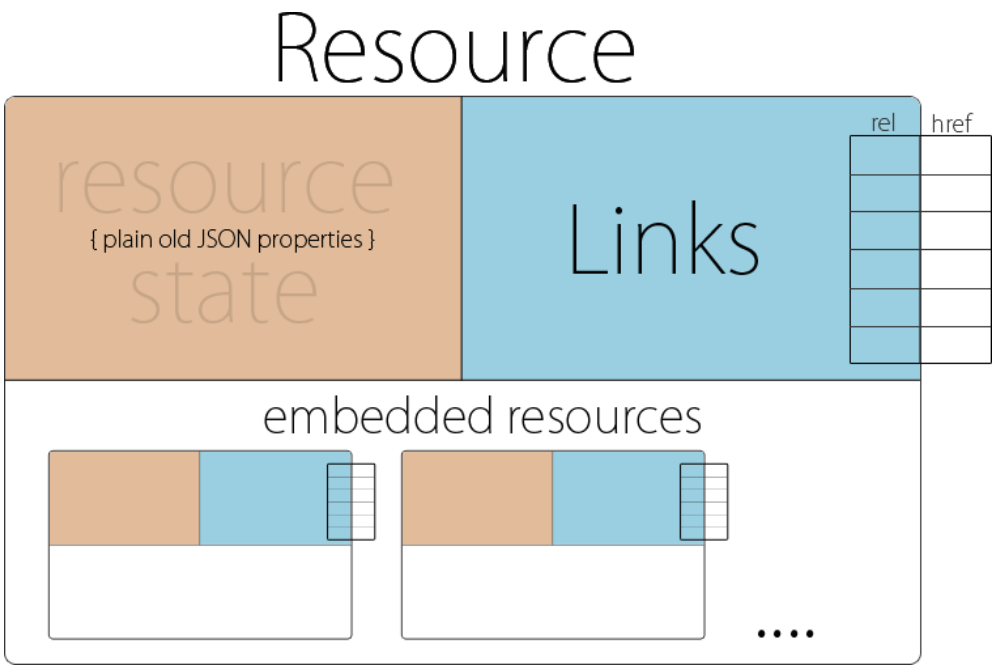
[3]Hypermedia jako základ aplikačního stavu

informací předem. V ideálním případě mu stačí adresa kořenového zdroje a všechny další informace (adresy souvisejících zdrojů, proveditelných akcí…) zjistí dynamicky z odpovědí serveru – obdobně jako uživatel při procházení HTML stránek.

HATEOAS je ale pouze princip, konkrétních implementací existuje hned několik. Mezi ty nejznámější patří následující.

HAL

HAL (Hypertext Application Language) je jednoduchý formát, který nabízí konzistentní způsob prolinkování zdrojů v API [79]. Definuje atributy `_links` a `_embedded` pro odkazy a vnořené zdroje, šablony pro odkazy na navazující zdroje a konvenci pro odkazování dokumentace. Schéma můžete vidět na obrázku 1.1.



Obrázek 1.1: Schéma zdroje ve formátu HAL [79]

JSON-LD

JSON-LD je formát pro serializaci prolinkovaných dat [78]. Používá se mj. pro sémantický web a RDF data [23], ale lze jej použít i pro REST API. Příklad můžete vidět v ukázce 1.1.

```
{
  "@context": "http://json-ld.org/contexts/person.jsonld",
  "@id": "http://dbpedia.org/resource/John_Lennon",
  "name": "John Lennon",
  "born": "1940-10-09",
  "spouse": "http://dbpedia.org/resource/Cynthia_Lennon"
}
```

Ukázka kódu 1.1: Příklad formátu JSON-LD [78]

Hydra (rozšíření JSON-LD)

Hydra je rozšíření pro JSON-LD, které využívá speciální slovník vhodný pro webová API [87].

JSON API

JSON API je specifikace pro webová API využívající JSON [45]. Jedná se o velmi komplexní formát, který u každého zdroje rozlišuje data, metadata, odkazy, vztahy a další prvky.

Collection+JSON

Collection+JSON je komplexní serializační formát postavený na JSONu určený pro kolekce dat [2]. Příklad můžete vidět v ukázce 1.2.

Siren

Siren je specifikace pro reprezentaci entit pomocí hypermédií [136]. Příklad můžete vidět v ukázce 1.3.

Vzhledem ke komplexitě možných případů nestanovuji škálu pevně, ale na základě vlastního textového hodnocení ohodnotím každý framework nula až třemi body.

```
{ "collection" :
  {
    "version" : "1.0",
    "href" : "http://example.org/friends/",

    "links" : [
      {"rel" : "feed", "href" : "http://example.org/friends/rss"}
    ],

    "items" : [
      {
        "href" : "http://example.org/friends/jdoe",
        "data" : [
          {"name" : "full-name", "value" : "J. Doe",
           "prompt" : "Full Name"},
          {"name" : "email", "value" : "jdoe@example.org",
           "prompt" : "Email"}
        ],
        "links" : [
          {"rel" : "blog", "href" : "http://examples.org/blogs/jdoe",
          "prompt" : "Blog"},
          {"rel" : "avatar",
           "href" : "http://examples.org/images/jdoe",
           "prompt" : "Avatar", "render" : "image"}
        ]
      }
    ],

    "queries" : [
      {"rel" : "search", "href" : "http://example.org/friends/search",
       "prompt" : "Search",
        "data" : [
          {"name" : "search", "value" : ""}
        ]
      }
    ],

    "template" : {
      "data" : [
        {"name" : "full-name", "value" : "", "prompt" : "Full Name"},
        {"name" : "email", "value" : "", "prompt" : "Email"},
        {"name" : "blog", "value" : "", "prompt" : "Blog"},
        {"name" : "avatar", "value" : "", "prompt" : "Avatar"}
      ]
    }
  }
}
```

Ukázka kódu 1.2: Příklad formátu Collection+JSON [2]

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

Ukázka kódu 1.3: Příklad formátu Siren [136]

1.1.9 Přístupová práva

Některé frameworky přístupová práva vůbec neřeší, jiné podporují jen autentizaci, ale ne různá práva pro různé klienty a různé zdroje, další obsahují mechanismy a postupy, jak autentizaci a autorizaci řešit. Některé dokonce obsahují předpřipravená řešení pro nejčastější případy, jako je HTTP autentizace uživatelským jménem a heslem nebo OAuth.

Vzhledem ke komplexitě možných případů nestanovuji škálu pevně, ale na základě vlastního textového hodnocení ohodnotím každý framework nula až třemi body.

1.1.10 Použitelnost

Jak je framework použitelný a přívětivý pro programátora se velice špatně stanovuje. Jedná se víceméně o subjektivní dojem; to, co jeden programátor považuje za přívětivé, jiný může považovat za příliš složité.

Místo vynášení soudů o použitelnosti, založených čistě na mém osobním názoru, nabídnu u každého frameworku ukázku z dokumentace, aby čtenář mohl použitelnost sám posoudit.

Jednotlivé ukázky se liší délkou i účelem. Ukázky z vybraných frameworků sloužící ke stejnému účelu najdete v kapitole *Implementace*.

1.1.11 Stav projektu

Pokud se rozhodujeme, jestli využít nějaký framework, mohly by nás zajímat i informace o projektu, jako například:

- Kdo projekt tvoří; jsou to jednotlivci, firma?
- Je projekt aktivně vyvíjen?
- Vycházejí nové verze?
- Reaguje někdo na hlášené chyby?
- Jsou přijímány úpravy od lidí mimo projekt?
- Jak dlouho již projekt existuje?
- Jak často vycházejí nové verze?
- Má projekt dokumentaci? Je aktuální?

Tyto informace se dají jen velice těžko srovnávat pomocí číselné škály, proto se pokusím na tyto otázky odpovědět alespoň textově.

1.2 Cornice

Cornice je RESTový framework pro Pyramid [101]. Je vyvíjen lidmi z Mozilla Services a vydán pod Mozilla Public License 2.0 [100], čímž se řadí do kategorie LGPL. Závisí na dvou dalších modulech (`pyramid` a `simplejson`), čímž nepřímo závisí na celkem devíti modulech a má s nimi 124 625 řádek kódu. Podporuje obě verze Pythonu. Na GitHubu má 270 hvězd a za poslední měsíc byl stažen více než desettisíckrát.

Projekt vznikl v roce 2011 a od té doby vyšla již více než dvacítka verzí. V době zkoumání byla nejnovější verze pouze několik týdnů stará, proto vývoj hodnotím jako aktivní. Na vývoji se podílelo více než šest desítek vývojářů, většina z nich formou drobné úpravy, která bývá rychle přijata i od lidí mimo projekt a mimo Mozilla Services.

V ukázce kódu 1.4 najdete příklad použití Cornice. V ukázce je definována služba, která umožňuje použít GET a POST na nějakou hodnotu `/values/{value}`, kde `value` reprezentuje ASCII název té hodnoty.

1.2.1 HATEOAS

Cornice nenabízí předem připravené mechanismy k prolinkování zdrojů. Pokud chcete použít JSON API, HAL či další obdobné standardy, budete je muset dodržet a naimplementovat sami. Cornice nezískává žádný bod.

1.2.2 Přístupová práva

Cornice nenabízí žádné zabudované možnosti, jak řešit přístupová práva. Ve výchozím stavu je celé API přístupné všem, můžete však napsat vlastní funkci v Pythonu, která práva bude řešit. Toto na jednu stranu nabízí téměř neomezené možnosti, na stranu druhou to není příliš pohodlné. Cornice získává jeden bod.

```python
from cornice import Service

values = Service(name='foo', path='/values/{value}',
                 description="Cornice Demo")

_VALUES = {}


@values.get()
def get_value(request):
    """Returns the value.
    """
    key = request.matchdict['value']
    return _VALUES.get(key)


@values.post()
def set_value(request):
    """Set the value.

    Returns *True* or *False*.
    """
    key = request.matchdict['value']
    try:
        # json_body is JSON-decoded variant of the request body
        _VALUES[key] = request.json_body
    except ValueError:
        return False
    return True
```

Ukázka kódu 1.4: Příklad použití z dokumentace Cornice [102]

Cornice působí jako solidní nízkoúrovňový REST framework: Pokud víte, co děláte, můžete pomocí něj naimplementovat REST službu, ale neudělá příliš věcí za vás. Speciální funkcí je pak podpora SPORE[4] [103].

1.3 Django REST framework

Django REST framework je nadstavbou k webovému frameworku Django. Na svém webu [62] uvádí tyto přednosti:

- webově procházetelné API (ukázku můžete vidět na na obrázku 1.3),
- autentizační pravidla včetně možnosti použití OAuth 1 či OAuth 2,

[4]Specification to a POrtable Rest Environment [25]

- serializace pro ORM[5] i jiná data,
- upravitelné na míru,
- rozsáhlá dokumentace a výborná komunitní podpora,
- používají ho i velké společnostmi jako Mozilla a Eventbrite.

Obrázek 1.2: Logo Django REST frameworku [62]

Za vývojem Django REST frameworku nestojí žádná firma, ale jednotlivec, Tom Christie. Kromě něj ale do projektu přispělo více než pět set přispěvatelů [47]. Projekt navíc absolvoval crowdfundingovou kampaň a podpořilo jej mnoho firem i jednotlivců [60]. Zveřejněn je podobně jako Django pod BSD licencí [113].

Příklad použití z dokumentace, mírně upravený, aby se vešel na stránku, najdete v ukázce 1.5.

První verze byla vydána již v roce 2011 a od té doby jich vyšlo celkem osmdesát; poslední cca tři týdny před psaním tohoto textu. Vývoj je velmi aktivní a dokumentace obsáhlá. Instalace závisí pouze na Djangu, které nemá žádné další závislosti, a společně s ním má 79 854 řádků kódu. Podporovány jsou obě verze Pythonu. Na GitHubu má více než pět a půl tisíce hvězd a za poslední měsíc byl více než třistatisíckrát stažen z PyPI.

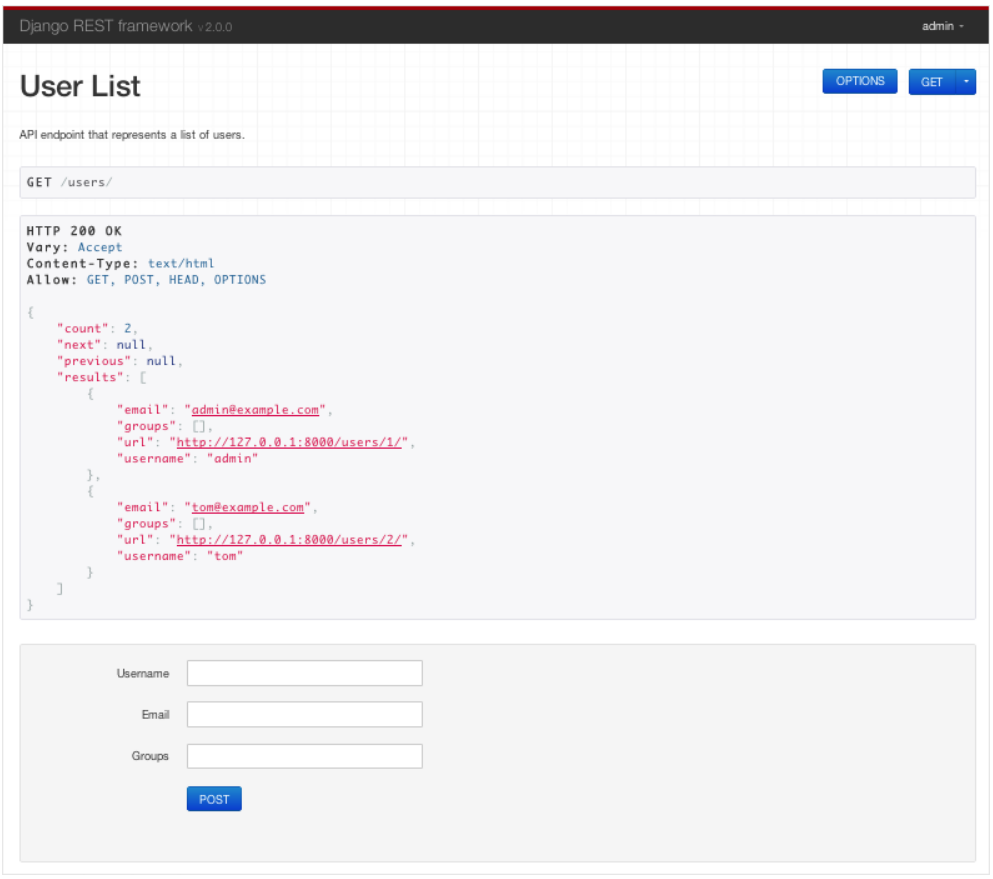
1.3.1 HATEOAS

Dokumentace obsahuje kapitolu o HATEOAS [63], která uvádí:

[5]Object-relational mapping neboli Objektově relační zobrazení [33]

```
# Serializers
from django.contrib.auth.models import User, Group
from rest_framework import serializers

class UserSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = User
        fields = ('url', 'username', 'email', 'groups')

class GroupSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Group
        fields = ('url', 'name')

# Views
from django.contrib.auth.models import User, Group
from rest_framework import viewsets
from tutorial.quickstart.serializers import UserSerializer
from tutorial.quickstart.serializers import GroupSerializer

class UserViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows users to be viewed or edited.
    """
    queryset = User.objects.all().order_by('-date_joined')
    serializer_class = UserSerializer

class GroupViewSet(viewsets.ModelViewSet):
    """
    API endpoint that allows groups to be viewed or edited.
    """
    queryset = Group.objects.all()
    serializer_class = GroupSerializer

# URLs
from django.conf.urls import url, include
from rest_framework import routers
from tutorial.quickstart import views

router = routers.DefaultRouter()
router.register(r'users', views.UserViewSet)
router.register(r'groups', views.GroupViewSet)

# Wire up our API using automatic URL routing.
# Additionally, we include login URLs for the browsable API.
urlpatterns = [
    url(r'^', include(router.urls)),
    url(r'^api-auth/', include('rest_framework.urls',
                               namespace='rest_framework'))
]
```

Ukázka kódu 1.5: Příklad použití z dokumentace Django REST frameworku [61]



Obrázek 1.3: Django REST framework: Webově procházetelné API [62]

Je zřejmé, že (Django) REST framework umožňuje vytvářet Hypermedia API. Procházetelné API, které nabízí, je postaveno na HTML – jazyku hypermédií webu.

(Django) REST framework také obsahuje serializační, parsovací a renderovací komponenty usnadňující vytváření patřičných mediálních typů, hyperlinkovaných vazeb pro konstrukci vysoce propojených systémů, a dobrou podporu pro vyjednávání obsahu.

Co ale (Django) REST framework neumí, je vytváření strojově čitelných formátů hypermédií jako HAL, Collection+JSON, JSON API či HTML mikroformátů ve výchozí konfiguraci, či automagické generování plně HATEOAS API s hypermediálním popisem formulářů a sémanticky označenými hyperlinky. To by vyžadovalo provedení

takových rozhodnutí o designu API, která by ve skutečnosti měla zůstat vně pole působnosti tohoto frameworku.

Hodnotím kladně, že se o principu HATEOAS v dokumentaci hovoří. Prolinkování zdrojů ve stylu HATEOAS je v Django REST frameworku jednoduché. Naplnění konkrétních implementací však zůstává v režii architekta či programátora; Django REST framework k tomu poskytuje dostatečné možnosti, uděluji mu tedy tři body.

1.3.2 Přístupová práva

Django REST framework umožňuje jak různé způsoby autentizace [64], tak i autorizace [65]. V základu poskytuje autentizaci:

- uživatelským jménem a heslem,
- tokenem,
- pomocí session.

Zároveň je možné implementovat vlastní způsoby. Existují další Python moduly, které tuto možnost využívají a přidávají tak do Django REST frameworku další možnosti autentizace, mezi ty nejdůležitější patří moduly pro OAuth 1 i OAuth 2. Například Django REST framework OAuth je modul, který byl dříve součástí Django REST frameworku, ale nyní je spravován samostatně.

Přístupová práva je možné řešit na úrovni objektů, modelů či konkrétních pohledů, tedy vlastně na základě URL. Pro první dvě kategorie se používají zabudované mechanismy přímo z Djanga. Jsou rozlišena práva pro čtení a pro zápis. Stejně jako u autentizace, i zde je možné si napsat vlastní způsob rozhodování přístupových práv a i zde vzniklo několik modulů třetích stran.

Možností poskytuje mnoho, Django REST framework tedy dostává i zde tři body.

Celkově Django REST framework působí jako obsáhlý nástroj poskytující mnoho možností a funkcí. Jeho jedinou zjevnou slabinou je těsná provázanost s Djangem.

1.4 Eve

Obrázek 1.4: Logo Eve [116]

Eve je open-source REST API framework navržený „pro lidi“. Umožňuje snadno vytvořit a nasadit vysoce upravitelné, plně funkční RESTful webové služby. Eve stojí nad nástroji Flask, Redis, Cerberus, Events a podporuje MongoDB i SQL backendy. [71]

Eve vychází z následujícího principu: Máte nějaká data a chcete k nim vytvořit REST API, pokud možno co nejvíce automaticky. Prakticky bez práce nabízí mj. tyto funkce a možnosti [72]:

- filtrování,
- řazení,
- stránkování,
- projekce,
- vnořené zdroje,
- JSON nebo XML serializaci,
- HATEOAS,
- ukládání souborů,
- limitování přístupu,
- cache,
- hromadné vkládání,
- kontrolu integrity (pomocí ETagu),
- validaci dat,
- GeoJSON,
- autentizaci a autorizaci,
- podporu obou verzí Pythonu i PyPy,
- verzování API,
- generovanou dokumentaci.

Jak vidno, možností poskytuje opravdu mnoho. Příklad použití si můžete prohlédnout v ukázce kódu 1.6.

Projekt vznikl v roce 2012, od té doby vyšlo dvacet verzí, poslední cca tři týdny před psaním tohoto textu. Jedná se tedy o aktivní projekt. Stojí za ním jednotlivec, Nicola Iarocci, a přispělo do něj více než sto dalších přispěvatelů [46]. Eve je vydáno pod BSD licencí [114].

Eve závisí celkem na deseti modulech (včetně Flasku a Werkzeugu) a tyto moduly již nemají žádné další závislosti. Celkem se závislostmi má 35 009 řádků kódu. Závislost na Python modulech pro MongoDB bohužel není volitelná.

1.4.1 HATEOAS

Eve automaticky prolinkovává jednotlivé zdroje a drží se konceptu HATEOAS [70]. Tuto funkci není potřeba speciálně nastavovat ani implementovat, je implicitně zapnutá. Každá odpověď na metodu GET obsahuje položku `_links` s odkazy na rodiče, subsekce, předchozí a další stránky apod. Příklad můžete vidět v ukázce 1.7.

Autoři pracují na přímé podpoře pro JSON-LD, HAL i Siren [72].

V této oblasti Eve hodnotím třemi body.

1.4.2 Přístupová práva

Eve umožňuje několik způsobů autentizace, například pomocí tokenu nebo HMAC[6] [73]. Pomocí externích knihoven je snadné přidat i OAuth 2 [68].

Eve umožňuje nastavovat přístupová práva podle rolí pro celé API, nebo jen pro některé zdroje, stejně tak pro konkrétní HTTP metody [73].

Dávám tedy i zde Eve tři body.

Celkově se Eve jeví jako framework s mnoha funkcemi, který dokáže ušetřit velké množství práce. Vytknul bych snad jen přílišnou vázanost na MongoDB, která je často patrná především z dokumentace.

[6] Hash Message Authentication Code [86]

```
# run.py
from eve import Eve
app = Eve()

if __name__ == '__main__':
    app.run()


# settings.py
DOMAIN = {'people': {}}


# GET /
{
    "_info": {
        "server": "Eve",
        "version": "a.b.c",
        "api_version": "x.y.z"
    },
    "_links": {
        "child": [
            {
                "href": "people",
                "title": "people"
            }
        ]
    }
}


# GET /people
{
    "_items": [],
    "_links": {
        "self": {
            "href": "people",
            "title": "people"
        },
        "parent": {
            "href": "/",
            "title": "home"
        }
    }
}
```

Ukázka kódu 1.6: Příklad použití z dokumentace Eve [69]

```
{
    "_links": {
        "self": {
            "href": "people",
            "title": "people"
        },
        "parent": {
            "href": "/",
            "title": "home"
        },
        "next": {
            "href": "people?page=2",
            "title": "next page"
        },
        "last": {
            "href": "people?page=10",
            "title": "last page"
        }
    }
}
```

Ukázka kódu 1.7: Příklad HATEOAS principu z Eve [70]

1.5 Falcon

Falcon je neuvěřitelně rychlý, minimalistický Python webový framework pro tvorbu „cloudových API“ a aplikačních backendů [49]. Mezi hlavní přednosti podle webové stránky [49] patří:

- závislost pouze na modulech `six` a `mimeparse`,
- rychlejší zpracování požadavků než u jiných populárních frameworků,
- podpora WSGI, CPythonu 2.6, 2.7, 3.3 a 3.4 i PyPy,
- svoboda volby detailů,
- spolehlivost.

Obrázek 1.5: Logo Falconu [50]

Falcon je bezesporu minimalistický – společně se závislostmi má pouze 3 034 řádků kódu. Je šířen pod permisivní Apache licencí [3] a nevyžaduje žádný webový framework.

Příklad použití můžete najít v ukázce 1.8. Jak je vidět, pomocí Falconu jdou vytvářet REST API, ale jedná se o velmi nízkoúrovňový framework, který spíše zastává vrstvu mezi HTTP a aplikací než velkého pomocníka při tvorbě API.

```
# sample.py
import falcon
import json

class QuoteResource:
    def on_get(self, req, resp):
        """Handles GET requests"""
        quote = {
            'quote': 'I\'ve always been more interested '
                     'in the future than in the past.',
            'author': 'Grace Hopper'
        }

        resp.body = json.dumps(quote)

api = falcon.API()
api.add_route('/quote', QuoteResource())
```

Ukázka kódu 1.8: Příklad použití z webu Falconu [49]

Projekt vytváří firma Rackspace pod vedením Kurta Griffithse. Do projektu přispívají i jednotlivci mimo Rackspace. Vznikl v roce 2012 a od té doby vyšlo celkem 27 verzí. Dva týdny před psaním tohoto textu vyšla verze 1.0.0rc1, brzy se tedy můžeme těšit na verzi 1.0.0. Jedná se o aktivní projekt, který se může chlubit stoprocentním pokrytím testy [15].

Vzhledem k nízkoúrovnosti frameworku neexistují žádné automatické mechanismy pro správu přístupových práv či HATEOAS. Falcon tedy za oba aspekty získává nula bodů.

1.6 hug (rozšíření pro Falcon)

Cílem hugu je, aby vytváření API v Pythonu bylo co nejjednodušší. Pomocí hugu lze vytvářet API nejen pro HTTP, ale i pro další média, například CLI aplikace. [18]

Obrázek 1.6: Logo hugu [20]

Mezi hlavní cíle hugu patří [19]:

- umožnit psaní tak stručného kódu Python API, jako by šlo o psanou definici,
- framework by měl podporovat psaní srozumitelného kódu,
- měl by být dostatečně rychlý; vývojář by neměl mít potřebu se kvůli výkonu poohlížet jinam,
- psaní testů pro API napsaná v hugu by mělo být jednoduché a intuitivní,
- magie by se měla odehrávat jen na jednom místě, ve frameworku, což je lepší než delegovat tento problém na uživatele,
- být základním kamenem API nové generace, díky nejnovějším technologiím.

Kvůli poslednímu bodu je hug kompatibilní pouze s Pythonem 3 a pro webová API staví na frameworku Falcon, o kterém jsem psal v části 1.5 [19].

Příklad použití s využitím typové anotace dostupné od Pythonu 3.5 můžete vidět v ukázce 1.9.

Hug je mladý projekt, vznikl teprve v červenci roku 2015. Více než tři tisíce hvězd na GitHubu za tak krátkou dobu ale napovídá, že půjde o projekt oblíbený; z PyPI byl stažen za poslední měsíc více než sedmtisíckrát. Vývoj

```
"""First hug API (local and HTTP access)"""
import hug


@hug.get(examples='name=Timothy&age=26')
@hug.local()
def happy_birthday(name: hug.types.text, age: hug.types.number,
                   hug_timer=3):
    """Says happy birthday to a user"""
    return {'message': 'Happy {0} Birthday {1}!'.format(age, name),
            'took': float(hug_timer)}


# GET /happy_birthday?name=Timothy&age=26
{
    "took": 0,
    "message": "Happy 26 Birthday Timothy"
}

# GET /happy_birthday?name=Timothy
{
    "errors": {
        "age": "Required parameter not supplied"
    }
}

# GET /happy_birthday?name=Timothy&age=twentysix
{
    "errors": {
        "age": "Invalid whole number provided"
    }
}
```

Ukázka kódu 1.9: Příklad použití z dokumentace hugu [17]

probíhá docela rapidně, již vyšlo více než čtyřicet verzí, průměrně tedy vychází rychleji než jednou týdně. Na to doplácí především dokumentace, která zdaleka neobsahuje všechny možnosti hugu; postrádá například kapitolu o autentizaci, přestože v kódu je tato funkcionalita obsažena. Za projektem stojí jednotlivec Timothy Edmund Crosley, ale přispěla již třicítka vývojářů.

Hug je zveřejněn pod MIT licencí [115]. Přímo závisí na Falconu a knihovně Requests, nepřímo tak má 4 závislosti a společně s nimi 16 545 řádků kódu.

1.6.1 HATEOAS

Hug bohužel zatím nepodporuje žádné automatické způsoby pro prolinkování jednotlivých zdrojů, nedostává tedy žádný bod.

1.6.2 Přístupová práva

Jak již bylo zmíněno výše, o přístupových právech dokumentace mlčí. Z pohledu do kódu [21] je však patrné, že podporuje autentizaci pomocí:

- HTTP Basic (jménem a heslem),
- API klíče v HTTP hlavičce,
- tokenu v HTTP hlavičce.

O autorizaci jsem však v kódu nic nenašel, proto dávám hugu pouze dva body.

Hug je moderní framework pro vytváření různých API v Pythonu. Jeho filozofie je rozhodně zajímavá, ale v současnosti jej hodnotím jako příliš mladý a zatím stále se rozvíjející projekt.

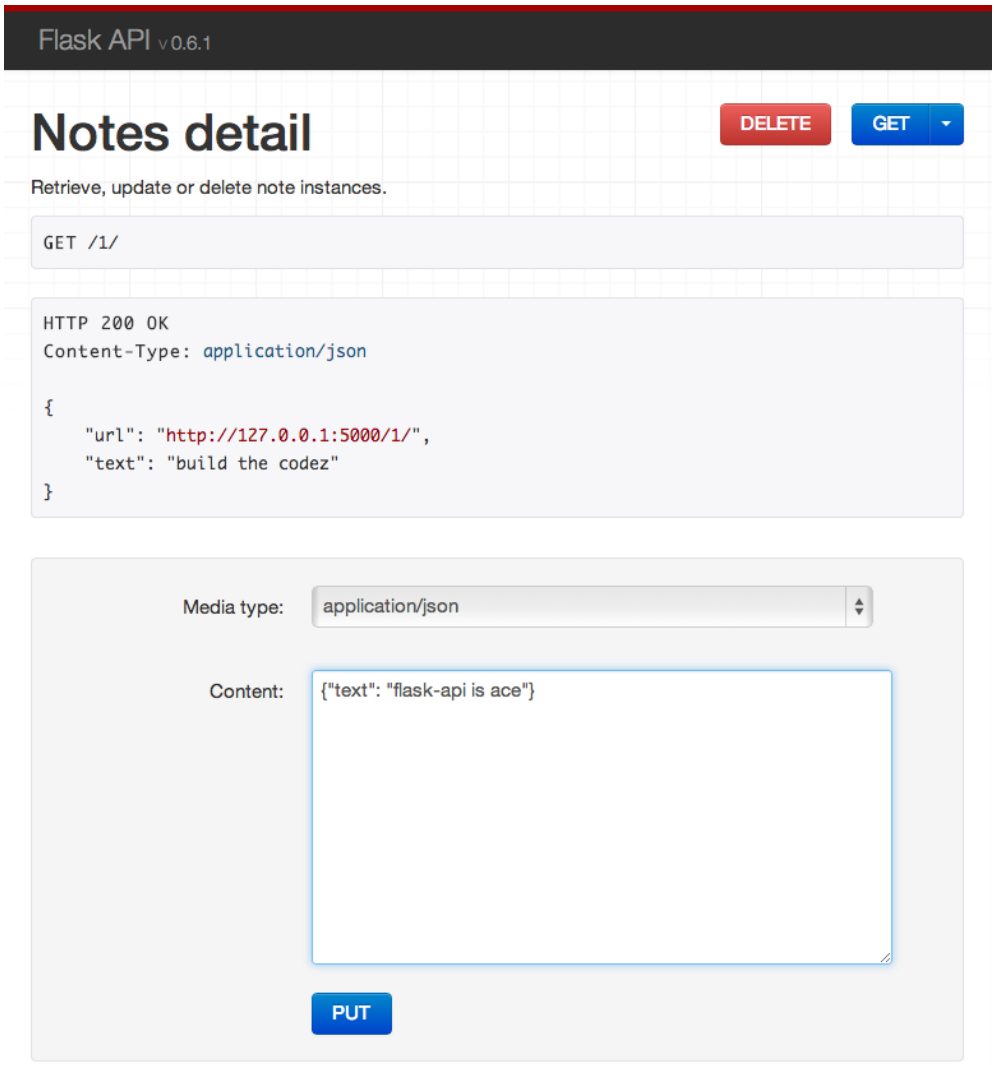
1.7 Flask API

Flask API je implementace stejných webově procházetelných API, které poskytuje Django REST framework [59] (o něm jsem psal v části 1.3 na straně 35), ale bez závislosti na Djangu.

Za projektem stojí autor Django REST frameworku, Tom Christie. Zatím se ale naneštěstí jedná o rozdělanou práci [59] a zdaleka nejde o hotový projekt. Práce na Flask API je zároveň pozastavená, kvůli závazkům z crowdfundingové kapaně k Django REST frameworku [67]. Tom Christie se od roku 2014 (kdy projekt i vznikl) na projektu aktivně nepodílí, existují však další jednotlivci, kteří projekt udržují a rozvíjí.

Pomocí Flask API je nyní možné webově procházet API, jak můžete vidět na obrázku 1.7, ale tvorba API zatím není příliš automatizovaná, viz příklad 1.10[7].

[7] Příklad byl mírně zhuštěn za účelem lepší prezentace na straně formátu A4.



Obrázek 1.7: Flask API: Webově procházetelné API [59]

```
from flask import request, url_for
from flask_api import FlaskAPI, status, exceptions

app = FlaskAPI(__name__)

notes = {
    0: 'do the shopping',
    1: 'build the codez',
    2: 'paint the door',
}

def note_repr(key):
    return {
        'url': request.host_url.rstrip('/') + \
               url_for('notes_detail', key=key),
        'text': notes[key]
    }

@app.route("/", methods=['GET', 'POST'])
def notes_list():
    """List or create notes."""
    if request.method == 'POST':
        note = str(request.data.get('text', ''))
        idx = max(notes.keys()) + 1
        notes[idx] = note
        return note_repr(idx), status.HTTP_201_CREATED

    # request.method == 'GET'
    return [note_repr(idx) for idx in sorted(notes.keys())]

@app.route("/<int:key>/", methods=['GET', 'PUT', 'DELETE'])
def notes_detail(key):
    """Retrieve, update or delete note instances."""
    if request.method == 'PUT':
        note = str(request.data.get('text', ''))
        notes[key] = note
        return note_repr(key)

    elif request.method == 'DELETE':
        notes.pop(key, None)
        return '', status.HTTP_204_NO_CONTENT

    # request.method == 'GET'
    if key not in notes:
        raise exceptions.NotFound()
    return note_repr(key)

if __name__ == "__main__":
    app.run(debug=True)
```

Ukázka kódu 1.10: Příklad použití z dokumentace Flask API [67]

Projekt v současnosti přímo závisí jen na frameworku Flask, tím má nepřímo pět závislostí a s nimi 20 938 řádků kódu. Je distribuován pod BSD licencí [113] a podporuje obě verze Pythonu.

Do budoucna je plánováno [59][67]:

- autentizace, mj. pomocí session, tokenu i jménem a heslem,
- přístupová práva,
- limitování počtu požadavků v čase,
- API zdroje pomocí tříd,
- vylepšení procházetelných API, například přidání drobečkové navigace,
- možnost vlastního zpracování výjimek,
- ochrana proti CSRF,
- přihlašování a odhlašování přes prohlížeč v případě procházetelných API,
- zdokumentování validace požadavků a prolinkování.

Je však otázka, kdy a jestli se toho dočkáme.

Zatím neexistují žádné automatické mechanismy pro správu přístupových práv či HATEOAS. Flask API tedy za oba aspekty získává nula bodů.

1.8 Flask-RESTful

Obrázek 1.8: Logo Flask-RESTful [56]

Flask-RESTful je rozšíření Flasku, které přidává podporu pro rychlé vytváření RESTových API. Jedná se o tenkou vrstvu abstrakce, která by měla fungovat s existujícím ORM a dalšími knihovnami. Flask-RESTful je navržen tak, aby ho uživatelé obeznámení s Flaskem co nejrychleji pochopili. [13]

Za vývojem Flask-RESTful stojí firma Twilio, ale přispělo do něj více než sto jednotlivců. Je zveřejněn pod BSD licencí [114]. Závisí na Flasku a dalších třech

modulech. Celkově tak nepřímo závisí na modulech devíti a společně s nimi má 27 718 řádků kódu. Na GitHubu má necelé dva tisíce hvězd a na PyPI má za poslední měsíc více než 170 tisíc stažení. Podporuje obě verze Pythonu. Projekt vznikl v roce 2012, od té doby vyšlo 27 verzí, poslední v prosinci roku 2015.

Příklad použití můžete vidět v ukázce 1.11[8].

Flask-RESTful je nízkoúrovňový framework, který zjednodušuje tvorbu REST API oproti použití čistého Flasku, ale nepřináší žádné pokročilé funkce jako podporu autentizace a autorizace, či prolinkování a HATEOAS. Nedostává tedy žádné body. Ze zajímavých funkcí Flask-RESTful mohu jmenovat vyjednávání o obsahu či podporu *blueprintů* (koncept z Flasku [128]).

1.9 Morepath

Obrázek 1.9: Koncept loga Morepathu [5]

Morepath je webový mikroframework podobně jako Flask nebo Bottle [27]. Nepatří tak úplně mezi frameworky na vytváření RESTových API, zařadil jsem jej proto, že přímo v sobě obsahuje součásti pro jejich tvorbu [26]. Na rozdíl od jiných mikroframeworků je modelově orientovaný [27].

Projekt vznikl v roce 2013, ale jeho historie sahá dále do minulosti [30]. Od roku 2013 vyšlo téměř dvacet verzí, poslední tři dny před psaním tohoto textu. Morepath přímo závisí na čtyřech a nepřímo na pěti modulech, společně s nimi má 9 156 řádků kódu. Je distribuován pod BSD licencí [114]. Autorem projektu je Martijn Faassen z firmy CONTACT Software a přispělo do něj celkem 14

[8]Příklad byl mírně zhuštěn za účelem lepší prezentace na straně formátu A4.

```
from flask import Flask
from flask_restful import reqparse, abort, Api, Resource

app = Flask(__name__)
api = Api(app)

TODOS = {'todo1': {'task': 'build an API'}, ...}

def abort_if_todo_doesnt_exist(todo_id):
    if todo_id not in TODOS:
        abort(404, message="Todo {} doesn't exist".format(todo_id))

parser = reqparse.RequestParser()
parser.add_argument('task')

# shows a single todo item and lets you delete a todo item
class Todo(Resource):
    def get(self, todo_id):
        abort_if_todo_doesnt_exist(todo_id)
        return TODOS[todo_id]

    def delete(self, todo_id):
        abort_if_todo_doesnt_exist(todo_id)
        del TODOS[todo_id]
        return '', 204

    def put(self, todo_id):
        args = parser.parse_args()
        task = {'task': args['task']}
        TODOS[todo_id] = task
        return task, 201

# shows a list of all todos, and lets you POST to add new tasks
class TodoList(Resource):
    def get(self):
        return TODOS

    def post(self):
        args = parser.parse_args()
        todo_id = int(max(TODOS.keys()).lstrip('todo')) + 1
        todo_id = 'todo%i' % todo_id
        TODOS[todo_id] = {'task': args['task']}
        return TODOS[todo_id], 201

# Actually setup the Api resource routing here
api.add_resource(TodoList, '/todos')
api.add_resource(Todo, '/todos/<todo_id>')

if __name__ == '__main__':
    app.run(debug=True)
```

Ukázka kódu 1.11: Příklad použití z dokumentace Flask-RESTful [14]

vývojářů. 226 hvězd na GitHubu a malý počet přispěvatelů dává tušit, že se nejedná o příliš slavný projekt, obsahuje však mnoho zajímavých funkcí [28].

V případě RESTu jde hlavně o jednoduché prolinkování v duchu HATEOAS, které můžete vidět v ukázce 1.12. Komplexnější příklad dokumentace neobsahuje. Morepath dostává za HATEOAS dva body.

Za účelem vytvoření webové služby je potřeba použít modely; to mohou být v Morepathu objekty v paměti, abstrakce databázových tabulek pomocí ORM, či data uložená v NoSQL databázi.

```
@App.json(model=DocumentCollection)
def collection_default(self, request):
    return {
       'type': 'document_collection',
       'documents': [dict(id=doc.id, link=request.link(doc))
                     for doc in self.documents],
       'add': request.link(documents, 'add')
    }
```

Ukázka kódu 1.12: Příklad použití z dokumentace Morepathu [26]

Přístupová práva umí Morepath nastavovat na úrovni modelů či pohledů a autentizace uživatele může proběhnout na základě session [29]. O žádných speciálních metodách autentizace v případě API se dokumentace nezmiňuje, dostává tedy jeden bod.

Hodnotím Morepath jako zajímavý webový mikroframewrok, pokud uživatel touží po modelech, ale nechce použít „velký“ MVC framework jako Django. Poskytnutím mechanismů pro tvorbu RESTful API přímo v základu frameworku se řadí mezi ojedinělé Python webové frameworky.

1.10 Nefertari

Nefertari je REST API framework pro Pyramid, který používá Elasticsearch pro čtení a MongoDB nebo PostgreSQL pro zápis [11].

V Nefertari je nejprve potřeba připravit model, což je entita mapovaná na databázi, a k danému modelu vytvořit pohled, což je mapování dané entity na HTTP metody. Ukázkový model a pohled můžete vidět v ukázkách 1.13 a 1.14. Serializaci do JSONu a mapování na URL za vás obstará framework.

```
from datetime import datetime
from nefertari import engine as eng
from nefertari.engine import BaseDocument


class Story(BaseDocument):
    __tablename__ = 'stories'

    _auth_fields = [
        'id', 'updated_at', 'created_at', 'start_date',
        'due_date', 'name', 'description']
    _public_fields = ['id', 'start_date', 'due_date', 'name']

    id = eng.IdField(primary_key=True)
    updated_at = eng.DateTimeField(onupdate=datetime.utcnow)
    created_at = eng.DateTimeField(default=datetime.utcnow)

    start_date = eng.DateTimeField(default=datetime.utcnow)
    due_date = eng.DateTimeField()

    name = eng.StringField(required=True)
    description = eng.TextField()
```

Ukázka kódu 1.13: Příklad použití z dokumentace Nefertari (model) [7]

Nefertari závisí na devíti, nepřímo na osmnácti modulech, což je oproti jiným zkoumaným frameworkům opravdu hodně. Instalace obsahuje celkem 54 339 řádků kódu. Podporován je Python 2 i 3. Projekt je distribuován pod permisivní Apache licencí [3].

První commit v projektu se datuje na březen 2015, jedná se tedy, v době psaní tohoto textu, zhruba o jeden rok starý projekt. Od té doby vyšlo téměř patnáct verzí, poslední v listopadu 2015. Za projektem stojí startup Brandicted[9], kromě nich do projektu příliš mnoho lidí nepřispívá.

[9]Hlavní služba startupu Brandicted.com je v době psaní tohoto textu nedostupná. Je otázkou, zdali jde o náhodu, nebo má Nefertari nejistou budoucnost. Repozitář na GitHubu (který má 37 hvězd) se přesunul do organizace *ramses-tech*, která ale obsahuje stejné vývojáře jako původní organizace *brandicted*.

```python
from nefertari.view import BaseView
from example_api.models import Story


class StoriesView(BaseView):
    Model = Story

    def index(self):
        return self.get_collection_es()

    def show(self, **kwargs):
        return self.context

    def create(self):
        story = self.Model(**self._json_params)
        return story.save(self.request)

    def update(self, **kwargs):
        story = self.Model.get_item(
            id=kwargs.pop('story_id'), **kwargs)
        return story.update(self._json_params, self.request)

    def replace(self, **kwargs):
        return self.update(**kwargs)

    def delete(self, **kwargs):
        story = self.Model.get_item(
            id=kwargs.pop('story_id'), **kwargs)
        story.delete(self.request)

    def delete_many(self):
        es_stories = self.get_collection_es()
        stories = self.Model.filter_objects(es_stories)

        return self.Model._delete_many(stories, self.request)

    def update_many(self):
        es_stories = self.get_collection_es()
        stories = self.Model.filter_objects(es_stories)

        return self.Model._update_many(
            stories, self._json_params, self.request)
```

Ukázka kódu 1.14: Příklad použití z dokumentace Nefertari (pohled) [8]

1.10.1 HATEOAS

Dokumentace Nefertari se nezmiňuje o způsobu, jak jednotlivé zdroje prolinkovat. V části *Vize* [9] dokonce přímo říká:

> Pro nás znamená „REST API“ něco jako „HTTP metody namapované na CRUD[10] operace nad zdroji popsanými v JSONu“. Nesnažíme se o úplný HATEOAS, ani o naplnění akademického ideálu o RESTu.

Nedostává tedy žádné body.

1.10.2 Přístupová práva

Nefertari používá model autentizace z frameworku Pyramid, pomocí cookies [10], což je pro REST API nevyhovující. Přístupová práva oproti tomu umožňuje nastavit velice variabilně na úrovni jednotlivých operací a zdrojů [10]. Dostává tedy jeden bod.

Nefertari vede k tomu, že se vývojář o některé věci vůbec nemusí starat: jak přesně jsou data uložena v databázi nebo jak se mapují zdroje na URL. To může být velkou výhodou, ale i nevýhodou. Podle dokumentace se zdá, že jednotlivá výchozí chování nelze příliš ovlivnit. Nefertari jistě ušetří mnoho práce za cenu svobody volby. Absence HATEOAS a autentizace pomocí cookies můj pohled na Nefertari příliš nezlepší. Existují jistě situace, kde bude Nefertari velmi vhodná, ale není dostatečně flexibilní pro širokou škálu případů.

1.11 Ramses (rozšíření pro Nefertari)

Ramses je framework, který generuje RESTful API pomocí RAMLu [11]. Používá Pyramid a Nefertari [6].

Ramses přináší stejné funkce jako Nefertari, o které jsem psal v části 1.10. Místo kódu v Pythonu se však používá deskriptivní jazyk RAML.

[10] *Create*, *Retrieve*, *Update*, *Delete*
[11] *RESTful API Modeling Language*, postavený na YAMLu [137]

Obrázek 1.10: Logo Ramsesu [134]

Ramses přímo závisí na sedmi modulech, nepřímo pak na téměř třiceti, instalace má celkem 68 594 řádků kódu. V počtu závislostí je tak v negativním slova smyslu vítězem.

Projekt vytváří stejní autoři jako Nefertari a všechny informace o aktivitě jsou prakticky stejné. Projekt vznikl na přelomu února a března 2015, poslední z dvanácti verzí vyšla v listopadu téhož roku. Kód je distribuován pod permisivní Apache licencí [3], stejně jako Nefertari. Na rozdíl od Nefertari má na GitHubu více než dvě stovky hvězd.

Zkrácený příklad RAML souboru pro vytvoření API můžete vidět v ukázce 1.15. V odpovědi v ukázce 1.16 pak můžete vidět absenci prolinkování.

Vzhledem k tomu, že Ramses je vrstva abstrakce nad Nefertari, nebudu zde opakovat sekce o HATEOAS a přístupových právech, jelikož by byly prakticky stejné. Vytváření REST API pomocí RAML souborů je možná směr, kterým se v budoucnu lidstvo vydá, ale bojím se, že na Ramsesu je třeba ještě zapracovat. Otázkou je, jestli bude dále vyvíjen.

```
#%RAML 0.8
---
title: pizza_factory API
documentation:
    - title: pizza_factory REST API
      content: |
        Welcome to the pizza_factory API.
baseUri: http://{host}:{port}/{version}
version: v1
mediaType: application/json
protocols: [HTTP]

/cheeses:
    displayName: Collection of different cheeses
    get:
        description: Get all cheeses
    post:
        description: Create a new cheese
        body:
            application/json:
                schema: !include schemas/cheeses.json

    /{id}:
        displayName: A~particular cheese ingredient
        get:
            description: Get a particular cheese
        delete:
            description: Delete a particular cheese
        patch:
            description: Update a particular cheese

/pizzas:
    displayName: Collection of pizza styles
    get:
        description: Get all pizza styles
    post:
        description: Create a new pizza style
        body:
            application/json:
                schema: !include schemas/pizzas.json

    /{id}:
        displayName: A~particular pizza style
        get:
            description: Get a particular pizza style
        delete:
            description: Delete a particular pizza style
        patch:
            description: Update a particular pizza style

# ...
```

Ukázka kódu 1.15: Příklad použití Ramsesu [51]

```
# POST /api/pizzas name=hawaiian toppings:=[1,2] ...
{
    "data": {
        "_type": "Pizza",
        "_version": 0,
        "cheeses": [
            1
        ],
        "crust": 1,
        "crust_id": 1,
        "description": null,
        "id": 1,
        "name": "hawaiian",
        "sauce": 1,
        "sauce_id": 1,
        "self": "http://localhost:6543/api/pizzas/1",
        "toppings": [
            1,
            2
        ],
        "updated_at": null
    },
    "explanation": "",
    "id": "1",
    "message": null,
    "status_code": 201,
    "timestamp": "2015-06-05T18:47:53Z",
    "title": "Created"
}
```

Ukázka kódu 1.16: Odpověď Ramsesu [51]

1.12 Piston

Piston je, respektive spíše byl, mini-framework pro Django určený pro vytváření RESTful API [105].

Piston je interně svázán s Djangem a podle své dokumentace nabízí následující funkce [105]:

- podporuje OAuth bez nutnosti použití další knihovny,
- nevyžaduje vazbu na modely, umožňuje vytvářet nezávislé zdroje,
- komunikuje pomocí JSONu, YAMLu, Python picklu a XML (a také pomocí HATEOAS),
- jde o Python knihovnu, kterou lze snadno použít,

Obrázek 1.11: Logo Pistonu [104]

- respektuje a nabádá ke správnému využití HTTP protokolu (návratové kódy apod.),
- má zabudovanou (volitelnou) validaci vstupů (pomocí Djanga), omezování počtu požadavků v čase apod.,
- podporuje streamování s malým využitím paměti,
- „neplete se do cesty“.

Projekt vnikl již v roce 2008, tedy tři roky po zveřejnění Djanga samotného, pod záštitou Bitbucketu. V roce 2010 jej však původní autor, Jesper Nøhr, přestal vyvíjet. Vývoje se následující rok ujal Joshua Ginsberg, který ale vydal jen dvě nové verze a vývoj na začátku roku 2012 taktéž opustil. Poslední vydaná verze 0.2.3 přidává podporu pro Django 1.4, nejvyšší podporovaná verze je tedy snad 1.5[12]. S Djangem 1.6 nebo vyšším Piston nefunguje [133]. Kód rovněž obsahuje syntaxi nekompatibilní s Pythonem 3. Piston je tedy jednoznačně mrtvý projekt.

Piston byl distribuován pod BSD licencí (není však jasné, jestli jde o třípoložkovou [114] nebo dvoupoložkovou [113] variantu, projekt neuvádí celý text licence). Závisí pouze na Djangu a společně s Djangem 1.5 má 75 311 řádků kódu.

Příklad použití z dokumentace můžete vidět v ukázkách 1.17 a 1.18.

[12]Django vždy nejprve označí funkcionalitu k odebrání, v následující verzi ji označí jako zastaralou (*deprecated*) a v další verzi ji odstraní [24]. Současné podporované verze jsou 1.8 a 1.9.

```
from django.conf.urls.defaults import *
from piston.resource import Resource
from piston.authentication import HttpBasicAuthentication

from myapp.handlers import BlogPostHandler, ArbitraryDataHandler

auth = HttpBasicAuthentication(realm="My Realm")
ad = { 'authentication': auth }

blogpost_resource = Resource(handler=BlogPostHandler, **ad)
arbitrary_resource = Resource(handler=ArbitraryDataHandler, **ad)

urlpatterns += patterns('',
    url(r'^posts/(?P<post_slug>[^/]+)/$', blogpost_resource),
    url(r'^other/(?P<username>[^/]+)/(?P<data>.+)/$',
        arbitrary_resource),
)
```

Ukázka kódu 1.17: Příklad použití z dokumentace Pistonu (urls.py) [105]

1.12.1 HATEOAS

Přestože o sobě Piston říká, že komunikuje pomocí HATEOAS principu [105], v celé dokumentaci není nikde zmínka o tom, jak z jednoho zdroje linkovat zdroj jiný. Veškeré příklady místo linkování zobrazují další zdroj vnořeně, poměrně komplikovaným způsobem lze uvést alespoň ID [88]. I když by bylo možné si pro každý druh odkazu nadefinovat vlastní metodu, považuji to za zbytečně komplikované. Myslím, že použití zkraty HATEOAS je tedy v popisu tohoto frameworku poměrně neoprávněné, a dávám nula bodů.

1.12.2 Přístupová práva

Piston nabízí dva druhy autentizace: základní HTTP autentizaci jménem a heslem a OAuth 1. Další způsoby je možné doimplementovat [106]. Pro autorizaci lze použít zabudovaný mechanismus, který umožňuje část API otevřít anonymním uživatelům a část pouze přihlášeným [107]. Pro komplikovanější použití je nutné manuálně kontrolovat, který uživatel je přihlášen, a podle toho nějakou akci provést či neprovést. Piston dostává za přístupová práva tři body.

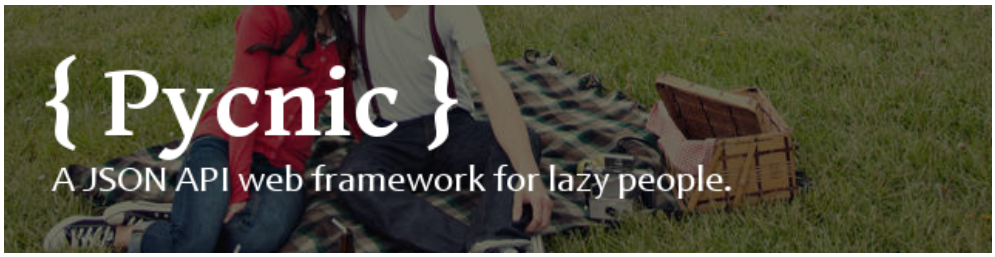
Piston je v dnešní době již nepoužitelný framework, jehož vývoj je úplně zastaven, vzhledem k tomu jej nemohu doporučit. Příliš komplikovaný způsob

```
import re

from piston.handler import BaseHandler
from piston.utils import rc, throttle

from myapp.models import Blogpost

class BlogPostHandler(BaseHandler):
    allowed_methods = ('GET', 'PUT', 'DELETE')
    fields = ('title', 'content',
              ('author', ('username', 'first_name')), 'content_size')
    exclude = ('id', re.compile(r'^private_'))
    model = Blogpost

    @classmethod
    def content_size(self, blogpost):
        return len(blogpost.content)

    def read(self, request, post_slug):
        post = Blogpost.objects.get(slug=post_slug)
        return post

    @throttle(5, 10*60) # allow 5 times in 10 minutes
    def update(self, request, post_slug):
        post = Blogpost.objects.get(slug=post_slug)

        post.title = request.PUT.get('title')
        post.save()

        return post

    def delete(self, request, post_slug):
        post = Blogpost.objects.get(slug=post_slug)

        if not request.user == post.author:
            return rc.FORBIDDEN # returns HTTP 401

        post.delete()

        return rc.DELETED # returns HTTP 204

class ArbitraryDataHandler(BaseHandler):
    methods_allowed = ('GET',)

    def read(self, request, username, data):
        user = User.objects.get(username=username)

        return { 'user': user, 'data_length': len(data) }
```

Ukázka kódu 1.18: Příklad použití z dokumentace Pistonu (handlers.py) [105]

linkování mezi zdroji (je-li vůbec nějaký) se k tvorbě RESTful API také příliš nehodí.

1.13 Pycnic



Obrázek 1.12: Logo Pycnicu [109]

Pycnic je malý, rychlý a jednoduchý framework na tvorbu JSON API. Umí routovat, komunikovat v JSONu a pracovat s cookies [110]. Pycnic neobsahuje žádné pokročilé funkce, ale ani je obsahovat nechce, jedná se opravdu o minimalistickou knihovnu.

Pycnic je poměrně mladý projekt, který vnikl v listopadu 2015. Jeho autorem je jednotlivec Aaron M., do kódu kromě něj nikdo nepřispěl[13]. Několik málo desítek hvězd na GitHubu také nasvědčuje tomu, že se zatím nejedná o příliš známý projekt.

Pycnic závisí jen na standardní knihovně Pythonu. Instalace zabírá pouze 76 KiB a kód obsahuje pouhých 226 řádek, což je pro představu zhruba desetkrát více než kód v ukázce 1.19, ve které najdete příklad použití z dokumentace.

Přestože Pycnic neobsahuje přímou podporu autentizace, v dokumentaci poskytuje komplexní příklad, jak autentizaci implementovat [112]. Kvůli jeho obsáhlosti jej zde neuvádím.

Vzhledem k nízkoúrovnosti frameworku nedostává Pycnic v žádném z bodovaných kritérií body.

[13]Pár jednotlivců přispělo do benchmarku, o kterém bude řeč dále, nikoliv však do samotného kódu Pycnicu.

```
from pycnic.core import Handler, WSGI
from pycnic.errors import HTTP_400

class UsersHandler(Handler):

    def post(self):

        if not self.request.data.get("username"):
            raise HTTP_400("Yo dawg, you need to provide a username")

        return {
            "username":self.request.data["username"],
            "authID":self.request.cookies.get("auth_id"),
            "yourIp":self.request.ip,
            "rawBody":self.request.body,
            "method":self.request.method,
            "json":self.request.data,
            "XForward":self.request.environ["HTTP_X_FORWARDED_FOR"]
        }

class app(WSGI):
    routes = [ ("/user", UserHandler()) ]
```

Ukázka kódu 1.19: Příklad použití z dokumentace Pycnicu [111]

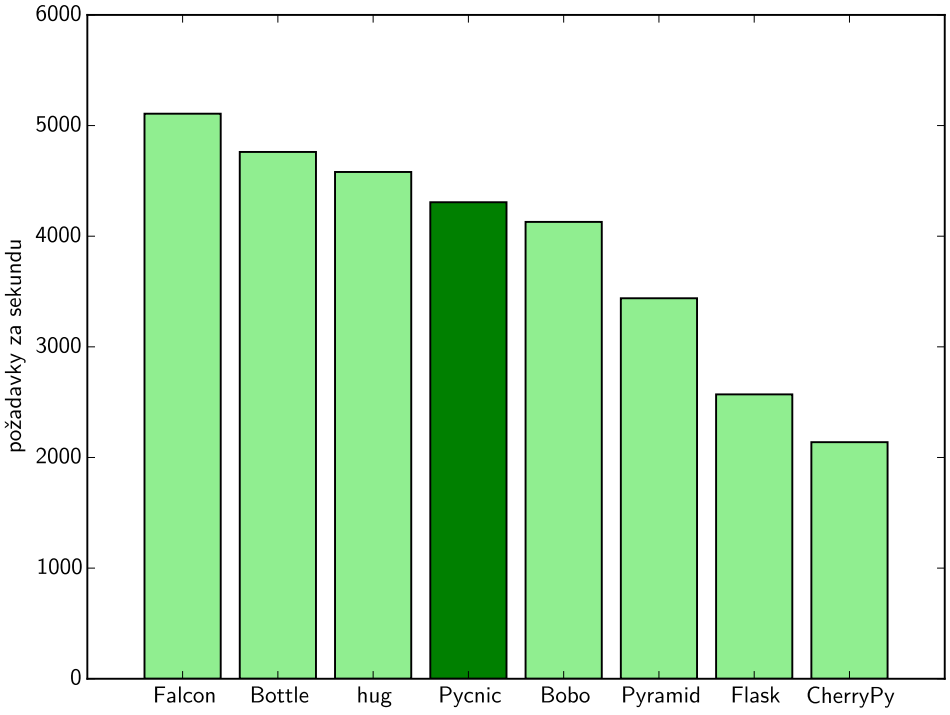
1.13.1 Benchmark

Součástí repozitáře na GitHubu je i jednoduchý benchmark, který měří, kolik požadavků za sekundu jednotlivé webové frameworky zvládnou. Je měřena jednoduchá aplikace, která vrací na adrese `/json` zprávu zakódovanou v JSONu [108]. Přestože se tato diplomová práce nezabývá webovými frameworky obecně, rozhodl jsem se měření provést. Ve výsledcích je kromě Pycnicu i Falcon a hug, které jsem zkoumal v části 1.5, respektive 1.6. Výsledky můžete vidět v grafu na obrázku 1.13.

1.14 Python REST API framework

Python REST API framework (zkráceně PRAF) je sada nástrojů postavená na Werkzeugu, pro snadnou tvorbu RESTful API pomocí MVC architektury [43]. Mezi hlavní funkce patří [43]:

- stránkování,



Obrázek 1.13: Pycnic: Výsledky benchmarku

- autentizace,
- autorizace,
- filtry,
- částečné odpovědi,
- řízení chyb,
- validátory dat,
- formátovače dat.

PRAF obsahuje několik součástí, které je potřeba využít k tvorbě API [44]:

- **datastore** je třída, která nějakým způsobem obstarává data, implicitně může využít buďto SQLite nebo reprezentaci v paměti, pro cokoli jiného musíte implementovat vlastní třídu podle daného rozhraní;
- **modely** slouží k popsání jednotlivých typů dat v *datastore*;
- **controller** obsluhuje jeden resource, ve kterém se přistupuje k datům z jednoho modelu v *datastore*;
- **pohledy** pak definují, jakým způsobem budou data prezentována.

Konkrétní příklad můžete vidět v ukázce 1.20. Dokumentace obsahuje také komplexnější příklad včetně obsáhlého tutoriálu, jak jej vytvořit [40]. Kromě tutoriálu je však dokumentace velmi stručná a místy se zdá, že možnosti PRAF příliš nepřesahují rozsah uvedeného příkladu.

Python REST API framework je distribuován pod MIT licencí [115]. Za projektem stojí jednotlivec Yohann Gabory, s minimálním přispěním od dalších vývojářů. PRAF vznikl v roce 2013 a od té doby vyšly pouze čtyři verze. Vývoj není příliš aktivní, v posledních dvou letech přibylo jen několik jednotek commitů.

Instalace přímo závisí na dvou knihovnách včetně Werkzeugu, celkem pak nepřímo na třech. Včetně závislostí má 15 988 řádků kódu. Na GitHubu má projekt pouze čtyři hvězdy a z PyPI byl za poslední měsíc stažen jen o něco málo více než dvousetkrát. Projekt neobsahuje informaci o tom, jestli podporuje Python 3, ale obsahuje minimálně jeden řádek kódu napsaný v nekompatibilní syntaxi, z čehož soudím, že Python 3 nepodporuje.

1.14.1 HATEOAS

Dokumentace v tutoriálu uvádí postup, jak prolinkovat jednotlivé zdroje mezi sebou [41][39]. Framework sám tuto funkci neobsahuje, ale ukazuje příklad formátovače, který je znovupoužitelný v celém projektu a zajistí, aby všechny cizí klíče byly reprezentovány pomocí odkazu. Můžete jej vidět v ukázce 1.21. Za možnost nějak prolinkovat zdroje dostává Python REST API framework jeden bod.

1.14.2 Přístupová práva

Tutoriál opět uvádí postup [42], jak implementovat autentizaci, v tomto konkrétním případě API klíčem předaným pomocí GET parametru zakódovaným v URL. Pokud chcete, můžete si samozřejmě implementovat způsob vlastní.

V případě autorizace nabízí PRAF pouze možnost zpřístupnit daný zdroj všem autentizovaným požadavkům [42], implementace komplexnějších přístupových práv je opět možná. Proto dávám dva body.

```
from rest_api_framework import models
from rest_api_framework.datastore import SQLiteDataStore
from rest_api_framework.views import JsonResponse
from rest_api_framework.controllers import Controller
from rest_api_framework.datastore.validators import UniqueTogether
from rest_api_framework.pagination import Pagination


class UserModel(models.Model):
    """Define how to handle and validate your data."""
    fields = [models.StringField(name="first_name", required=True),
              models.StringField(name="last_name", required=True),
              models.PkField(name="id", required=True)
              ]


def remove_id(response, obj):
    """Do not show the id in the response."""
    obj.pop(response.model.pk_field.name)
    return obj


class UserEndPoint(Controller):
    ressource = {
        "ressource_name": "users",
        "ressource": {"name": "adress_book.db", "table": "users"},
        "model": UserModel,
        "datastore": SQLiteDataStore,
        "options": {"validators": [UniqueTogether("first_name",
                                                  "last_name")]}
        }

    controller = {
        "list_verbs": ["GET", "POST"],
        "unique_verbs": ["GET", "PUT", "DELETE"],
        "options": {"pagination": Pagination(20)}
        }

    view = {"response_class": JsonResponse,
            "options": {"formaters": ["add_ressource_uri",
                                      remove_id]}}


if __name__ == '__main__':
    from werkzeug.serving import run_simple
    from rest_api_framework.controllers import WSGIDispatcher
    app = WSGIDispatcher([UserEndPoint])
    run_simple('127.0.0.1', 5000, app, use_debugger=True,
               use_reloader=True)
```

Ukázka kódu 1.20: Příklad použití z dokumentace PRAF [43]

```
def format_foreign_key(response, obj):
    from rest_api_framework.models.fields import ForeignKeyField
    for f in response.model.get_fields():
        if isinstance(f, ForeignKeyField):
            obj[f.name] = \
                "/{0}/{1}/".format(f.options["foreign"]["table"],
                                   obj[f.name])
    return obj
```

Ukázka kódu 1.21: PRAF: Formátovač pro prolinkování dat [39]

Python REST API framework nabízí určitou strukturu, jak REST API v Pythonu budovat, nenabízí ale velký výběr stavebních kamenů. Předpokládá se, že programátor si je dobuduje sám, což nepovažuji nutně za špatnou věc. Je však třeba vytknout v současnosti zpomalený vývoj projektu a především absenci podpory pro Python 3. Nízká oblíbenost projektu může být důsledkem těchto problémů.

1.15 RESTArt

RESTArt je knihovna pro tvorbu REST API. Je inspirovaná Flaskem, ale nezávisí na něm [120]. Závisí na knihovně Werkzeug a dalších třech modulech, nepřímo tak celkem na pěti. Instalace má 20 105 řádků kódu.

Projekt vznikl v květnu 2015 a od té doby vyšlo šest verzí. Je stále aktivně vyvíjen, avšak pouze autorem, jednotlivcem Luem Pengem. Je distribuován pod MIT licencí [115]. Deset hvězd na GitHubu a žádné zapojení jiných vývojářů dává tušit, že nejde o všeobecně známý a používaný projekt.

RESTArt umožňuje vytvářet pro jednotlivé zdroje třídy s metodami korespondujícími s HTTP metodami pro REST API, příklad z dokumentace můžete vidět v ukázce 1.22. Je možné vytvářet tzv. middleware třídy, které mohou předzpracovávat požadavky a ovlivňovat obsah odpovědí. Tímto způsobem lze naimplementovat i autentizaci a autorizaci, RESTArt samotný žádné možnosti v této oblasti nepřináší. Pomocí middleware třídy by mělo jít i nějak automaticky prolinkovat jednotlivé zdroje, dokumentace se o této možnosti ale nijak nezmiňuje. Dávám tedy u obou kritérií jeden bod.

```
from restart.api import RESTArt
from restart.resource import Resource

api = RESTArt()

@api.route(methods=['GET'])
class Greeting(Resource):
    name = 'greeting'

    def read(self, request):
        return {'hello': 'world'}
```

Ukázka kódu 1.22: Příklad použití z dokumentace RESTArtu [117]

RESTArt obsahuje i kód REST klienta, pro účely testování [118].

Kromě Werkzeugu lze požít i jiné webové frameworky, například Flask nebo Falcon [119], případně lze napsat adaptér na jakýkoliv jiný.

RESTArt oproti jiným frameworkům nepřináší nic nového, největší výhodou je možnost vybrat si vlastní webový framework, například pokud již existující součást aplikace nějaký používá.

1.16 restless

Restless je miniframework pro tvorbu REST API. Podporuje Django, Flask, Pyramid, Tornado a Itty[14], ale měl by fungovat s jakýmkoliv webovým frameworkem v Pythonu [91].

Hlavní myšlenkou restlessu je udělat věci jednoduše a příliš je nekomplikovat, mezi hlavní výhody patří [91]:

- malý a rychlý kód,
- výchozí výstup v JSONu,
- koncept RESTful,
- podpora Pythonu 3.3+ (i staršího 2.6+),
- flexibilita.

[14] Itty je webový framework od autora restlessu.

Restless v dokumentaci rovnou uvádí, že nebude podporovat automatickou integraci ORM, XML, autorizaci ani HATEOAS [91][92]. Dostává tedy u obou bodovaných kritérií nula bodů.

Za projektem stojí jednotlivec Daniel Lindsley, který je mj. autorem frameworku Tastypie, o kterém budu mluvit v části 1.19 na straně 76. Dokumentace restlessu projekt Tastypie často zmiňuje a zdůrazňuje, že restless vznikl poučením se z chyb při tvorbě Tastypie. Hlavní chybou bylo pokoušet se o vytvoření příliš „všemocného“ frameworku, restless jde tedy opačnou cestou a většinu rozhodnutí nechává na uživateli [92].

Restless, na rozdíl od Tastypie, není vázán přímo na Django, ale webový framework si můžete zvolit. Přímo v kódu existují třídy pro frameworky Django, Flask, Pyramid, Tornado a Itty, od kterých stačí dědit. Pro jiný framework si takovou třídu můžete samozřejmě dopsat sami. V případě změny frameworku by mělo stačit třídu vyměnit. Příklad z dokumentace s použitím třídy pro Django můžete vidět v ukázce 1.23.

Restless závisí pouze na knihovně six, kvůli zpětné kompatibilitě s Pythonem 2. Instalace tak zabírá pouze čtvrt mebibajtu a obsahuje 1 140 řádek kódu, ale tato informace je zavádějící, protože restless ještě vyžaduje nějaký webový framework, samostatně nefunguje.

Projekt vznikl v lednu 2014, autor jej aktivně vyvíjel do srpna toho roku, od té doby prakticky pouze přijímá cizí příspěvky, kterých však není příliš mnoho, poslední byl přijat v létě 2015. Od té doby se kupí další a další, čekající na schválení, které možná nikdy nepřijde. Zatím poslední verze, v pořadí sedmá, vyšla v srpnu 2014. Daniel Lindsley se restlessu zjevně nevěnuje. Uživatelé však nadále hlásí chyby a snaží se přispět svým kódem. Více než pět stovek hvězd na GitHubu u projektu, který byl aktivně vyvíjen půl roku, svědčí o tom, že měl potenciál.

1.17 ripozo

Ripozo je nástroj pro vytváření RESTful/HATEOAS API. Poskytuje silné, jednoduché, plně kvalifikované odkazy mezi zdroji; podporuje více protokolů

```python
from django.contrib.auth.models import User

from restless.dj import DjangoResource
from restless.preparers import FieldsPreparer

from posts.models import Post


class PostResource(DjangoResource):
    # Controls what data is included in the serialized output.
    preparer = FieldsPreparer(fields={
        'id': 'id',
        'title': 'title',
        'author': 'user.username',
        'body': 'content',
        'posted_on': 'posted_on',
    })

    # GET /
    def list(self):
        return Post.objects.all()

    # GET /pk/
    def detail(self, pk):
        return Post.objects.get(id=pk)

    # POST /
    def create(self):
        return Post.objects.create(
            title=self.data['title'],
            user=User.objects.get(username=self.data['author']),
            content=self.data['body']
        )

    # PUT /pk/
    def update(self, pk):
        try:
            post = Post.objects.get(id=pk)
        except Post.DoesNotExist:
            post = Post()

        post.title = self.data['title']
        post.user = User.objects.get(username=self.data['author'])
        post.content = self.data['body']
        post.save()
        return post

    # DELETE /pk/
    def delete(self, pk):
        Post.objects.get(id=pk).delete()
```

Ukázka kódu 1.23: Příklad použití s Djangem z dokumentace restlessu [96]

Obrázek 1.14: Logo ripoza [97]

(Siren a HAL). Ripozo je velmi flexibilní, dá se použít s libovolným webovým frameworkem v Pythonu a libovolnou databází. [98]

Základní příklad použití typu *hello world* můžete vidět v ukázce 1.24. V ukázce je vynecháno napojení na webový framework. Můžete využít existujících knihoven pro napojení na Django a Flask, či si napsat vlastní třídu pro napojení na jiný webový framework [98]. Příklad, který přes REST API nabízí kompletní CRUD+L[15], pak můžete vidět v ukázce 1.25. Pokud chcete nabízet jen některé akce, můžete použít mixiny[16].

Projekt vznikl v roce 2014, od té doby vyšlo více než třicet verzí, nejnovější asi měsíc před psaním tohoto textu. Za projektem stojí firma Vertical Knowledge, vyvíjí jej hlavně Tim Martin, ale přispěli i jednotlivci nesouvisející s touto firmou. Ripozo je distribuováno pod copyleftovou licencí GNU General Public License verze 2 [34] nebo vyšší, čímž se odlišuje od naprosté většiny ostatních zde diskutovaných frameworků.

Instalace závisí jen na knihovně six, kvůli kompatibilitě s oběma verzemi Pythonu, zabírá pouze půl mebibajtu a obsahuje 2 130 řádků kódu. Vzhledem k tomu, že instalace samotného ripoza je nepoužitelná, jelikož je potřeba použít nějaký webový framework, je tato informace zavádějící. Například po

[15] *Create*, *Retrieve*, *Update*, *Delete* a *List* [52]

[16] Mixin je třída, kterou v Pythonu použijete jako rodiče nebo jednoho z rodičů, abyste rozšířili funkcionalitu. Ničím se neliší od jiné třídy, termín mixin se používá pouze na odlišení významu. V tomto konkrétním případě tak například můžete použít mixiny *restmixins.Create* a *restmixins.List* pro poskytnutí akcí pouze pro čtení. [89]

```
from ripozo import apimethod, adapters, ResourceBase
# import the dispatcher class for your preferred webframework

class MyResource(ResourceBase):
    @apimethod(methods=['GET'])
    def say_hello(cls, request):
        return cls(properties=dict(hello='world'))

# initialize the dispatcher for your framework
# e.g. dispatcher = FlaskDispatcher(app)
dispatcher.register_adapters(adapters.SirenAdapter,
                             adapters.HalAdapter)
dispatcher.register_resources(MyResource)
```

Ukázka kódu 1.24: Příklad použití z dokumentace ripoza [98]

```
from ripozo import restmixins
from fake_ripozo_extension import Manager
# An ORM model for example a sqlalchemy or Django model:
from myapp.models import MyModel

class MyManager(Manager):
    fields = ('id', 'field1', 'field2',)
    model = MyModel

class MyResource(restmixins.CRUDL):
    manager = MyManager()
    pks = ('id',)

# Create your dispatcher and register the resource...
```

Ukázka kódu 1.25: Příklad použití z dokumentace ripoza (CRUD+L) [98]

instalaci modulů na spolupráci s Flaskem a SQLAlchemy je již závislostí sedm (nepočítaje tři vlastní moduly `ripozo`, `flask-ripozo` a `ripozo-sqlalchemy`) a instalace zabírá 14 MiB.

1.17.1 HATEOAS

Již v úvodu jsem zmínil, že ripozo umožňuje jednoduše vytvářet linky mezi zdroji ve stylu HATEOAS a také že ripozo podporuje Siren [136] a HAL [79]. Získává tedy tři body. Představu o vytváření odkazů získáte nejlépe z ukázky 1.26.

```
from ripozo import restmixins, Relationship

class MyResource(restmixins.CRUDL):
    manager = MyManager()
    pks = ('id',)
    _relationships = [Relationship('related',
                                   relation='RelatedResource')]

class RelatedResource(restmixins.CRUDL)
    manager = RelatedManager()
    pks = ('id',)
```

Ukázka kódu 1.26: Příklad použití z dokumentace ripoza (linkování) [98]

1.17.2 Přístupová práva

Ripozo nenabízí přímo žádnou funkcionalitu pro autentizaci či autorizaci. Obsahuje však možnost předzpracovávat požadavky pomocí funkcí. V dokumentaci se říká, že tímto způsobem můžete například zpřístupnit zdroj pouze autentizovaným uživatelům [99]. Ripozo zde tedy dostává dva body.

Ripozo je framework, který umožňuje vytvářet RESTful HATEOS API pomocí Siren a HAL, prakticky bez práce. Možnost výběru vlastního frameworku i databáze je velké plus. Nevýhodou může v některých případech být copyleftová licence[17].

1.18 sandman2

Sandman2 je nástroj, který automaticky poskytuje REST API nad SQL databází [83]. Stačí mu předat údaje k databázi a je možné jej hned začít používat. Je možné jej spustit jako samostatnou službu nebo jej integrovat do vlastní aplikace. Ve výchozí konfiguraci zpřístupní celou databázi, toto chování lze ale změnit a zpřístupnit jen část, případně u některých tabulek povolit jen některé operace. Příklad úpravy můžete vidět v ukázce 1.27.

Sandman2 používá Flask a SQLAlchemy, takže podporuje širokou škálu SQL databází včetně MySQL, PostgreSQL, Oracle, Microsoft SQL Serveru a SQLite [85]. Závisí přímo na čtyřech a nepřímo na dvanácti modulech, instalace má

[17] Richard M. Stallman by určitě namítal, že to je výhodou.

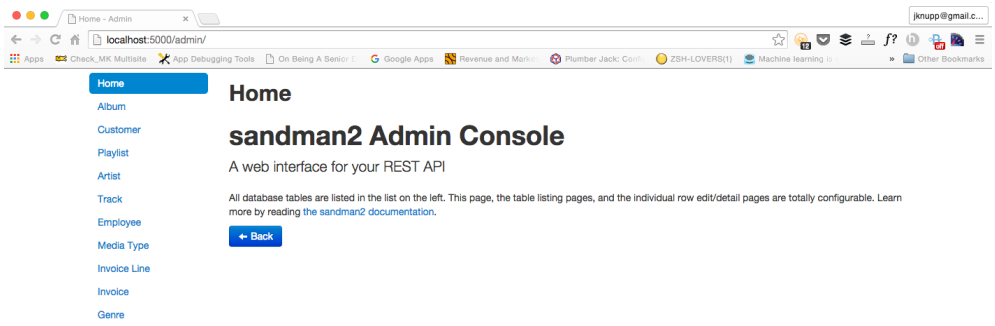
```
class Style(Model):
    """Model mapped to the "Genre" table

    Has a custom endpoint ("styles" rather than the default, "genres").
    Only supports HTTP methods specified.
    Has a custom validator for the GET method.

    """

    __tablename__ = 'Genre'
    __endpoint__ = 'styles'
    __methods__ = ('GET', 'DELETE')
    __top_level_json_name__ = 'Genres'

    @staticmethod
    def validate_GET(resource=None):
        """Return False if the request should not be processed.

        :param resource: resource related to current request
        :type resource: :class:`sandman.model.Model` or None

        """

        if isinstance(resource, list):
            return True
        elif resource and resource.GenreId == 1:
            return False
        return True
```

Ukázka kódu 1.27: sandman: Příklad úpravy chování [84]

celkem 82 207 řádků kódu a tím – v negativním slova smyslu – předčí všechny ostatní frameworky.

Kromě samotného REST API vytvoří sandman2 i webové rozhraní, kde je možné s daty manipulovat. Můžete jej vidět na obrázku 1.15.

Sandman2 je pokračovatelem projektu sandman [81], který je momentálně opuštěný a obsahuje informaci o tom, že by uživatelé měli používat druhou verzi [84]. Sandman2 ale nemá tak obsáhlou dokumentaci jako sandman. Původní sandman vznikl v roce 2013 a vyšlo celkem 45 verzí, sandman2 je zde od roku 2014, verzí vyšlo sedm, poslední v lednu 2016. Za sandmenem stojí jednotlivec Jeff Knupp, do projektu přispělo několik jednotek dalších přispěvatelů. Na GitHubu má sandman2 pouze 128 hvězd, ale sandman jich má přes dva tisíce.



Obrázek 1.15: sandman2: Webové rozhraní [82]

1.18.1 HATEOAS

Sandman automaticky prezentuje SQL sloupce typu cizí klíč jako odkazy [84]. Předpokládám, že sandman2 to dělá stejně, ale tuto informaci jsem nikde nenašel potvrzenou. Dávám zde tedy dva body na základě vlastnosti z první verze sandmanu.

1.18.2 Přístupová práva

Dokumentace k původnímu sandmanu uvádí, že je možné použít HTTP autentizaci jménem a heslem a na základě ní zpřístupnit celé API pouze přihlášeným uživatelům [80]. Jiné zabudované možnosti podporovány zatím nejsou. Je ale možné napsat speciální funkci, která předzpracovává všechny požadavky a v této implementovat jiný způsob autentizace a autorizace.

Sandman2 informace o přístupových právech v dokumentaci neobsahuje, tato funkcionalita zde ještě není; dostává tedy nula bodů.

Sandman2 je nástroj, který mapuje REST rozhraní k SQL databázi a dělá to dobře. Nepříjemný je přerod projektu sandman do projektu sandman2, na který doplácí hlavně dokumentace, což se doufám časem zlepší.

1.19 Tastypie

Tastypie je framework pro vytváření API k webovým službám určený pro Django. Poskytuje abstrakci pro vytváření rozhraní ve stylu REST. Zjednodušuje

Obrázek 1.16: Logo Tastypie [90]

zveřejňování modelů a umožňuje vybrat, které části budou přes API přístupné. Kromě ORM dat je možné použít i jiné zdroje. [90]

Mezi hlavní funkce patří [90]:

- podpora HTTP metod GET, POST, PUT, DELETE a PATCH,
- rozumné chování ve výchozím stavu,
- rozšiřitelnost,
- podpora různých serializačních formátů (JSON/XML/YAML/bplist),
- HATEOAS,
- dobré testy a dokumentace.

Projekt vznikl již v roce 2010, od té doby vyšlo více než dvacet verzí, poslední necelý měsíc před psaním tohoto textu. Autorem projektu je jednotlivec Daniel Lindsley, který se projektu již příliš nevěnuje, v současnosti se o něj stará Seán Hayes, přispělo celkem více než 150 přispěvatelů. Projekt je distribuován pod permisivní BSD licencí [114].

Pokud si vystačíte s JSON serializací, závisí Tastypie přímo na třech, nepřímo na čtyřech modulech a zabírá 41 MiB. Pro použití XML, YAML nebo bplistu je potřeba nainstalovat další moduly. Tastypie funguje na Pythonu 2 i 3 a podporuje poslední verze Djanga.

1.19.1 HATEOAS

Příklad použití můžete vidět v ukázkách 1.28 a 1.29. Přímo v tomto příkladu vznikne prolinkování mezi zdroji pomocí URL. Tastypie dostává tři body.

```python
from django.contrib.auth.models import User
from tastypie import fields
from tastypie.resources import ModelResource
from myapp.models import Entry


class UserResource(ModelResource):
    class Meta:
        queryset = User.objects.all()
        resource_name = 'user'


class EntryResource(ModelResource):
    user = fields.ForeignKey(UserResource, 'user')

    class Meta:
        queryset = Entry.objects.all()
        resource_name = 'entry'
```

Ukázka kódu 1.28: Příklad použití z dokumentace Tastypie (api.py) [93]

```python
from django.conf.urls import url, include
from tastypie.api import Api
from myapp.api import EntryResource, UserResource

v1_api = Api(api_name='v1')
v1_api.register(UserResource())
v1_api.register(EntryResource())

urlpatterns = [
    # The normal jazz here...
    url(r'^blog/', include('myapp.urls')),
    url(r'^api/', include(v1_api.urls)),
]
```

Ukázka kódu 1.29: Příklad použití z dokumentace Tastypie (urls.py) [93]

1.19.2 Přístupová práva

Tastypie umožňuje autentizaci přes HTTP jméno a heslo, pomocí API klíče, session a OAuth 1; lze si také dopsat vlastní způsob [94]. Existují moduly třetích stran přidávající podporu OAuth 2 [122]. Na úrovni zdrojů lze pak nastavit, jaký autorizační model se použije, k dispozici je buďto varianta povolit všechno, nebo povolit jen číst, případně lze použít propracovanější systém Djanga, který mapuje práva uživatele na konkrétní objekty; implementace vlastní logiky je také možná [95]. I zde tedy Tastypie získává tři body.

Tastypie se jeví jako velmi použitelný framework pro Django. Důstojně konkuruje Django REST frameworku, o kterém jsem psal v části 1.3 na straně 35. Případná volba mezi těmito dvěma frameworky hodně závisí na konkrétních potřebách a preferencích uživatele.

1.20 Srovnání

V tabulce 1.1 najdete udělené body za podporu HATEOASu a řízení přístupových práv.

V tabulce 1.2 najdete srovnání měřitelných kritérií. Jednotlivé sloupce mají zjednodušené názvy, ale jejich funkce odpovídá popisu v části 1.1 na straně 25. Tučně jsou označeny hodnoty, které v daném sloupci dominují.

V tabulce 1.3 pak najdete informační přehled o zkoumaných frameworcích: webový framework, URL domovské stránky a číslo zkoumané verze.

Pro implementaci si vybírám frameworky Eve a ripozo, na základě vysokého hodnocení v oblasti HATEOAS i přístupových práv.

Vysoké hodnocení získaly i Django REST framework a Tastypie. Jelikož oba tyto frameworky rozšiřují Django, implementace v nich by byla velmi podobná. Vybírám si proto pouze Django REST framework, který je podle indikátorů ze všech zkoumaných frameworků nejoblíbenější.

Navíc si vybírám sandman2, který nemá tak dobré hodnocení, ale slibuje automatické vytvoření API. Rád bych ze stejného důvodu zkoumal i Ramses, ale ten není možné použít s daty v MySQL databázi.

Tabulka 1.1: Bodové ohodnocení

Framework	HATEOAS	Přístupová práva
Cornice		•
Django REST fr.	•••	•••
Eve	•••	•••
Falcon		
hug		••
Flask API		
Flask-RESTful		
Morepath	••	•
Nefertari		•
Ramses		•
Piston		•••
Pycnic		
Python REST API fr.	•	••
RESTArt	•	•
restless		
ripozo	•••	••
sandman2	••	
Tastypie	•••	•••

Tabulka 1.2: Srovnání měřitelných kritérií

Framework	Druh licence	Webový fr.	MiB	Řádky	Ř. včetně	Závisl.	Py	GitHub	PyPI
Cornice	LGPL	lightweight	12	1 198	24 625	2/9	**3+2**	270	10 903
Django REST fr.	**permisivní**	MVC	43	7 057	79 854	1/1	**3+2**	**5 606**	**316 772**
Eve	**permisivní**	lightweight	10	3 440	35 009	10/10	**3+2**	3 121	7 480
Falcon	**permisivní**	**standalone**	0,9	2 352	3 034	2/2	**3+2**	2 756	51 071
hug	**permisivní**	lightweight	4	2 367	16 545	2/4	3	3 020	7 674
Flask API	**permisivní**	lightweight	6	620	20 938	1/5	**3+2**	688	7 594
Flask-RESTful	**permisivní**	lightweight	9	967	27 718	4/9	**3+2**	1 920	172 775
Morepath	**permisivní**	**standalone**	4	1 940	9 156	4/5	**3+2**	226	1 594
Nefertari	**permisivní**	lightweight	16	2 905	54 339	9/18	**3+2**	37	812
Ramses	**permisivní**	lightweight	19	1 067	68 594	7/29	**3+2**	216	661
Piston	**permisivní**	MVC	49	1 935	75 311	1/1	2	–	2 419
Pycnic	**permisivní**	**standalone**	**0,08**	**226**	**226**	**0/0**	**3+2**	33	304
Python REST API fr.	**permisivní**	lightweight	3	954	15 988	2/3	2	4	248
RESTArt	**permisivní**	lightweight	3	798	20 105	4/5	**3+2**	10	829
restless	**permisivní**	lightw./MVC	$\geq$ 0,25	528	$\geq$ 1 140	$\geq$ 1/1	**3+2**	520	7 909
ripozo	copyleft	lightw./MVC	$\geq$ 0,5	1 518	$\geq$ 2 130	$\geq$ 1/1	**3+2**	151	2 411
sandman2	**permisivní**	lightweight	23	446	82 207	4/12	**3+2**	128	625
Tastypie	**permisivní**	MVC	41	3 292	80 139	3/4	**3+2**	2 940	28 966

Tabulka 1.3: Informace o frameworcích

Framework	Webový fr.	Webová stránka	Zk. verze
Cornice	Pyramid	`https://cornice.readthedocs.org/`	1.2.1
Django REST fr.	Django	`http://www.django-rest-framework.org/`	3.3.3
Eve	Flask	`http://python-eve.org/`	0.6.3
Falcon	–	`http://falconframework.org/`	0.3.0
hug	Falcon	`http://www.hug.rest/`	2.0.6
Flask API	Flask	`http://www.flaskapi.org/`	0.6.5
Flask-RESTful	Flask	`https://flask-restful.readthedocs.org/`	0.3.5
Morepath	–	`https://morepath.readthedocs.org/`	0.13
Nefertari	Pyramid	`https://nefertari.readthedocs.org/`	0.6.1
Ramses	Pyramid	`http://ramses.tech/`	0.5.1
Piston	Django	`https://bitbucket.org/jespern/django-piston/`	0.2.3
Pycnic	–	`http://pycnic.nullism.com/`	0.0.5
Python REST API fr.	Werkzeug	`https://python-rest-framework.readthedocs.org/`	1.3
RESTArt	Werkzeug	`https://restart.readthedocs.org/`	0.1.3
restless	*volitelný*	`https://restless.readthedocs.org/`	2.0.1
ripozo	*volitelný*	`https://ripozo.readthedocs.org/`	1.3.0
sandman2	Flask	`http://pythonhosted.org/sandman2/`	0.0.7
Tastypie	Django	`http://tastypieapi.org/`	0.13.3

Kapitola 2

Návrh API pro rozvrhová data ÚTVS ČVUT

2.1 Poskytnuté databázové pohledy

V MySQL databázi Ústavu tělesné výchovy a sportu ČVUT v Praze jsem dostal k dispozici sadu SQL pohledů, která zpřístupňují data vhodná pro sestavení rozvrhů. Nad těmito pohledy budu navrhovat RESTful službu. V této části nejprve čtenáře seznámím se strukturou dat. Informace čerpám z wiki FIT ČVUT v Praze provozované Oddělením pro rozvoj [76].

2.1.1 Destinace

Destinace pro výcvikové kurzy (vícedenní kurzy mimo areál školy). Struktura je znázorněna v tabulce 2.1.

Tabulka 2.1: Struktura pohledu v_destination

Název sloupce	Datový typ	Popis
id_destination	smallint(10)	primární klíč
name	varchar(50)	název
url	varchar(250)	URL stránky na utvs.cvut.cz

2.1.2 Haly

Sportoviště ÚTVS, ve kterých probíhá výuka. Struktura je znázorněna v tabulce 2.2.

Tabulka 2.2: Struktura pohledu v_hall

Název sloupce	Datový typ	Popis
id_hall	smallint(6)	primární klíč
name	varchar(50)	název
url	varchar(250)	URL stránky na utvs.cvut.cz

2.1.3 Vyučující

Vyučující jednotlivých kurzů ÚTVS. Struktura je znázorněna v tabulce 2.3.

Tabulka 2.3: Struktura pohledu v_lectors

Název sloupce	Datový typ	Popis
id_lector	tinyint(10)	primární klíč
title_before	varchar(50)	tituly před jménem
name	varchar(50)	křestní jméno
surname	varchar(50)	příjmení
title_behind	varchar(50)	tituly za jménem
pers_number	varchar(20)	osobní číslo (peridno v KOS)
url	varchar(250)	URL stránky na utvs.cvut.cz

2.1.4 Sporty

Tabulka sportů, které se na ÚTVS praktikují. Struktura je znázorněna v tabulce 2.4.

Tabulka 2.4: Struktura pohledu v_sports

Název sloupce	Datový typ	Popis
id_sport	smallint(10)	primární klíč
short	varchar(50)	kód (tříznaková zkratka)
sport	varchar(50)	název
description	text	popis

2.1.5 Zápisy studentů

V této tabulce, nepřesně nazvané jako studenti, se eviduje zápis studenta na konkrétní předmět v daném semestru. V jednom semestru zde student může mít i více záznamů. Záznamy se po několika letech promazávají. Struktura je znázorněna v tabulce 2.5.

Semestr je ve formátu `YYYY/ZZ_S`, kde `YYYY/ZZ` značí akademický rok (např. `2012/13`) a `S` období semestru (1 – zimní; 2 – letní).

Tabulka 2.5: Struktura pohledu v_students

Název sloupce	Datový typ	Popis
id_student	int(11)	primární klíč
personal_number	int(11)	osobní číslo (peridno v KOS)
kos_kod	varchar(20)	kód zapsaného předmětu TV v KOS
utvs	int(11)	ID zapsaného předmětu ÚTVS (v_subjects.id_subjects)
semester	varchar(10)	semestr zápisu
registration_date	timestamp	datum a čas zápisu
tour	int(0)	příznak udávající, zda je zapsaný předmět kurz
kos_code	int(0)	příznak udávající, zda kos_kod obsahuje skutečný kód z KOS

2.1.6 Předměty ÚTVS

Předmětem je zde myšlena konkrétní instance vyučovaného sportu, v daný den a hodinu. Pokud bychom chtěli najít paralelu se systémem KOS, tak tato entita představuje sloučeninu instance předmětu, její paralelky a rozvrhového lístku. Struktura je znázorněna v tabulce 2.6.

Všimněte si, že některé číselné údaje jsou uloženy textově.

Tabulka 2.6: Struktura pohledu v_subjects

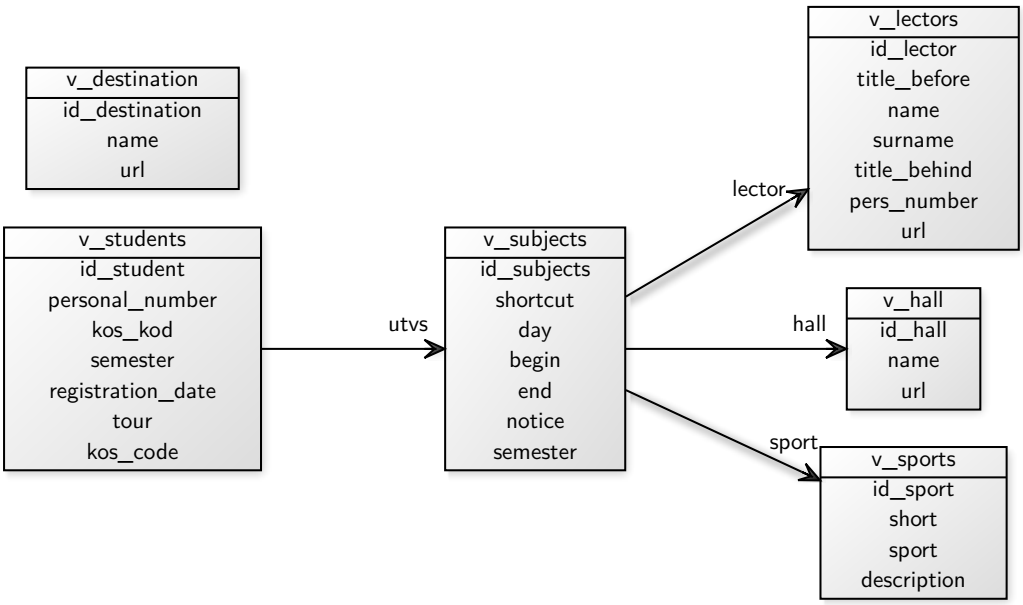
Název sloupce	Datový typ	Popis
id_subjects	smallint(10)	primární klíč
sport	varchar(10)	ID sportu (v_sports.id_sport)
shortcut	varchar(50)	kód ÚTVS předmětu (např. TUR01)
day	varchar(10)	den v týdnu (1–7)
begin	varchar(10)	čas začátku výuky (HH:MM)
end	varchar(10)	čas konce výuky (HH:MM)
hall	varchar(50)	ID haly (v_hall.id_hall)
lector	varchar(50)	ID vyučujícího (v_lectors.id_lector)
notice	text	poznámka (v HTML)
semester	tinyint(4)	parita semestru, ve kterém se vypisuje (1 – zimní, 2 – letní)

2.1.7 Diagram

Na obrázku 2.1 můžete vidět diagram vztahů.

2.2 API zdroje

V této části navrhnu jednotlivé API zdroje (*resources*) a režim přístupu k nim. Nebudu se snažit o striktní návrh, ve kterém bych definoval přesnou podobu odpovědí; to mi umožní nechat přesnou podobu na použitém frameworku. Jednotlivé zdroje budou odpovídat poskytnutým databázovým pohledům.



Obrázek 2.1: Diagram poskytnutých databázových pohledů [76]

2.2.1 /destinations

Poskytne přístup k datům z pohledu `v_destination`. Jednotlivě pomocí primárního klíče (`/destinations/{id}`) nebo hromadně. V odpovědi budou zahrnuta všechna data z tabulky 2.1.

- Položka `id_destination` bude přejmenována na `id`.

Data budou přístupná pro všechny autentizované klienty.

2.2.2 /halls

Poskytne přístup k datům z pohledu `v_hall`. Jednotlivě pomocí primárního klíče (`/halls/{id}`) nebo hromadně. V odpovědi budou zahrnuta všechna data z tabulky 2.2.

- Položka `id_hall` bude přejmenována na `id`.

Data budou přístupná pro všechny autentizované klienty.

2.2.3 /teachers

Poskytne přístup k datům z pohledu `v_lectors`. Jednotlivě pomocí primárního klíče (`/teachers/{id}`) nebo hromadně. K přejmenování dochází kvůli sjednocení s KOSapi, Siriem a dalšími službami. V odpovědi budou zahrnuta všechna data z tabulky 2.3.

- Položka `id_lector` bude přejmenována na `id`.
- Položka `pers_number` bude přejmenována na `personal_number`.
- Položka `title_before` bude přejmenována na `degrees_before`.
- Položka `title_behind` bude přejmenována na `degrees_after`.
- Položka `name` bude přejmenována na `first_name`.
- Položka `surname` bude přejmenována na `last_name`.

Data budou přístupná pro všechny autentizované klienty.

2.2.4 /sports

Poskytne přístup k datům z pohledu `v_sports`. Jednotlivě pomocí primárního klíče (`/sports/{id}`) nebo hromadně. V odpovědi budou zahrnuta všechna data z tabulky 2.4.

- Položka `id_sport` bude přejmenována na `id`.
- Položka `sport` bude přejmenována na `name`.
- Položka `short` bude přejmenována na `shortcut`.

Data budou přístupná pro všechny autentizované klienty.

2.2.5 /enrollments

Poskytne přístup k datům z pohledu `v_students`. Jednotlivě pomocí primárního klíče (`/enrollments/{id}`) nebo hromadně. V odpovědi budou zahrnuta všechna data z tabulky 2.5 kromě položky `kos_code`.

- Položka `id_student` bude přejmenována na `id`.
- Položka `kos_kod` bude přejmenována na `kos_course_code` a bude nastavena na *null*, pokud je `kos_code` 0.
- Položka `utvs` bude přejmenována na `course` a bude obsahovat odkaz na daný zdroj.

- Položka `tour` bude reprezentována jako boolean.

Přístupová práva

- Autentizovaným uživatelům/studentům budou zpřístupněna data o jejich osobě (osobní číslo musí odpovídat osobnímu číslu přihlášeného uživatele).
- Autentizovaným uživatelům/zaměstnancům budou zpřístupněna všechna data.
- Speciálním autentizovaným klientům budou zpřístupněna všechna data, pro služby jako Sirius a podobné.

2.2.6 /courses

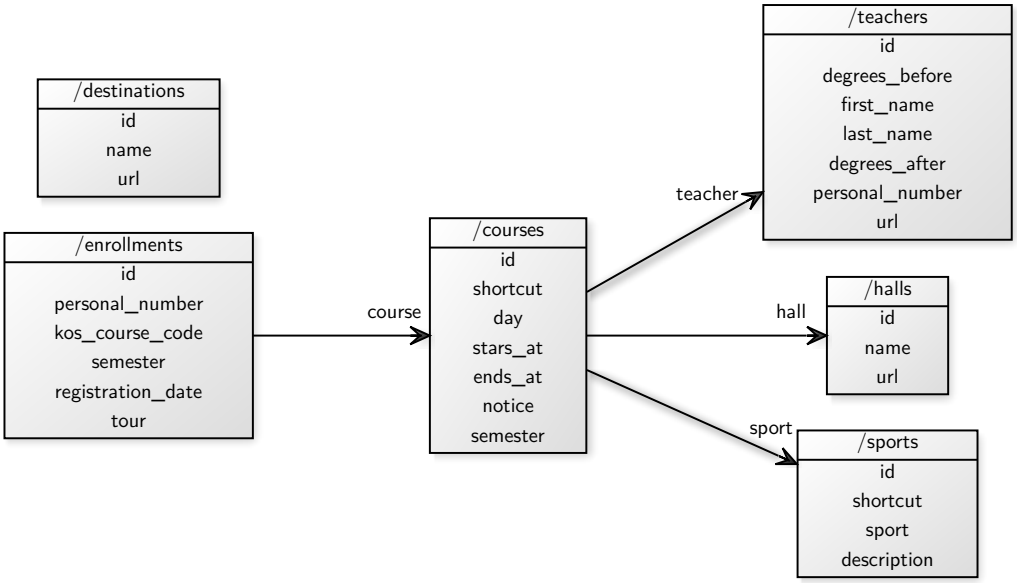
Poskytne přístup k datům z pohledu `v_subjects`. Jednotlivě pomocí primárního klíče (`/courses/{id}`) nebo hromadně. K přejmenování dochází kvůli sjednocení s KOSapi, Siriem a dalšími službami. V odpovědi budou zahrnuta všechna data z tabulky 2.6.

- Položka `id_subjects` bude přejmenována na `id`.
- Položka `lector` bude přejmenována na `teacher`.
- Položka `begin` bude přejmenována na `starts_at`.
- Položka `end` bude přejmenována na `ends_at`.
- Cizí klíče budou reprezentovány odkazem na daný zdroj.

Data budou přístupná pro všechny autentizované klienty.

2.2.7 Diagram

Na obrázku 2.2 můžete vidět upravený diagram vztahů.



Obrázek 2.2: Diagram API zdrojů

Kapitola 3

Implementace

V této kapitole budu zkoumat různé aspekty implementace ukázkové služby z kapitoly *Návrh API pro rozvrhová data ÚTVS ČVUT* ve vybraných frameworcích.

3.1 Zkoumané aspekty implementací

Pro každý aspekt zhodnotím, zda danou věc framework umožňuje, nakolik je to systematické řešení, nakolik jde o řešení pracné a nastíním ukázkou, jak k řešení dojít. Zde nejprve čtenáře krátce seznámím s jednotlivými aspekty.

3.1.1 Namapování dat z pohledů na zdroje

Základní funkcí, kterou snad každý testovaný framework bude disponovat, je namapování dat z databázových pohledů na jednotlivé zdroje. Příkladem je namapování dat z pohledu `v_students` na zdroj `/enrollments`.

3.1.2 Přejmenování položek

V návrhu došlo k přejmenování některých položek. Příkladem je přejmenování položky `surname` na `last_name`.

3.1.3 Prolinkování zdrojů ve stylu HATEOAS

Data jsou v databázi prolinkována pomocí klíčů, v RESTful API je ale žádoucí docílit toho, aby vztahy byly reprezentovány odkazem. Příkladem je odkaz na učitele konkrétního kurzu.

Kromě toho je třeba zobrazit navigační odkazy, například u stránkování na další a předchozí stránku apod.

3.1.4 Úprava zobrazených dat

Některá data se musí zobrazit jinak, než jak jsou uložena v databázi. Příkladem je přetypování řetězců na čísla nebo zobrazení zkratky předmětu pouze v případě, kdy je nastaven patřičný příznak.

3.1.5 Zobrazení dat ve standardizované podobě

Některé frameworky zavádí vlastní formát dat v datových reprezentacích zdrojů. V některých případech je ale vhodnější použít nějaký standardizovaný formát jako JSON API, HAL nebo Siren.

3.1.6 Použití přirozených identifikátorů

Pokud to data umožňují, je vhodné k identifikaci zdroje použít přirozený identifikátor namísto syntetických databázových identifikátorů. Využití syntetických identifikátorů v RESTful API lze považovat za tzv. *leaky abstraction*[18] [135]. Příkladem přirozeného identifikátoru je zkratka sportu, kdy URI nemusí být `/sports/{id}`, ale může být `/sports/{shortcut}`.

Provedl jsem analýzu poskytnutých dat a tabulka sportů je bohužel jediná, která obsahuje použitelný přirozený identifikátor. Ostatní tabulky buď přirozený identifikátor nemají vůbec nebo není unikátní – jednotlivé předměty v různých časech sdílí stejnou zkratku, ne všichni učitelé mají v datech osobní číslo apod.

[18] Nenašel jsem vhodný překlad tohoto termínu do češtiny.

3.1.7 Přístupová práva

Důležitým požadavkem jsou přístupová práva; z hlediska autentizace i autorizace. Příkladem je, že student může vidět jen své vlastní zápisy kurzů.

Pro autentizaci a autorizaci použiji OAuth 2.0 autorizační server (OAAS) FIT ČVUT [75], který umožňuje na základě tokenu poskytnutého klientem určit, jestli je klient autentizován a jaká má práva. Pokud je token svázán s konkrétním uživatelem, z Usermap API [74] zjistím jeho osobní číslo, abych toto mohl porovnávat s osobními čísly učitelů a studentů v databázi ÚTVS.

Vzhledem k tomu, že komunikace s OAAS i Usermap API je na zvoleném frameworku nezávislá, vytvořil jsem malý Python modul, který budu využívat ve všech implementacích; jeho nejpodstatnější součást můžete vidět v ukázce 3.1. Součástí modulu je i jednoduchý server, který simuluje OAAS a Usermap API pro účely testování.

Kompletní implementaci tohoto modulu najdete na přiloženém médiu a na adrese:

`https://github.com/hroncok/utvsapitoken`

3.1.8 Generování dokumentace

Jednou z funkcí, kterou některé frameworky nabízejí, je generování dokumentace přímo z kódu. Příkladem je, že u definice nějakého zdroje použiji Python *docstring* (dokumentační řetězec), a uživatel API bude moci takto definovaný popis vidět.

3.2 Zkoumané funkce služby

Kromě aspektů ve smyslu „jak lze něčeho ve frameworku dosáhnout“ budu zkoumat i tyto funkce implementovaných RESTful API:

- stránkování,
- filtrování,
- řazení,
- vyjednávání o obsahu,

```python
class TokenClient:
    '''Class for making requests for tokens'''

    def __init__(self, check_token_uri=None, usermap_uri=None):
        self.turi = check_token_uri or \
            'https://auth.fit.cvut.cz/oauth/check_token'
        self.uuri = usermap_uri or \
            'https://kosapi.fit.cvut.cz/usermap/v1/people'

    @classmethod
    def _raise_if_error(cls, info, e):
        if 'error' in info:
            msg = info['error']
            if 'error_description' in info:
                msg = info['error_description']
            raise e(msg)

    def token_to_info(self, token):
        '''For given token, produces an info dict'''
        r = requests.get(self.turi, {'token': token})
        info = json.loads(r.text)
        self._raise_if_error(info, TokenInvalid)
        if info['exp'] <= time.time():
            raise TokenExpired('Token is expired')

        if 'user_name' in info:
            pnum, roles = self._extra_from_username(
                info['user_name'], token)
            if pnum is not None:
                info.update({'personal_number': pnum})
            if roles is not None:
                info.update({'roles': roles})

        return info

    def _extra_from_username(self, username, token):
        r = requests.get(
            self.uuri + '/' + username,
            headers={'Authorization': 'Bearer %s' % token})
        info = json.loads(r.text)
        self._raise_if_error(info, UsermapError)
        try:
            pnum = info['personalNumber']
        except KeyError:
            pnum = None
        try:
            roles = info['roles']
        except KeyError:
            roles = None
        return pnum, roles
```

Ukázka kódu 3.1: utvsapitoken: Získání informací o tokenu

- rozcestník.

Rozcestníkem je zde myšlen kořenový zdroj, který poskytuje odkazy na jednotlivé zdroje.

Budu se zabývat tím, jestli dané funkce existují a jak je lze použít. Pokud některá služba bude nabízet i další funkce, zde neuvedené, zmíním je samozřejmě také.

3.3 Django REST framework

3.3.1 Namapování dat z pohledů na zdroje

Pro namapování dat z pohledů na zdroje je jedním z řešení vytvořit Django modely, a pro ty vytvořit serializační třídy a pohledy. Django REST framework umožňuje serializovat i data, která nepochází z modelů, ale to by v tomto případě bylo zbytečně složité.

Jeden model, serializační třídu a pohled můžete vidět v ukázce 3.2; implementační detaily, jako importy, jsou pro stručnost vynechány. Vzhledem k tomu, že je jednodušší rovnou některé položky přejmenovat, je v této ukázce již tak učiněno; detailnější vysvětlení najdete v další části.

Některé kroky, například vytvoření serializační třídy, lze jednoduše automatizovat, jak je vidět z ukázky 3.3. Podobným způsobem by bylo možné zautomatizovat i vytváření pohledů. Ovšem vzhledem k tomu, že dokumentační řetězce u pohledů se zobrazují ve webově procházetelném API, je příhodnější nechat je definované jako jednotlivé třídy.

Namapování dat z pohledů na zdroje v Django REST frameworku je možné, systematické a jednoduché, ale pro tento jednoduchý příklad zbytečně komplexní.

3.3.2 Přejmenování položek

Pro přejmenování položek stačí jinak pojmenovat atribut a poskytnout konstruktoru argument `db_column` s názvem sloupce. Ten je potřeba poskytnout

```python
# models.py
class Course(Model):
    id = SmallIntegerField(primary_key=True,
                           db_column='id_subjects')
    shortcut = StringField()
    day = SmallIntegerField()
    starts_at = TinyStringField(db_column='begin')
    ends_at = TinyStringField(db_column='end')
    notice = TextField()
    semester = SmallIntegerField()
    sport = ForeignKey(Sport, db_column='sport')
    hall = ForeignKey(Hall, db_column='hall')
    teacher = ForeignKey(Teacher, db_column='lector')

    class Meta:
        db_table = 'v_subjects'

# serializers.py
class CourseSerializer(HyperlinkedModelSerializer):
    class Meta:
        model = Course
        fields = tuple(
            f.name for f in model._meta.fields \
            if not f.name.startswith('_'))

# views.py
class CourseViewSet(ReadOnlyModelViewSet):
    '''
    API endpoint that allows courses to be viewed.
    '''
    queryset = Course.objects.all()
    serializer_class = CourseSerializer

# urls.py
router = DefaultRouter()
router.register(r'courses', CourseViewSet)
```

Ukázka kódu 3.2: DRF: Namapování dat z pohledů na zdroje

```
def serializer(model_):
    '''Get a default Serializer class for a model'''
    class _Serializer(HyperlinkedModelSerializer):
        class Meta:
            model = model_
            fields = # ...
    return _Serializer


CourseSerializer = serializer(Course)
```

Ukázka kódu 3.3: DRF: Automatizace vytvoření serializační třídy

u cizích klíčů i v případě, kdy se atribut jmenuje stejně jako položka, protože Django jinak očekává, že se sloupec bude jmenovat `{field}_id`.

Konkrétní příklad přejmenování položek u kurzu můžete vidět na začátku ukázky 3.2.

Přejmenování položek v Django REST frameworku je možné, systematické a triviální.

3.3.3 Prolinkování zdrojů ve stylu HATEOAS

Django REST framework při použití `HyperlinkedModelSerializer` automaticky serializuje cizí klíče jako odkazy.

Prolinkování zdrojů ve stylu HATEOAS v Django REST frameworku je možné, automatické, systematické a triviální.

Navigační odkazy se vytvářejí rovněž automaticky.

3.3.4 Úprava zobrazených dat

Jednou z variant, jak upravit zobrazená data, je vytvořit přímo v modelu metody, které budou data měnit a místo původních dat serializovat výsledky těchto metod. Příklad pro kód předmětu v KOSu můžete vidět v ukázce 3.4.

Pokud má model nadefinované některé položky jako číselné, zobrazují se v odpovědích API číselně, takže není nutné je nijak upravovat.

```
class Enrollment(models.Model):
    # ...
    _kos_course_code = ShortStringField(db_column='kos_kod')
    _kos_code_flag = models.BooleanField(db_column='kos_code')

    @property
    def kos_course_code(self):
        return self._kos_course_code if self._kos_code_flag else None


class EnrollmentSerializer(HyperlinkedModelSerializer):
    class Meta:
        model = Enrollment
        fields = ('kos_course_code', ...) # no _kos_course_code
```

Ukázka kódu 3.4: DRF: Úprava zobrazených dat

Úprava zobrazených dat v Django REST frameworku je možná, systematická a triviální.

3.3.5 Zobrazení dat ve standardizované podobě

Django REST framework data zobrazuje ve velmi jednoduché podobě. Pokud toto chceme změnit, je třeba vytvořit vlastní třídy zodpovědné za stránkování a prezentaci dat.

Naštěstí již existuje modul `drf-hal-json`, který poskytuje třídy pro serializaci do HAL; jeho použití najdete v ukázce 3.5 a výstup v ukázce 3.6. Existují i knihovny pro jiné serializace, např. `djangorestframework-jsonapi` pro JSON API.

Zobrazení dat ve standardizované podobě v Django REST frameworku je možné, systematické, ale pracné, naštěstí však existují knihovny, které lze rovnou použít.

3.3.6 Použití přirozených identifikátorů

Pro použití přirozených identifikátorů stačí v pohledu nastavit hodnotu proměnné `lookup_field` a změnit odkazy vedoucí na daný zdroj, což můžete vidět

```
# settings.py
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
        'drf_hal_json.pagination.HalPageNumberPagination',
    'DEFAULT_PARSER_CLASSES': ('drf_hal_json.parsers.JsonHalParser',),
    'DEFAULT_RENDERER_CLASSES': (
        'drf_hal_json.renderers.JsonHalRenderer',
        'rest_framework.renderers.BrowsableAPIRenderer',
    ),
    'URL_FIELD_NAME': 'self',
    # ...
}


# serializers.py
def serializer(model_):
    '''Get a default Serializer class for a model'''
    class _Serializer(HalModelSerializer):
        # ...
    return _Serializer
```

Ukázka kódu 3.5: DRF: Použití modulu drf-hal-json pro HAL

```
{
    "_links": {
        "self": "http://127.0.0.1:8000/courses/1/",
        "sport": "http://127.0.0.1:8000/sports/3/",
        "hall": "http://127.0.0.1:8000/halls/1/",
        "teacher": "http://127.0.0.1:8000/teachers/6/"
    },
    "id": 1,
    "shortcut": "BAS01",
    "day": 1,
    "starts_at": "13:30",
    "ends_at": "15:00",
    "notice": null,
    "semester": 1
}
```

Ukázka kódu 3.6: DRF: Příklad výstupu pro HAL

v ukázce 3.7. Změna odkazů vyžaduje poměrně mnoho argumentů, které považuji za zbytečné.

```
# serializers.py:
class SportSerializer(HyperlinkedModelSerializer):
    self = HyperlinkedIdentityField(
        read_only=True,
        view_name='sport-detail',
        lookup_field='shortcut')
    # ...

class CourseSerializer(HyperlinkedModelSerializer):
    sport = HyperlinkedRelatedField(
        read_only=True,
        view_name='sport-detail',
        lookup_field='shortcut')
    # ...

# views.py:
class SportViewSet(ReadOnlyModelViewSet):
    # ...
    lookup_field = 'shortcut'
```

Ukázka kódu 3.7: DRF: Použití přirozených identifikátorů

Použití knihovny `drf-hal-json` v kombinaci s přirozenými identifikátory vede k chybě, kterou jsem autorům nahlásil. Pokud knihovna `drf-hal-json` není použita, přirozené identifikátory fungují dle očekávání.

Použití přirozených identifikátorů v Django REST frameworku je možné, systematické, ale zbytečně pracné.

3.3.7 Přístupová práva

Pro přístupová práva se v Django REST frameworku používají třídy dvojího typu: autentizační a autorizační.

Pro autentizaci lze použít již poskytnutou třídu `TokenAuthentication` a přepsat metodu zodpovědnou za validaci tokenu, která vrací informace o uživateli a autorizaci. Vzhledem k tomu, že uživatelem je zde myšlen model `User` frameworku Django a zde tento model nepoužívám, protože aplikace přistupuje k databázi v režimu jen pro čtení, vracím informace o klientu v druhé z návratových hodnot. Toto můžete vidět v ukázce 3.8.

```
class CtuTokenAuthentication(TokenAuthentication):
    '''
    Simple token based authentication using utvsapitoken.

    Clients should authenticate by passing the token
    key in the 'Authorization' HTTP header,
    prepended with the string 'Token '.  For example:

        Authorization: Token 956e252a-513c-48c5-92dd-bfddc364e812
    '''

    def authenticate_credentials(self, key):
        c = TokenClient()
        try:
            info = c.token_to_info(key)
        except:
            raise exceptions.AuthenticationFailed(
                _('Invalid token.'))
        return (None, info)

# settings.py
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'api.authentication.CtuTokenAuthentication',
    ),
    # ...
}
```

Ukázka kódu 3.8: DRF: Autorizační třída a její použití

Zde si dovolím malou odbočku. Třída `TokenAuthentication` z Django REST frameworku očekává v autorizační hlavičce slovo *Token*, ale RFC 6750 říká, že by to v případě OAuthu 2 mělo být *Bearer* [77]. Pokud bych v současnosti chtěl toto změnit, musel bych celý kód třídy zkopírovat a změnit zde právě toto jedno slovo. Navrhl jsem tedy autorům frameworku úpravu, která umožní příslušné slovo změnit jednodušeji. Tato úprava byla přijata a bude dostupná v další vydané verzi frameworku.

Pro autorizaci a samotná přístupová práva jsem napsal dvě třídy; jednu obecně pro všechny zdroje, druhou pouze pro zdroj `/enrollments/`. Obě můžete vidět v ukázce 3.9.

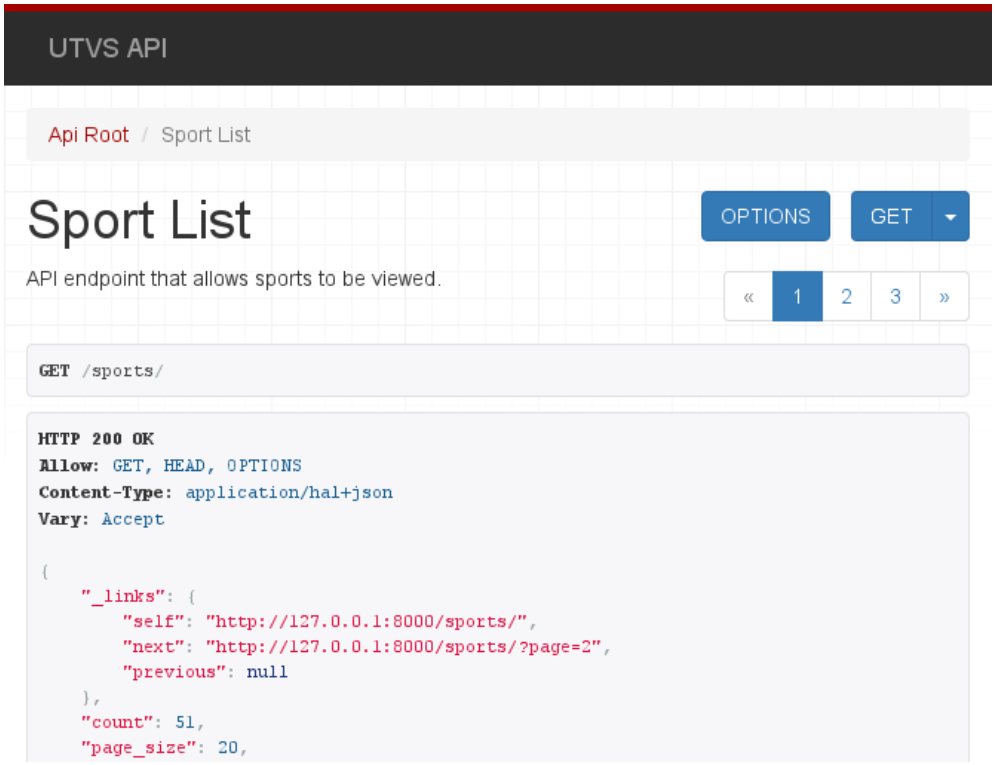
Řízení přístupových práv v Django REST frameworku je možná, systematické a jednoduché.

```python
class HasGeneralReadScopeOrIsApiRoot(BasePermission):
    def has_permission(self, request, view):
        if view.get_view_name() == 'Api Root':
            return True
        return (
            request.auth and
            'cvut:utvs:general:read' in request.auth['scope']
        )


class HasEnrollmentsAcces(BasePermission):
    def has_permission(self, request, view):
        if not request.auth:
            return False

        if 'cvut:utvs:enrollments:all' in request.auth['scope']:
            return True

        if ('cvut:utvs:enrollments:by-role' in request.auth['scope']
                and 'B-00000-ZAMESTNANEC' in request.auth['roles']):
            return True

        if ('cvut:utvs:enrollments:personal' in request.auth['scope']
                and 'personal_number' in request.auth):
            # we should check for this in has_object_permission()
            # but it doesn't apply for list queries
            # so filter the queryset instead
            view.queryset = view.queryset.filter(
                personal_number=request.auth['personal_number'])
            return True

        return False


# settings.py
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': (
        'api.permissions.HasGeneralReadScopeOrIsApiRoot',
    ),
    # ...
}


# views.py
class EnrollmentViewSet(*base):
    # ...
    permission_classes = (permissions.HasGeneralReadScopeOrIsApiRoot,
                          permissions.HasEnrollmentsAcces)
```

Ukázka kódu 3.9: DRF: Třídy pro přístupová práva

3.3.8 Generování dokumentace

Django REST framework nabízí webově procházetelná API. Z dokumentačního hlediska to znamená, že je možné ke každému pohledu napsat dokumentační řetězec, který bude uživateli u jednotlivých zdrojů zobrazen, jak můžete vidět na obrázku 3.1. Pokud je nainstalován modul `markdown`, lze v dokumentačním řetězci použít jazyk Markdown, který je v procházetelném API nahrazen za patřičné HTML značky.



Obrázek 3.1: DRF: Webově procházetelné API

Existují také moduly třetích stran, které slouží ke generování dokumentace API [66].

Generování dokumentace v Django REST frameworku je možné, automatické, systematické a triviální.

3.3.9 Funkce služby

Stránkování

Stránkování funguje automaticky. Lze použít parametry `page` a `page_size`.

```
GET /courses/?page=2&page_size=10
```

Filtrování

Filtrování nefunguje automaticky a jeho zprovoznění není triviální. Je potřeba nainstalovat `django-filter`, nastavit výchozí filtrovací backend a na úrovni pohledů specifikovat položky, podle kterých se dá filtrovat. Což umožňuje velkou kontrolu nad tím, co uživatel smí dělat, ale neumožňuje globálně říct, že se dá všude filtrovat všechno. Pro filtrovaní všech položek ve všech modelech jsem proto vytvořil mixin, který můžete vidět v ukázce 3.10.

```python
class FilterAllFieldsMixin:
    @classproperty
    def filter_fields(cls):
        model = cls.serializer_class.Meta.model
        return serializers.fields(model)


base = (ReadOnlyModelViewSet, FilterAllFieldsMixin)


class DestinationViewSet(*base):
    # ...


# settings.py
REST_FRAMEWORK = {
    'DEFAULT_FILTER_BACKENDS': (
        'rest_framework.filters.DjangoFilterBackend',
    ),
    # ...
}
```

Ukázka kódu 3.10: DRF: Mixin pro filtrování podle všech položek

Poté je možné filtrovat pomocí parametrů v URL:

```
GET /courses/?starts_at=07:30
```

Řazení

Řazení není povoleno automaticky, ale jde o jednoduchou úpravu nastavení, kterou můžete vidět v ukázce 3.11.

```
REST_FRAMEWORK = {
    'DEFAULT_FILTER_BACKENDS': (
        'rest_framework.filters.OrderingFilter',
        # ...
    ),
    # ...
}
```

Ukázka kódu 3.11: DRF: Povolení řazení podle URL

Poté jde položky řadit pomocí parametru `ordering` (název parametru lze v nastavení také změnit). Je možné řadit vzestupně, sestupně i podle více klíčů. Pro seřazení kurzů podle jejich začátku v týdnu od nejpozdějšího lze použít například:

```
GET /courses/?ordering=-day,-starts_at
```

Vyjednávání o obsahu

Django REST framework volí patřičnou zobrazovací třídu podle hlavičky `Accept`. Pokud *není* použita knihovna `drf-hal-json`, je možné nastavovat hlavičkou i způsob odsazování apod.

```
GET /courses/1          Accept: application/json; indent=2
```

Podobně lze volit serializaci do YAMLu nebo XML. Příslušné zobrazovací třídy musí být povoleny v konfiguraci.

Rozcestník

Django REST framework automaticky vytváří rozcestník.

3.3.10 Další poznámky

Hlavním problémem Django REST frameworku je zobrazení dat ve standardizované podobě. Knihovny, které toto umožňují, blokují jinak fungující funkce.

Vlastní implementace je příliš obtížná. Pokud si vystačíte s podobou dat, kterou framework nabízí implicitně, nenarazíte na velký problém.

3.3.11 Kompletní implementace

Kompletní implementaci REST API pro rozvrhová data ÚTVS ČVUT v Django REST frameworku najdete na přiloženém médiu a na adrese:

`https://github.com/hroncok/utvsapi-django`

3.4 Eve

3.4.1 Namapování dat z pohledů na zdroje

Ve výchozím stavu Eve předpokládá uložení dat v NoSQL databázi MongoDB. Je možné si napsat vlastní správce dat, ale jednodušší je použít již existující modul `eve-sqlalchemy`. Pro namapování dat je třeba popsat jednotlivé zdroje pomocí SQLAlchemy modelů a poté je zaregistrovat.

V ukázce 3.12 můžete vidět příklad modelu pro kurzy i vlastní dekorátor pro jeho registraci. Funkce `registerSchema()` z `eve-sqlalchemy` vygeneruje pro každý model schéma, které je následně možné upravit.

Je třeba zdůraznit, že Eve očekává primární klíče pojmenované ve tvaru *_id*, což sice umožňuje změnit, ale pouze globálně (například zde v ukázce na *id*); proto již je položka *id_subjects* přejmenována na *id*, ačkoli o přejmenování položek bude řeč až dále.

Namapování dat z pohledů na zdroje v Eve je možné, systematické a jednoduché.

3.4.2 Přejmenování položek

Jak můžete vidět v ukázce 3.13, pro přejmenování položek stačí provést jednoduchou úpravu modelu – přejmenovat třídní atributy a poskytnout konstruktoru `Column` název sloupce jako první argument.

Přejmenování položek v Eve je možné, systematické a triviální.

```
domain = {}
config.ID_FIELD = config.ITEM_LOOKUP_FIELD = 'id'


def register(cls):
    '''Decorator that registers it and keeps track of it'''
    plural = cls.__name__.lower() + 's'
    registerSchema(plural)(cls)
    domain[plural] = cls._eve_schema[plural]

    # make sure id is our id_field
    # IMHO this should happen automatically but it doesn't
    domain[plural]['id_field'] = config.ID_FIELD

    # make all ids of type objectid
    # should not be neccessry, but feels good :)
    domain[plural]['schema']['id']['type'] = 'objectid'

    return cls

@register
class Course(Base):
    __tablename__ = 'v_subjects'

    id = Column('id_subjects', Integer,
                primary_key=True)
    shortcut = Column(String)
    day = Column(Integer)
    begin = Column(String)
    end = Column(String)
    notice = Column(String)
    semester = Column(Integer)
    sport = Column(Integer, ForeignKey('v_sports.id_sport'))
    hall = Column(Integer, ForeignKey('v_hall.id_hall'))
    lector = Column(Integer, ForeignKey('v_lectors.id_lector'))


SETTINGS = {
    # ...
    'DOMAIN': domain,
}

app = Eve(settings=SETTINGS)
```

Ukázka kódu 3.12: Eve: Namapování dat z pohledů na zdroje

```
@register
class Teacher(Base):
    __tablename__ = 'v_lectors'

    id = Column('id_lector', Integer, primary_key=True)
    degrees_before = Column('title_before', String)
    first_name = Column('name', String)
    last_name = Column('surname', String)
    degrees_after = Column('title_behind', String)
    personal_number = Column('pers_number', Integer)
    url = Column(String)
```

Ukázka kódu 3.13: Eve: Přejmenování položek

3.4.3 Prolinkování zdrojů ve stylu HATEOAS

Eve se dozví o relacích mezi objekty ze schématu. Protože automaticky vytvořené schéma není dostatečné, je třeba jej mírně upravit. Toho jsem docílil úpravou v dekorátoru `@register`, kterou můžete vidět v ukázce 3.14.

```
def register(cls):
    # ...

    # change data_relation's schema a bit
    for field, value in domain[plural]['schema'].items():
        # is it a field with data_relation
        if 'data_relation' in value:
            # resource is the table name by default
            # eve-sqlalchemy juts hopes it will be the same
            # since we rename things, we need to rename it here as well
            # fortunately, we are consistent and can construct it
            value['data_relation']['resource'] = field + 's'
            # make it embeddable, cannot enable it globally
            value['data_relation']['embeddable'] = True

    return cls
```

Ukázka kódu 3.14: Eve: Úprava schématu

Toto v Eve nestačí k vytvoření odkazů, ale pouze umožní odkazované objekty zobrazit vnořeně. Odkazy je sice možné do odpovědi manuálně vložit, ale nejde o koncepční řešení. Eve nabízí možnost úpravy zobrazených dat, kterou více popíši v další části, nyní jen demonstruji manuální vkládání odkazů

v ukázce 3.15. Kód není příliš složitý, ale to jen proto, že jednoduše zkonstruuje odkaz z číselného identifikátoru a názvu položky, což je možné jen díky jednoduchosti této konkrétní ukázkové služby.

```
def make_links(response, *args):
    for arg in args:
        if isinstance(response[arg], dict):
            # embedded
            id = response[arg]['id']
        else:
            id = response[arg]
        response[config.LINKS][arg] = {
            'href': '{}s/{}'.format(arg, id),
            'title': arg.title()
        }

# na úrovni jednotlivého modelu:
make_links(response, 'hall', 'sport', 'teacher')
```

Ukázka kódu 3.15: Eve: Vložení odkazů

Prolinkování zdrojů ve stylu HATEOAS v Eve je možné, nesystematické a v tomto konkrétním případě jednoduché.

Navigační odkazy se vytvářejí automaticky.

3.4.4 Úprava zobrazených dat

Eve nabízí možnost vytvoření funkce, která se naváže k nějaké události. V našem případě nás zajímají události získání dat o nějaké položce nebo položkách. V této funkci pak lze na základě jména zdroje upravit odpověď před serializací do JSONu. V ukázce 3.16 můžete vidět, jak jsem této možnosti využil k úpravě dat.

Eve vkládá do všech objektů datum a čas vytvoření a změny, pokud ho nemůže zjistit (například pokud v databázi není sloupec s tímto údajem), použije 1. leden 1970 (počátek unixového času). Tuto chybnou informaci jsem tedy rovnou v procesu úpravy zobrazených dat odstranil.

Úprava zobrazených dat v Eve je možná, systematická a jednoduchá.

```
classes = {}

def register(cls):
    # ...
    classes[plural] = cls
    return cls

@register
class Course(Base):
    # ...
    def __display_func__(response):
        make_ints(response, 'day', 'hall', 'sport', 'teacher')
        make_links(response, 'hall', 'sport', 'teacher')

@register
class Enrollment(Base):
    # ...
    def __display_func__(response):
        if not response['kos_code_flag']:
            response['kos_course_code'] = None
        del response['kos_code_flag']
        make_links(response, 'course')

def make_links(response, *args):
    # ...

def make_ints(response, *args):
    for arg in args:
        if not isinstance(response[arg], dict):
            # not embedded
            response[arg] = int(response[arg])

def remove_dates(response):
    del response[config.LAST_UPDATED]
    del response[config.DATE_CREATED]

def on_fetched_item(resource, response):
    remove_dates(response)
    if hasattr(classes[resource], '__display_func__'):
        return classes[resource].__display_func__(response)

def on_fetched_resource(resource, response):
    for item in response[config.ITEMS]:
        remove_dates(item)
        if hasattr(classes[resource], '__display_func__'):
            classes[resource].__display_func__(item)

app = Eve(...)
app.on_fetched_item += on_fetched_item
app.on_fetched_resource += on_fetched_resource
```

Ukázka kódu 3.16: Eve: Úprava zobrazených dat

3.4.5 Zobrazení dat ve standardizované podobě

Eve serializuje do formátu, který se podobá HALu, ale nikde se neříká, že to HAL je; příklad můžete vidět v ukázce 3.17. Případná úprava je možná stejným způsobem jako při úpravě zobrazovaných dat.

Zobrazení dat ve standardizované podobě v Eve je možné a částečně automatické, případná úprava je však nesystematická a složitá.

3.4.6 Použití přirozených identifikátorů

Již v úvodu jsem zmínil, že Eve předpokládá primární klíče pojmenované konkrétním způsobem. Není tedy možné použít různé přirozené identifikátory. Umožňuje však přidání sekundárního identifikátoru, což prezentuji v ukázce 3.18.

Poté je možné přistupovat k nějakému sportu pomocí `/sports/{id}` i pomocí `/sports/{shortcut}`.

Použití přirozených identifikátorů v Eve je možné pouze současně s číselným identifikátorem, ale je systematické a triviální.

3.4.7 Přístupová práva

Pro přístupová práva nabízí Eve možnost implementovat speciální třídu. Tu je možné nastavit globálně, ale i na úrovni jednotlivých zdrojů. Současně lze jednoduše použít nějaký atribut objektu k ověření autorizace. Vše je vidět v ukázce 3.19.

Přístupová práva v Eve jsou možná, systematická a velmi jednoduchá.

3.4.8 Generování dokumentace

Samotné Eve generování dokumentace neumožňuje, ale existuje modul `eve-docs`, který tuto funkcionalitu přidává [32].

Tento modul generuje HTML a JSON dokumentaci pouze na základě schématu, nepřidává možnost k jednotlivým zdrojům, metodám a položkám přidávat žádnou textovou informaci. Existuje však zatím nepřijatý návrh na úpravu, která umožňuje i toto [53].

```
{
    "_items": [
        {
            "_etag": "f27aef6240aecc4ccaaad785dc1bfd89b7a6b889",
            "_links": {
                "hall": {"href": "halls/1", "title": "Hall"},
                "self": {"href": "courses/1", "title": "Course"},
                "sport": {"href": "sports/3", "title": "Sport"},
                "teacher": {"href": "teachers/6", "title": "Teacher"}
            },
            "day": 1,
            "ends_at": "15:00",
            "hall": 1,
            "id": 1,
            "notice": null,
            "semester": 1,
            "shortcut": "BAS01",
            "sport": 3,
            "starts_at": "13:30",
            "teacher": 6
        }
    ],
    "_links": {
        "last": {
            "href": "courses?max_results=1&page=729",
            "title": "last page"
        },
        "next": {
            "href": "courses?max_results=1&page=2",
            "title": "next page"
        },
        "parent": {
            "href": "/",
            "title": "home"
        },
        "self": {
            "href": "courses?max_results=1",
            "title": "courses"
        }
    },
    "_meta": {
        "max_results": 1,
        "page": 1,
        "total": 729
    }
}
```

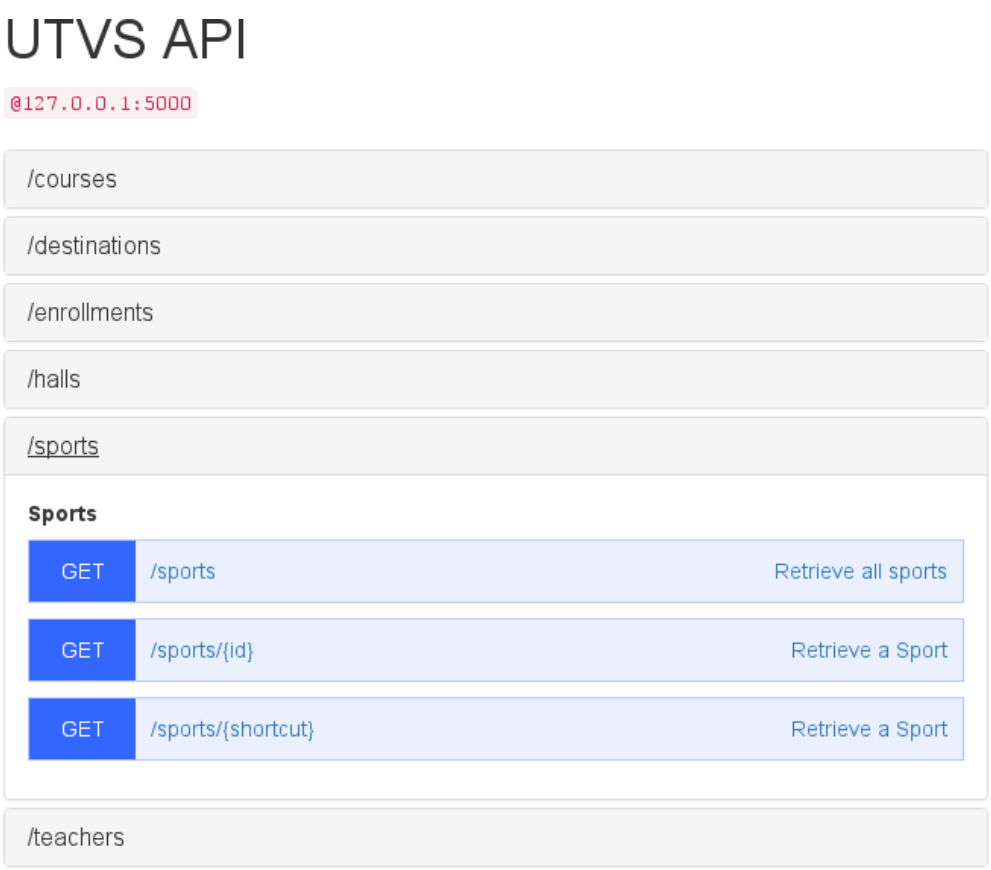

Ukázka kódu 3.17: Eve: Příklad výstupu

```
domain['sports']['additional_lookup'] = {'url': 'regex("[\w]+")',
                                         'field': 'shortcut'}
```

Ukázka kódu 3.18: Eve: Použití přirozených identifikátorů

Vzhledem ke stáří tohoto návrhu, nulové reakci od autora `eve-docs` a dalším faktorům lze usuzovat, že `eve-docs` je mrtvý projekt. Stále je však možné upravenou variantu použít, případně si dopsat úpravy vlastní; například možnost psát popisy v jazyce Markdown apod.

Pro zapnutí generování dokumentace stačí modul naimportovat a registrovat, pro využití zmíněné úpravy je pak možné přidat do schématu další položky. Obojí je znázorněno v ukázce 3.20, výsledek můžete vidět na obrázku 3.2.



Obrázek 3.2: Eve: Vygenerovaná HTML dokumentace

```python
from flask import request


class BearerAuth(BasicAuth):
    '''Overrides Eve's built-in basic authorization scheme
    and uses utvsapitoken to validate bearer token'''
    def auth_logic(self, info, resource, method):
        return 'cvut:utvs:general:read' in info['scope']

    def check_auth(self, token, resource, method):
        c = TokenClient()
        try:
            info = c.token_to_info(token)
        except:
            return False

        return self.auth_logic(info, resource, method)

    def authorized(self, allowed_roles, resource, method):
        try:
            token = request.headers.get('Authorization').split(' ')[1]
        except:
            return False
        return self.check_auth(token, resource, method)


class EnrollmentsAuth(BearerAuth):
    '''Overrides auth_logic for Enrollments'''
    def auth_logic(self, info, resource, method):
        if not super().auth_logic(info, resource, method):
            return False

        if 'cvut:utvs:enrollments:all' in info['scope']:
            return True

        if ('cvut:utvs:enrollments:by-role' in info['scope'] and
                'B-00000-ZAMESTNANEC' in info['roles']):
            return True

        if ('cvut:utvs:enrollments:personal' not in info['scope'] or
                'personal_number' not in info):
            return False

        # only see your enrollments, pretty easy:
        self.set_request_auth_value(info['personal_number'])
        return True

domain['enrollments']['authentication'] = EnrollmentsAuth
domain['enrollments']['auth_field'] = 'personal_number'

app = Eve(auth=BearerAuth, ...)
```

Ukázka kódu 3.19: Eve: Autorizační třídy

```
def register(cls):
    # ...
    domain[plural]['description'] = {'general': cls.__name__ + 's'}
    if cls.__doc__:
        domain[plural]['description'].update(
            {'methods': {'GET': cls.__doc__}})
    # ...


@register
class Destination(Base):
    '''This resource represents a destination etc.'''
    # ...


from eve_docs import eve_docs
from flask.ext.bootstrap import Bootstrap

app = Eve(...)
Bootstrap(app)
app.register_blueprint(eve_docs, url_prefix='/docs')
```

Ukázka kódu 3.20: Eve: Generování dokumentace

Generování dokumentace v Eve je možné s dalším modulem, systematické a triviální.

3.4.9 Funkce služby

Stránkování

Stránkování se děje automaticky, zobrazenou stránku lze ovlivnit parametrem `page` a počet výsledků na stránce parametrem `max_results`.

```
GET /courses/?page=3&max_results=5
```

Filtrování

Filtrovat se dá pomocí JSONu v parametru `where`. Například takto:

```
GET /courses/?where={"teacher": 2}
```

Nelze stanovit žádnou podmínku, například větší než apod. Nelze kombinovat více filtrů.

Řazení

Řadit se dá parametrem `sort` podle různých položek a to včetně určení směru řazení a použití více řadících podmínek. Následující příklad seřadí kurzy podle dnu v týdnu od nejpozdějšího a následně v případě shody podle čísla haly.

```
GET /courses/?sort=-day,hall
```

Řazení, filtrování a stránkování se dá libovolně kombinovat.

Vyjednávání o obsahu

Na základě hlavičky *Accept* Eve serializuje do JSONu (výchozí) nebo do XML (ukázka 3.21).

```
GET /courses/1          Accept: application/xml
```

```
<?xml version="1.0"?>
<resource href="courses/1" title="Course">
  <link rel="collection" href="courses" title="courses"/>
  <link rel="hall" href="halls/1" title="Hall"/>
  <link rel="parent" href="/" title="home"/>
  <link rel="sport" href="sports/3" title="Sport"/>
  <link rel="teacher" href="teachers/6" title="Teacher"/>
  <_etag>f27aef6240aecc4ccaaad785dc1bfd89b7a6b889</_etag>
  <day>1</day>
  <ends_at>15:00</ends_at>
  <hall>1</hall>
  <id>1</id>
  <notice/>
  <semester>1</semester>
  <shortcut>BAS01</shortcut>
  <sport>3</sport>
  <starts_at>13:30</starts_at>
  <teacher>6</teacher>
</resource>
```

Ukázka kódu 3.21: Eve: Serializace do XML

Rozcestník

Eve automaticky vytváří rozcestník.

Vnořené položky

Eve umožňuje zobrazit odkazované položky vnořeně, pomocí JSON parametru `embedded`. Toto je potřeba povolit ve schématu (ukázka 3.14 na straně 108).

```
GET /enrollments/28477/?embedded={"course": true}
```

3.4.10 Další poznámky

Největším problém Eve je absence automaticky vytvořených odkazů a víceméně fixně daný formát výstupu, ten je ale poměrně dobře navržen.

3.4.11 Kompletní implementace

Kompletní implementaci REST API pro rozvrhová data ÚTVS ČVUT ve frameworku Eve najdete na přiloženém médiu a na adrese:

`https://github.com/hroncok/utvsapi-eve`

3.5 ripozo

3.5.1 Namapování dat z pohledů na zdroje

Pro namapování SQL dat na zdroje je možné použít modul `ripozo-sqlalchemy`. Ke každé entitě je potřeba vytvořit třídy pro model, správce a zdroj. Příklad pro zdroj `/enrollments` můžete vidět v ukázce 3.22.

`ripozo-sqlalchemy` také nabízí funkci `create_resource()`, která přijímá model a automaticky vytvoří třídy pro správce a zdroj. Nepřišla mi ale dostatečně flexibilní, tak jsem si podobnou napsal sám, ve formě dekorátoru, který můžete vidět v ukázce 3.23.

Při použití dekorátoru `@register` tak stačí vytvořit pouze třídu pro model. V případě, kdy se primární klíč nejmenuje *id*, je potřeba ještě nastavit jeho název do třídního atributu `__pks__`.

Nutnost vytvořit tři třídy pro každý zdroj se může jevit přehnaná, umožňuje to ale velkou míru přizpůsobení, například pokud by každý zdroj byl namapován na jinou databázi apod.

```
class Enrollment(db.Model):
    __tablename__ = 'v_students'

    id_student = db.Column(db.Integer, primary_key=True)
    personal_number = db.Column(db.Integer)
    kos_kod = db.Column(db.String)
    utvs = db.Column(db.Integer,
                     db.ForeignKey('v_subjects.id_subjects'))
    semester = db.Column(db.String)
    registration_date = db.Column(db.DateTime)
    tour = db.Column(db.Boolean)
    kos_code = db.Column(db.Boolean)


class EnrollmentManager(AlchemyManager):
    model = Enrollment
    fields = ('id_student', 'personal_number', 'kos_kod', 'utvs',
              'semester', 'registration_date', 'tour', 'kos_code')


class PersonResource(restmixins.RetrieveRetrieveList):
    manager = EnrollmentManager(session_handler)
    pks = ('id_student',)
    resource_name = 'enrollments'


dispatcher.register_resources(PersonResource)
```

Ukázka kódu 3.22: ripozo: Namapování dat z pohledů na zdroje

Namapování dat z pohledů na zdroje v ripozu je možné, systematické, pro jednoduché aplikace příliš komplexní, ale ne příliš složité.

3.5.2 Přejmenování položek

Pro přejmenování položek stačí provést jednoduchou úpravu modelu. Jak můžete vidět v ukázce 3.24, stačí přejmenovat třídní atributy a poskytnout konstruktoru `Column` název sloupce jako první argument.

Přejmenování položek v ripozu je možné, systematické a triviální.

3.5.3 Prolinkování zdrojů ve stylu HATEOAS

Pro prolinkování zdrojů je potřeba:

```
resources = {}

def register(cls):
    '''Create default Manager and Resource class for model
    and register it'''
    fields = tuple(
        f for f in cls.__dict__.keys() if not f.startswith('_'))
    pks = getattr(cls, '__pks__', ('id',))

    manager_cls = type(cls.__name__ + 'Manager',
                       (AlchemyManager,),
                       {'fields': fields,
                        'model': cls})

    resource_cls = type(cls.__name__ + 'Resource',
                        (restmixins.RetrieveRetrieveList,),
                        {'manager': manager_cls(session_handler),
                         'resource_name': cls.__name__.lower() + 's',
                         'pks': pks})

    resources[cls.__name__] = resource_cls
    return cls

# later:
dispatcher.register_resources(*resources.values())
```

Ukázka kódu 3.23: ripozo: Dekorátor pro registraci modelů

```
@register
class Teacher(db.Model):
    __tablename__ = 'v_lectors'

    id = db.Column('id_lector', db.Integer, primary_key=True)
    degrees_before = db.Column('title_before', db.String)
    first_name = db.Column('name', db.String)
    last_name = db.Column('surname', db.String)
    degrees_after = db.Column('title_behind', db.String)
    personal_number = db.Column('pers_number', db.Integer)
    url = db.Column(db.String)
```

Ukázka kódu 3.24: ripozo: Přejmenování položek

1. Přidat do modelu další atribut reprezentující vztah/odkaz.
2. Přidat vztah do atributu `_relationships` třídy zdroje.

Toto lze také udělat automaticky, pokud data dodržují nějakou jmennou konvenci. Automatický způsob, který předpokládá, že cizí klíče jsou pojmenované jako `{model}_id`, můžete vidět v ukázce 3.25.

```
def fk_magic(cls, fields):
    '''Create links automagically'''
    fks = tuple(field for field in fields if field.endswith('_id'))
    rels = []
    for fk in fks:
        unfk = fk[:-3]  # foo_id -> foo
        setattr(cls, unfk,
                relationship(unfk.title(),
                             foreign_keys=(getattr(cls, fk),)))
        rels.append(Relationship(unfk,
                                 property_map={fk: 'id'},
                                 relation=unfk.title() + 'Resource'))
    return tuple(rels)  # must be a tuple

def register(cls):
    # ...
    rels = fk_magic(cls, fields)

    # ...
    resource_cls = type(..., {'_relationships': rels, ...})

    # ...
    return cls


@register
class Course(db.Model):
    __tablename__ = 'v_subjects'

    id = db.Column('id_subjects', db.Integer,
                   primary_key=True)
    # ...
    sport_id = db.Column('sport', db.Integer,
                         db.ForeignKey('v_sports.id_sport'))
    hall_id = db.Column('hall', db.Integer,
                        db.ForeignKey('v_hall.id_hall'))
    teacher_id = db.Column('lector', db.Integer,
                           db.ForeignKey('v_lectors.id_lector'))
```

Ukázka kódu 3.25: ripozo: Automatické vytvoření odkazů

Prolinkování zdrojů ve stylu HATEOAS v ripozu je možné, systematické, ale zbytečně komplexní.

Navigační odkazy se vytvářejí podle druhu výstupu automaticky.

3.5.4 Úprava zobrazených dat

Ripozo nabízí *preprocesory* a *postprocesory*, které lze použít i pro úpravu zobrazených dat.

Postprocesor v našem případě musíme aplikovat pro požadavek na jeden zdroj i na seznam zdrojů. Bohužel se oba takové postprocesory musí chovat trochu jinak, naštěstí to ale také můžeme zautomatizovat. V ukázce 3.26 je posprocesor pro kód kurzu z KOSu i dekorátor, který způsobí, že bude korektně aplikován v obou výše zmíněných případech. Pre- a postprocesory se nastavují ve třídě zdroje, proto je v ukázce i drobná úprava dekorátoru `@register`.

Úprava zobrazených dat v ripozu je možná, systematická a jednoduchá.

3.5.5 Zobrazení dat ve standardizované podobě

Jednou z hlavních výhod ripoza je integrovaná podpora pro HAL, Siren i JSON API. Jednotlivé formáty lze použít dokonce i zároveň; ripozo pak vrátí ten, o který si klient zažádá, případně první v pořadí registrace (ukázka 3.27).

Příklad výstupu pro HAL můžete vidět v ukázkce 3.28.

Zobrazení dat ve standardizované podobě v ripozu je možné, systematické a automatické.

3.5.6 Použití přirozených identifikátorů

Pro použití přirozených identifikátorů stačí změnit primární klíč, jak můžete vidět v ukázce 3.29.

Bohužel pak přestanou fungovat odkazy, jelikož ripozo v momentě konstrukce odkazu zná pouze sloupec, na který je odkaz vázán (což je zde *id* a ne *shortcut*).

```
def register(cls):
    # ...
    pres = getattr(cls, '__preprocessors__', tuple())
    posts = getattr(cls, '__postprocessors__', tuple())

    resource_cls = type(..., {'preprocessors': pres,
                              'postprocessors': posts,
                              ...})
    # ...
    return cls

def onemany(func):
    '''
    Decorator for postprocessors in order to run a given function
    an all resources
    '''
    def processor(cls, function_name, request, resource):
        if function_name == 'retrieve':
            return func(cls, function_name, request, resource)
        if function_name == 'retrieve_list':
            for one in resource.related_resources[0].resource:
                func(cls, 'retrieve', request, one)
    return processor


@register
class Enrollment(db.Model):
    # ...
    kos_course_code = db.Column('kos_kod', db.String)
    kos_code_flag = db.Column('kos_code', db.Boolean)

    @onemany
    def _post_kos_code_null(cls, function_name, request, resource):
        '''This will be called as a function, so no self!'''
        if not resource.properties['kos_code_flag']:
            resource.properties['kos_course_code'] = None
        del resource.properties['kos_code_flag']

    __postprocessors__ = (_post_kos_code_null,)
```

Ukázka kódu 3.26: ripozo: Úprava zobrazených dat

```
dispatcher.register_adapters(adapters.HalAdapter,
                             adapters.SirenAdapter,
                             adapters.JSONAPIAdapter,
                             adapters.BasicJSONAdapter)
```

Ukázka kódu 3.27: ripozo: Zobrazení dat ve standardizované podobě

```
{
    "_embedded": {},
    "_links": {
        "hall": {
            "href": "/halls/29"
        },
        "self": {
            "href": "http://127.0.0.1:5000/courses/2158"
        },
        "sport": {
            "href": "/sports/107"
        },
        "teacher": {
            "href": "/teachers/42"
        }
    },
    "day": 5,
    "ends_at": "15:30",
    "id": 2158,
    "notice": "P\u0158KS /k\u00f3dy 17PBPTV2 a 17BPTV2/ - sebeobrana",
    "semester": 2,
    "shortcut": "FBM14",
    "starts_at": "14:00"
}
```

Ukázka kódu 3.28: ripozo: Příklad výstupu pro HAL

```
@register
class Sport(db.Model):
    __tablename__ = 'v_sports'
    __pks__ = ('shortcut',)

    id = db.Column('id_sport', db.Integer, primary_key=True)
    shortcut = db.Column('short', db.String)
    name = db.Column('sport', db.String)
    description = db.Column(db.String)
```

Ukázka kódu 3.29: ripozo: Použití přirozených identifikátorů

Použití přirozených identifikátorů v ripozu je sice možné, systematické a triviální, ale rozbije to jinou část.

3.5.7 Přístupová práva

Ripozo nepřináší žádný zabudovaný mechanismus pro správu přístupových práv, podle dokumentace je na to vhodné použít pre- a postprocesory [99].

Vytvořil jsem tedy hlavní preprocesor, který v dekorátoru `@register` vkládám ke všem zdrojům. Tento preprocesor ověří token pomocí modulu `utvsapitoken` a vyhodnotí, jestli má klient právo ke čtení. Preprocesor můžete vidět v ukázce 3.30. Použité třídy výjimek jsem si musel vytvořit, ale kvůli trivialitě je zde neuvádím. Ripozo zařídí, že se výjimky správně projeví v odpovědi serveru (stavovým kódem a zprávou o chybě).

Pro komplikovanější logiku je potřeba přidat pre-/postprocesor na úrovni zdroje. U zdroje `/enrollments` musíme zajistit, aby data byla dostupná pouze pro speciálně autorizované klienty. Pro výpis zápisů je potřeba použít preprocesor, pro konkrétní zápis pak postprocesor, abychom mohli přistupovat ke zdroji a zjistit, jakému studentovi náleží apod.

V ukázce 3.31 můžete vidět zjednodušenou variantu funkce, která slouží zároveň jako preprocesor i jako postprocesor. Kompletní kód včetně vysvětlujících komentářů je součástí implementace služby.

Přístupová práva v ripozu jsou možná, částečně systematická, ale pro složitější logiku mohou být příliš komplikovaná.

3.5.8 Generování dokumentace

Ripozo toto neumožňuje.

3.5.9 Funkce služby

Dokumentace ripoza neuvádí nic o možnostech stránkování, filtrování apod. Prohlídkou kódu jsem zjistil, že tyto možnosti obstarává `ripozo-sqlalchemy`, který dokumentací spíše šetří, informace zde uvedené jsou tedy získány experimentálně.

```
def headers_to_token(headers, *, authorization='authorization',
                     bearer='Bearer '):
    '''
    Get the auth token form the headers

    Returns None if not found
    '''
    if authorization in headers:
        header = headers[authorization]
        if header.startswith(bearer):
            return header[len(bearer):].strip()


def preprocessor(cls, function_name, request):
    token = headers_to_token(request.headers)
    if not token:
        raise exceptions.UnauthorizedException(
            'Token not provided. Use the following header: '
            'Authorization: Bearer {token}')
    c = TokenClient()

    try:
        info = c.token_to_info(token)
    except:
        raise exceptions.UnauthorizedException(
            'Token not valid. Please provide a valid token.')

    # default behavior for all of our resources
    if 'cvut:utvs:general:read' not in info['scope']:
        raise exceptions.ForbiddenException(
            'Permission denied. You need '
            'cvut:utvs:general:read scope.')

    # add the information to the request,
    # for further pre/postprocessors
    request.client_info = info
```

Ukázka kódu 3.30: ripozo: Autorizační preprocesor

```
class Enrollment(db.Model):
    # ...

    def _prepost_auth_logic(cls, message, request, resource=None):
        scope = request.client_info['scope']

        # can read anything
        if 'cvut:utvs:enrollments:all' in scope:
            return

        if 'cvut:utvs:enrollments:by-role' in scope:
            if 'B-00000-ZAMESTNANEC' in request.client_info['roles']:
                return

        if 'cvut:utvs:enrollments:personal' in scope:
            pnum = request.client_info['personal_number']
            if not pnum:
                raise exceptions.ForbiddenException(
                    'Permission denied.')

            if resource:
                # this is one resource
                # you are the student of this resource
                if pnum == resource.properties['personal_number']:
                    return
            else:
                # this is a list of resources
                # you are a person, but not a teacher
                # we'll filter all the enrollments by personal_number
                request.query_args.update({'personal_number': [pnum]})
                return

        # out of options
        raise exceptions.ForbiddenException('Permission denied.')
```

Ukázka kódu 3.31: ripozo: Autorizační pre-/postprocesor zdroje Enrollment

Stránkování

Stránkovat je možné standardně pomocí parametrů `count` a `page`.

```
GET /courses/?page=5&count=5
```

Filtrování

Filtrovat výsledky se dá pouze jednoduchým způsobem, například takto zobrazíme seznam kurzů probíhajících v pátek:

```
GET /courses/?day=5
```

Nelze ale filtrovat na základě cizích klíčů, ani nastavit podmínku (větší než apod.). Při špatně provedeném dotazu může výsledek skončit chybou ripoza, což jsem nahlásil jako chybu, na jejíž opravě již autor `ripozo-sqlalchemy` pracuje.

Filtrování a stránkování se dá kombinovat, je možné použít více filtrů. Navigační odkazy na další stránky neobsahují použitý filtr, což jsem také nahlásil jako chybu autorovi.

Řazení

Nepřišel jsem na způsob, jak seznam řadit jinak než implicitně. Zde je třeba zdůraznit, že se jedná o nedostatek modulu `ripozo-sqlalchemy`, nikoliv ripoza.

Vyjednávání o obsahu

Na základě hlavičky `Accept` ripozo volí vhodný *adaptér* (HAL, Siren atd.).

Rozcestník

Rozcestník je automaticky vytvořen, odpovídá ale pouze na metodu *OPTIONS*.

Seznam položek

Zásadním nedostatkem služby je nemožnost zobrazení seznamu položek jinak než formou odkazů. To vede k nutnosti zaslání $N + 1$ dotazů, potřebujme-li

získat informace o N položkách, přičemž u jiných implementací toho lze docílit jediným dotazem.

3.5.10 Další poznámky

Při implementaci byl použit *dispatcher* pro Flask. Ripozo umožňuje využití jiných frameworků, ale v současné době je k dispozici pouze navázání na Flask a Django.

Vzhledem k výsledkům benchmarku v části 1.13.1 na straně 64 by bylo z hlediska rychlosti zajímavé implementovat napojení na webový framework Falcon. Jelikož se ale tato práce obecně rozdíly mezi webovými frameworky nezabývá, nechávám tuto možnost otevřenou.

3.5.11 Kompletní implementace

Kompletní implementaci REST API pro rozvrhová data ÚTVS ČVUT ve frameworku ripozo najdete na přiloženém médiu a na adrese:

`https://github.com/hroncok/utvsapi-ripozo`

3.6 sandman2

3.6.1 Namapování dat z pohledů na zdroje

Podle dokumentace sandmanu2 [83] by mělo stačit spustit příkaz z ukázky 3.32 a API by se mělo „samo vytvořit“.

```
$ sandman2ctl 'mysql://uzivatel:heslo@server/databaze'
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Ukázka kódu 3.32: sandman2: Automatické vytvoření REST API

Pokud ale používáme databázové pohledy, nikoliv přímo tabulky, a potřebujeme ovlivnit názvy zdrojů, nezbývá nám, než nadefinovat modely ručně pomocí SQLAlchemy modelů. Model pro `/destinations` můžete vidět v ukázce 3.33.

```
class Destinations(sandman2.model.db.Model):
    __tablename__ = 'v_destination'

    id_destination = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String)
    url = db.Column(db.String)

app = sandman2.get_app(url, user_models=[Destinations], read_only=True)
```

Ukázka kódu 3.33: sandman2: Namapování dat z pohledů na zdroje

Namapování dat z pohledů na zdroje v sandmanu2 je možné, systematické a jednoduché, ale ne plně automatické, jak by se z popisu sandmanu2 mohlo zdát.

3.6.2 Přejmenování položek

Pro přejmenování položek stačí provést jednoduchou úpravu modelu. Jak můžete vidět v ukázce 3.34, stačí přejmenovat třídní atributy a poskytnout konstruktoru `Column` název sloupce jako první argument a název atributu jako argument `key`.

Bohužel sandman2 s tím nepočítá a je potřeba předefinovat jednu metodu, která místo nového názvu vrací název sloupce v tabulce. Dle mého názoru se jedná o chybu a její opravu jsem navrhl autorovi na GitHubu, zatím bez odezvy. Vytvořil jsem tedy mixin, který použitým modelům tuto metodu předefinuje (vrchní část ukázky 3.34).

Přejmenování položek v sandmanu2 je možné, do určité míry systematické[19] a triviální.

[19]Má výhrada zde směřuje k nutnosti opakování názvu atributu. Nutnost předefinovat metodu je patrně nezamýšlená.

```
class CustomizingMixin(Model):
    '''Mixin that fixes primary_key method'''
    def primary_key(self):
        '''Return the key of the model's primary key field'''
        return list(self.__table__.primary_key.columns)[0].key


class Teachers(CustomizingMixin, db.Model):
    __tablename__ = 'v_lectors'

    id = db.Column('id_lector', db.Integer,
                   primary_key=True, key='id')
    degrees_before = db.Column('title_before', db.String,
                               key='degrees_before')
    first_name = db.Column('name', db.String, key='first_name')
    last_name = db.Column('surname', db.String, key='last_name')
    degrees_after = db.Column('title_behind', db.String,
                              key='degrees_after')
    personal_number = db.Column('pers_number', db.Integer,
                                key='personal_number')
    url = db.Column(db.String)
```

Ukázka kódu 3.34: sandman2: Přejmenování položek

3.6.3 Prolinkování zdrojů ve stylu HATEOAS

Sandman2 odkazy nevytváří automaticky, je ale poměrně jednoduché je vytvořit ručně. Stačí na modelu předefinovat metodu `to_dict()` a zde odkazy sestrojit z cizích klíčů. Přidal jsem tedy upravenou variantu této metody do již vytvořeného mixinu (ukázka 3.35).

Narazil jsem na problém, že z cizího klíče sice poznám tabulku, ale ne model. Vyřešil jsem to tak, že před přidáním modelů do aplikace je registruji do reverzního seznamu podle tabulek (pomocí dekorátoru). Tento způsob se mi příliš nelíbí, ale sandman2 žádný vlastní způsob nenabízí.

Prolinkování zdrojů ve stylu HATEOAS v sandmanu2 je možné, velmi nesystematické, ale poměrně jednoduché.

Navigační odkazy se automaticky nevytvářejí a úprava tohoto chování není možná.

```
class CustomizingMixin(Model):
    # ...

    def to_dict(self):
        '''Return the resource as a dictionary'''
        result_dict = {}
        for column in self.__table__.columns:
            name = column.key
            value = result_dict[name] = getattr(self, name, None)
            if column.foreign_keys:
                # Foreign key, turn it to a link, HATEOAS, yay!
                # We always have only one f. key in one column
                fk = list(column.foreign_keys)[0]
                model = modelstore.reverse_lookup(fk.column.table)
                instance = model.query.get(int(value))
                if instance:
                    result_dict[name] = instance.resource_uri()
        result_dict['self'] = self.resource_uri()
        return result_dict
```

Ukázka kódu 3.35: sandman2: Prolinkování zdrojů ve stylu HATEOAS

3.6.4 Úprava zobrazených dat

Úpravu zobrazených dat lze provést v metodě `to_dict()`. Bylo by možné používat různé varianty této metody pro různé modely, ale v našem případě si vystačíme s jedinou metodou. Úpravu pro číselné typy a kód kurzu z KOSu můžete vidět v ukázce 3.36.

Úprava zobrazených dat v sandmanu2 je možná, nepříliš systematická, ale jednoduchá.

3.6.5 Zobrazení dat ve standardizované podobě

Úpravu způsobu zobrazení jedné entity je možné provést v metodě `to_dict()`, úpravu pro seznamu entit však provést nejde.

V ukázce 3.37 je vidět úprava ve stylu HAL.

Příklad výstupu pro HAL můžete vidět v ukázkce 3.38.

Zobrazení dat ve standardizované podobě v sandmanu2 je částečně možné, nepříliš systematické a jednoduché v závislosti na zvoleném standardu.

```
class CustomizingMixin(Model):
    # ...

    def to_dict(self):
        '''Return the resource as a dictionary'''
        result_dict = {}
        for column in self.__table__.columns:
            name = column.key
            value = result_dict[name] = getattr(self, name, None)
            if column.foreign_keys:
                # ...
            elif isinstance(column.type, db.Integer):
                # Return the value as int, otherwise it might
                # get returned as str due to bad SQL type
                result_dict[name] = int(value)
            # ...
        try:
            if not result_dict['_kos_code']:
                result_dict['kos_course_code'] = None
            del result_dict['_kos_code']
        except KeyError:
            pass
        return result_dict
```

Ukázka kódu 3.36: sandman2: Úprava zobrazených dat

3.6.6 Použití přirozených identifikátorů

Pro použití přirozeného identifikátoru lze v modelu nastavit jiný primární klíč. Následně je v našem případě potřeba v metodě `to_dict()` změnit řádku kódu, která najde patřičný objekt podle cizího klíče. Obojí můžete vidět v ukázce 3.39.

Použití přirozených identifikátorů v sandmanu2 je možné, systematické a jednoduché.

3.6.7 Přístupová práva

Implementace přístupových práv bez velkého zásahu do kódu sandmanu2 není možná.

```
class CustomizingMixin(Model):
    # ...
    def to_dict(self):
        '''Return the resource as a dictionary'''
        result_dict = {'_links': {}}
        for column in self.__table__.columns:
            name = column.key
            value = result_dict[name] = getattr(self, name, None)
            if column.foreign_keys:
                fk = list(column.foreign_keys)[0]
                model = modelstore.reverse_lookup(fk.column.table)
                instance = model.query.get(int(value))
                if instance:
                    result_dict['_links'][name] = \
                        {'href': instance.resource_uri()}
                    del result_dict[name]
            # ...
            elif isinstance(value, datetime.datetime):
                # Display datetimes in ISO format
                result_dict[name] = value.isoformat()
        result_dict['_links']['self'] = {'href': self.resource_uri()}
        # ...
        return result_dict
```

Ukázka kódu 3.37: sandman2: Zobrazení dat ve standardizované podobě

3.6.8 Generování dokumentace

Sandman2 toto neumožňuje.

3.6.9 Funkce služby

Dokumentace sandmanu2 o těchto možnostech mlčí, existuje však iniciativa za zdokumentování těchto funkcí [12].

Zde je také potřeba zmínit, že URI zdrojů fungují jen bez koncového lomítka.

Stránkování

Je možné zvolit pouze číslo stránky pomocí parametru `page`. Velikost stránky nelze ovlivnit (je vždy 20). Bez použití parametru `page` se implicitně vrátí celý seznam, což v případě velkého počtu položek představuje problém.

```
{
    "_links": {
        "hall": {
            "href": "/halls/29"
        },
        "self": {
            "href": "/courses/2158"
        },
        "sport": {
            "href": "/sports/107"
        },
        "teacher": {
            "href": "/teachers/42"
        }
    },
    "day": 5,
    "ends_at": "15:30",
    "id": 2158,
    "notice": "P\u0158KS /k\u00f3dy 17PBPTV2 a 17BPTV2/ - sebeobrana",
    "semester": 2,
    "shortcut": "FBM14",
    "starts_at": "14:00"
}
```

Ukázka kódu 3.38: sandman2: Příklad výstupu pro HAL

```
class CustomizingMixin(Model):
    # ...
    def to_dict(self):
        '''Return the resource as a dictionary'''
        # ...
            # WAS: instance = model.query.get(int(value))
            instance = model.query.filter_by(id=int(value)).first()
        # ...
        return result_dict


@modelstore.register
class Sports(mixins.CustomizingMixin, db.Model):
    __tablename__ = 'v_sports'

    id = db.Column('id_sport', db.Integer, key='id')
    shortcut = db.Column('short', db.String, primary_key=True,
                         key='shortcut')
    # ...
```

Ukázka kódu 3.39: sandman2: Použití přirozených identifikátorů

```
GET /courses?page=5
```

Je možné použít parametr `limit`, ne však v kombinaci s parametrem `page`.

```
GET /courses?limit=5
```

Filtrování

Filtrovat výsledky se dá pouze jednoduchým způsobem, například takto můžeme zobrazit seznam kurzů probíhajících v pátek:

```
GET /courses?day=5
```

Nelze ale filtrovat na základě cizích klíčů, ani nastavit podmínku (větší než apod.). Při špatně provedeném dotazu může výsledek skončit chybou sandmanu2, což jsem nahlásil autorovi.

Řazení

Je možné použít parametr `sort` pro zvolení položky, podle které se budou výsledky řadit. Není však možné zvolit směr řazení. Řazení lze kombinovat se stránkováním, ale ne s parametrem `limit`.

```
GET /courses?page=1&sort=starts_at
```

Vyjednávání o obsahu

Není v sandmanu2 podporováno.

Rozcestník

Není v sandmanu2 podporován.

3.6.10 Další poznámky

Pokud máme kontrolu nad databází, nabízí sandman2 jednoduchý automatický způsob, jak vytvořit API alespoň částečně ve stylu REST. Pokud však potřebujeme data prezentovat trochu jiným způsobem, začne nám sandman2 házet pomyslné klacky pod nohy a základní výhoda – tedy automatické vytvoření API – přestane hrát velkou roli.

3.6.11 Kompletní implementace

Kompletní implementaci REST API pro rozvrhová data ÚTVS ČVUT ve frameworku sandman2 najdete na přiloženém médiu a na adrese:

`https://github.com/hroncok/utvsapi-sandman`

3.7 Souhrn

Žádná ze čtyř implementací se neobešla bez komplikací, neexistuje tedy žádný pomyslný vítěz. U Django REST frameworku, Eve a ripoza se jednalo o nedostatky, které by se pravděpodobně daly, s rozumným množstvím úsilí, vyřešit přispěním do samotných frameworků či dalších použitých knihoven. Framework sandamn2 zaostával natolik, že jej nemohu doporučit.

Kapitola 4

Měření odezvy

Pro měření jsem použil nástroj *ab*, určený pro měření odezvy HTTP serverů [4]. K běhu testovaných služeb jsem použil Gunicorn, HTTP server napsaný v Pythonu [57].

U měření zaměřených na autentizaci a autorizaci jsem vynechal implementaci v sandmanu2, jelikož ta příslušné části neobsahuje. Autentizaci jsem prováděl proti falešnému OAAS z vlastního modulu `utvsapitoken`, který běžel na stejném počítači jako testované služby. U ostatních měření jsem autentizaci i autorizaci vypnul, aby neovlivňovala měření.

Data byla při měření dostupná v MariaDB databázi běžící na stejném počítači jako testované služby.

HTTP server byl spuštěn s dvěma vlákny. Měřící nástroj *ab* službu testoval pěti tisíci požadavky, v dávkách po jednom stu, kde jedna dávka vždy probíhala současně.

Zde prezentované měření rychlosti odezvy bylo prováděno na konkrétních implementacích popsaných v kapitole *Implementace*. Vzhledem k tomu, že jednotlivé měřené implementace jsou netriviální, nelze zde prezentované výsledky v žádném případě generalizovat na celý použitý framework.

U jednotlivých měření uvádím příklad požadavku. Pro jednotlivé implementace se tyto požadavky mohou mírně lišit, ale pro přehlednost uvádím pouze jeden.

Záznamy z měření s kompletními požadavky, výstupy a naměřenými rychlostmi jsou dostupné na přiloženém médiu. Testovací skripty jsou také dostupné na přiloženém médiu a na adrese:

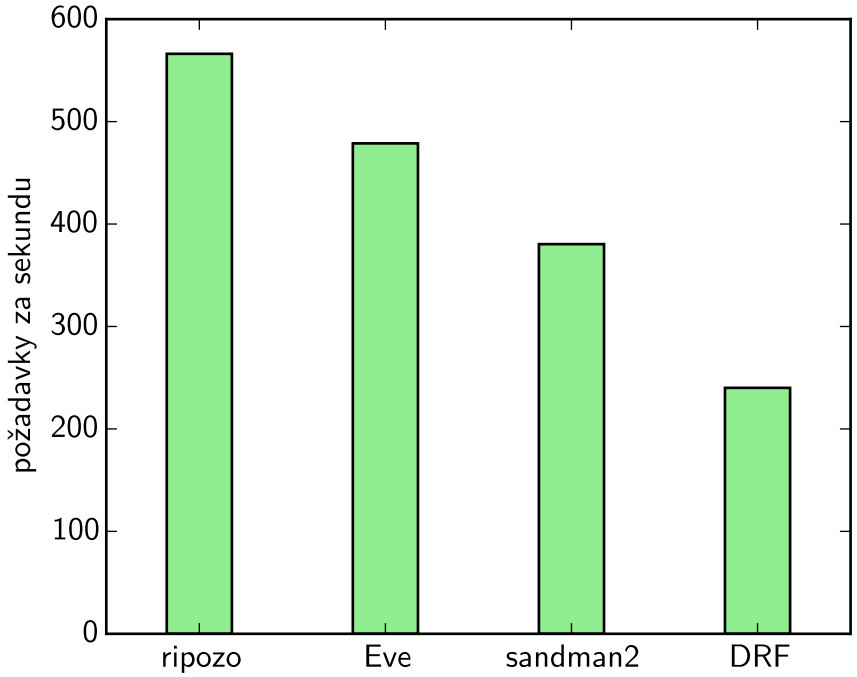
`https://github.com/hroncok/utvsapi-benchamrk`

4.1 Zobrazení jedné položky

Při tomto měření byl testován požadavek na jednu položku:

```
GET /enrollments/25563/
```

Jak můžete vidět z grafu 4.1, nejrychlejší je zde implementace v ripozu a nejpomalejší v Django REST frameworku.



Obrázek 4.1: Rychlost: Jedna položka

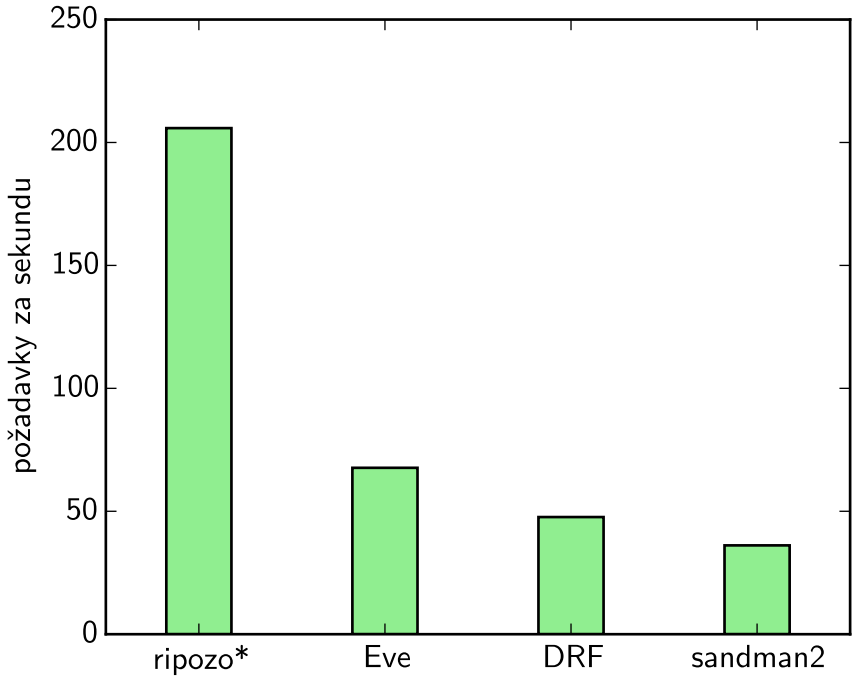
4.2 Zobrazení seznamu položek

Při tomto měření byl testován požadavek na jednu stránku seznamu položek o délce dvacet:

```
GET /enrollments/?page_size=20
```

Je třeba poznamenat, že ripozo v seznamu uvádí jen odkazy na jednotlivé položky a ostatní frameworky serializují všech dvacet objektů. Bohužel ripozo jinou možnost nenabízí, tento test má tedy méně vypovídající hodnotu. Proto je v grafu u ripoza hvězdička.

Jak můžete vidět z grafu 4.2, nejrychlejší je zde nepřekvapivě právě implementace v ripozu, nejpomalejší pak v sandmanu2.



Obrázek 4.2: Rychlost: Seznam položek

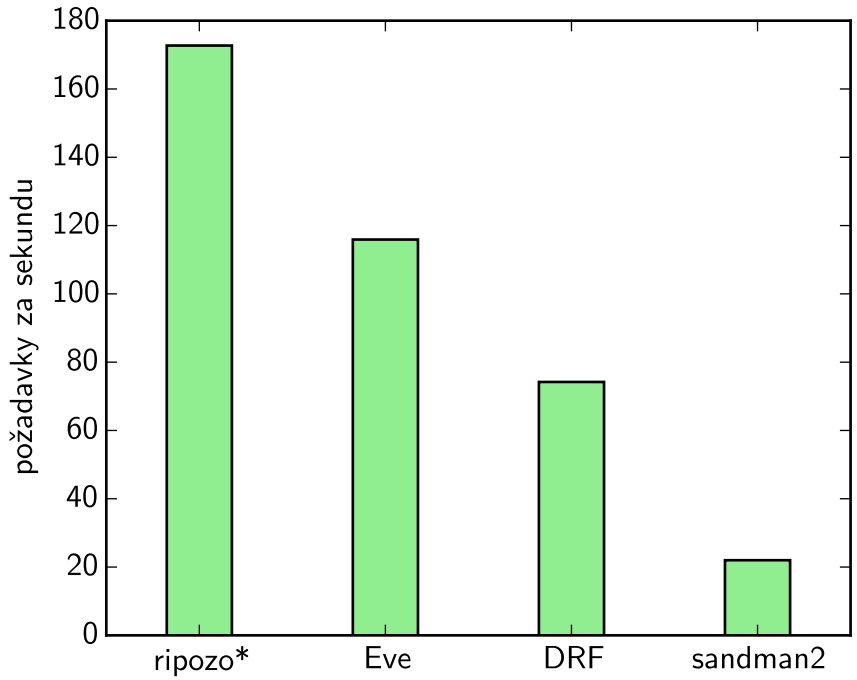
4.3 Filtrování seznamu položek

Při tomto měření byl testován požadavek na seznam kurzů, které probíhají v pátek, a velikost byla opět omezena na dvacet:

```
GET /courses/?page_size=20&day=5
```

Platí stejná poznámka jako u minulého měření – ripozo je zde ve značné výhodě. V grafu je proto označeno hvězdičkou.

Jak můžete vidět z grafu 4.3, nejrychlejší je opět implementace v ripozu, ale již nemá takový náskok, nejpomalejší je opět implementace v sandmanu2.



Obrázek 4.3: Rychlost: Filtrovaný seznam položek

4.4 Jednoduchá autorizace

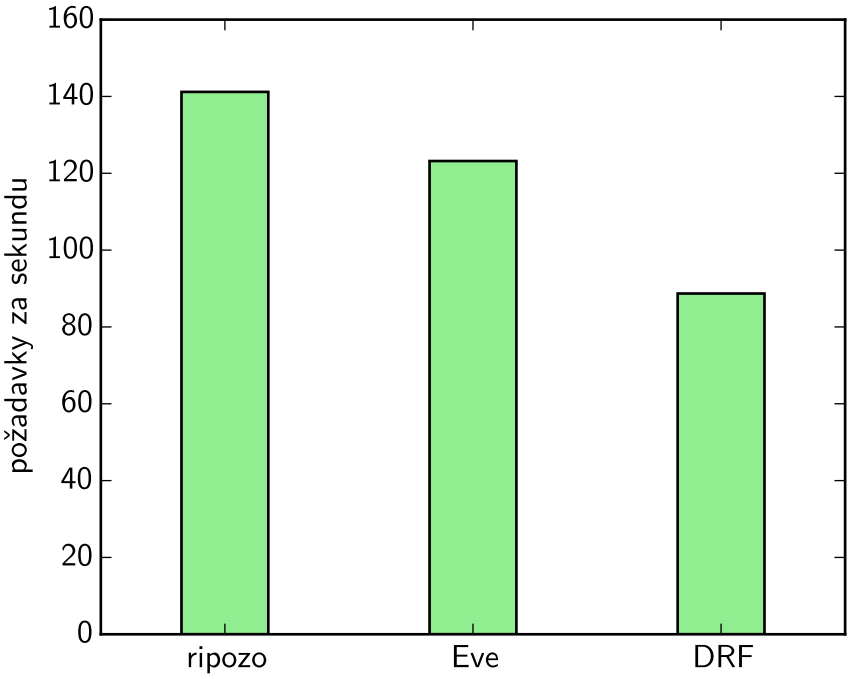
Při tomto měření byl testován požadavek na jednu položku z jiného zdroje než `/enrollments/`, tedy u zdroje, kde je autorizační logika jednodušší:

```
GET /courses/1/                    Authorization: Bearer ...
```

Jak můžete vidět z grafu 4.4, nejrychlejší je opět implementace v ripozu (zde už *není* zvýhodněný), nejpomalejší implementace v Django REST frameworku.

4.5 Komplexní autorizace

Při tomto měření byl testován požadavek na jednu položku ze zdroje, kde je autorizační logika komplexní:



Obrázek 4.4: Rychlost: Jedna položka s jednoduchou autorizací

```
GET /enrollments/25563/          Authorization: Bearer ...
```

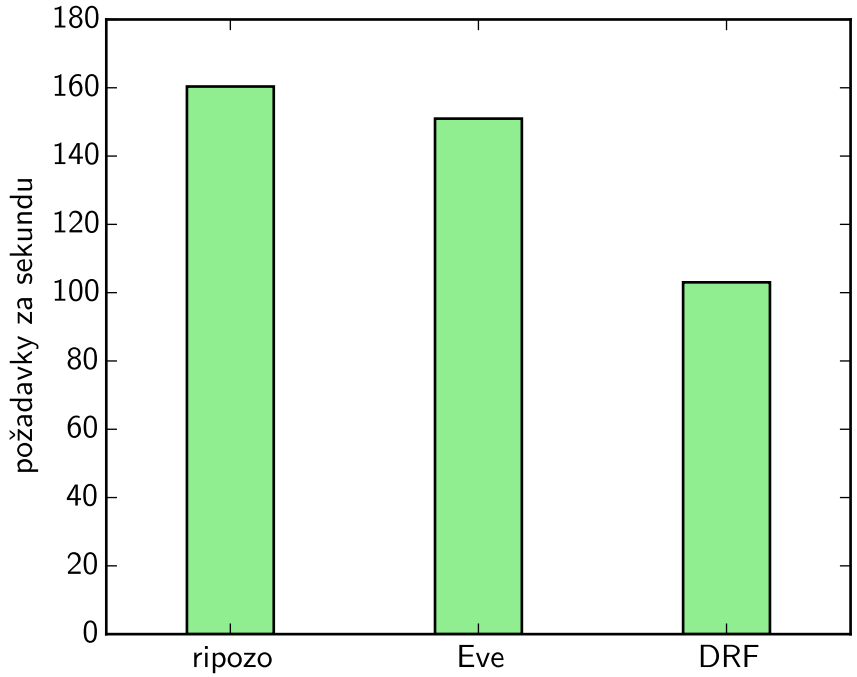
Byly provedeny tři měření, pokaždé s jiným druhem tokenu (studentský, zaměstnanecký a „všemocný“). Vzhledem k tomu, že se jednotlivé výsledky lišily jen o malou aditivní konstantu, prezentuji zde průměr z těchto tří měření.

Jak můžete vidět z grafu 4.5, nejrychlejší je opět implementace v ripozu, nejpomalejší pak implementace v Django REST frameworku.

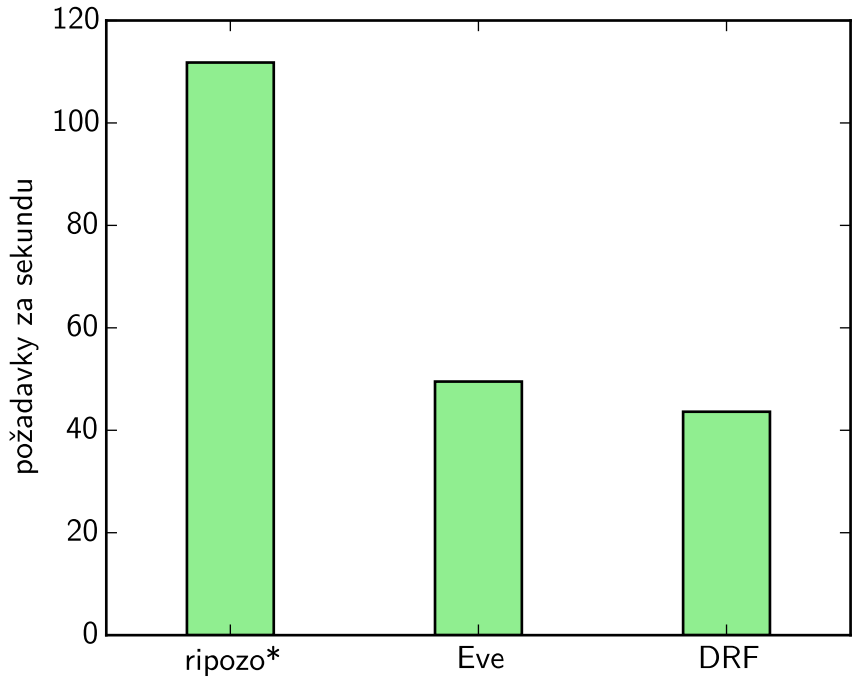
Provedl jsem i měření pro seznam položek s komplexní autorizací:

```
GET /enrollments/?page_size=20   Authorization: Bearer ...
```

Zde se výsledky lišily podle toho, jestli se jednalo o studentský token či nikoliv. V grafu 4.6 je zobrazen průměr pro zaměstnanecký a všemocný token; nejrychlejší je implementace v ripozu, která je zde opět ve velké výhodě, protože se jedná o seznam položek. Implementace v Django REST frameworku je mírně pomalejší než implementace v Eve.

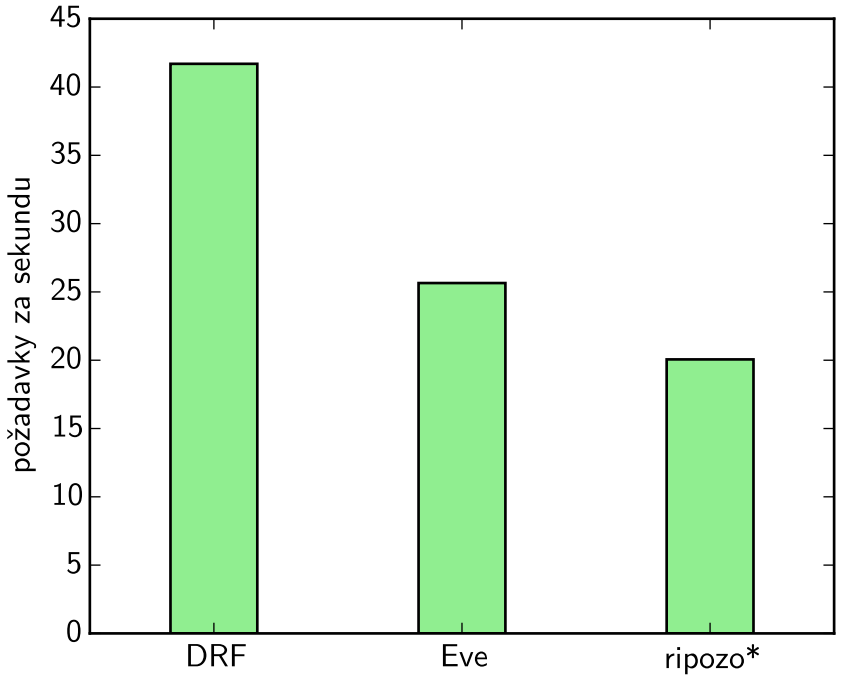


Obrázek 4.5: Rychlost: Jedna položka s komplexní autorizací



Obrázek 4.6: Rychlost: Seznam položek s komplexní autorizací (nestudent)

V grafu 4.7 jsou vidět výsledky pro studentský token. Zde je ripozo pořád ve výhodě, ale přesto zaostává za nejrychlejším Django REST frameworkem i druhým Eve.



Obrázek 4.7: Rychlost: Seznam položek s komplexní autorizací (student)

Vzhledem k rozdílným výsledkům pro různé druhy požadavků se zdráhám z měření vyvozovat nějaké závěry.

Závěr

Open-source knihoven a frameworků pro Python, které umožňují vytvářet RESTful API, je opravdu mnoho. V této práci jsem prozkoumal osmnáct takových frameworků, v kapitole *Frameworky pro RESTful API*. Zkoumal jsem především úroveň podpory HATEOAS a řízení přístupových práv.

Na základě stanovených kritérií jsem vybral čtyři frameworky, ve kterých jsem implementoval ukázkovou službu pro účely podrobnějšího zkoumání.

Návrhem ukázkové služby jsem se zabýval v kapitole *Návrh API pro rozvrhová data ÚTVS ČVUT*.

Různé aspekty implementace REST služby ve frameworcích Django REST framework, Eve, ripozo a sandman2 jsem popsal v kapitole *Implementace*. Během implementace služeb jsem ve frameworcích narazil na různý chyby, které jsem autorům nahlásil a v některých případech i navrhl úpravy. Jejich seznam najdete v dodatku B.

Implementované služby jsem podrobil testování rychlosti v kapitole *Měření odezvy*. Z měření nevzešel jasný vítěz.

Možnosti dalšího rozvoje

Některé zkoumané frameworky trpěly nedostatky, které by v budoucnu bylo možné ve spolupráci s autory těchto frameworků opravit.

Zkoumaný framework ripozo umožňuje napojení na různé webové frameworky, bylo by proto zajímavé naprogramovat napojení na nějaký zatím nepodporovaný rychlý webový framework, jako například Falcon.

Zdroje

1. ALSTOTT, Jeff. *PyPI download counts seem unrealistic* [online]. 2012 [cit. 2016-04-02]. Dostupný z WWW: ⟨http://stackoverflow.com/questions/9648015/pypi-download-counts-seem-unrealistic⟩.
2. AMUNDSEN, Mike. *Collection+JSON: Hypermedia Type* [online]. 2013 [cit. 2016-04-03]. Dostupný z WWW: ⟨http://amundsen.com/media-types/collection/⟩.
3. APACHE SOFTWARE FOUNDATION. *Apache License: Version 2.0* [online]. 2004 [cit. 2016-04-08]. Dostupný z WWW: ⟨http://www.apache.org/licenses/LICENSE-2.0⟩.
4. APACHE SOFTWARE FOUNDATION. *ab: Apache HTTP server benchmarking tool* [online]. 2016 [cit. 2016-05-05]. Dostupný z WWW: ⟨https://httpd.apache.org/docs/2.4/programs/ab.html⟩.
5. BERNSTEIN, Michael R. *Morepath logo draft* [online]. 2014 [cit. 2016-04-09]. Dostupný z WWW: ⟨https://github.com/morepath/morepath/issues/180#issuecomment-65237866⟩.
6. BRANDICTED. *Ramses* [online]. 2015 [cit. 2016-04-22]. Dostupný z WWW: ⟨https://ramses.readthedocs.org/⟩.
7. BRANDICTED. *Nefertari documentation: Configuring Models* [online]. 2016 [cit. 2016-04-22]. Dostupný z WWW: ⟨https://nefertari.readthedocs.org/en/stable/models.html⟩.

8. BRANDICTED. *Nefertari documentation: Configuring Views* [online]. 2016 [cit. 2016-04-22]. Dostupný z WWW: ⟨https://nefertari.readthedocs.org/en/stable/views.html⟩.
9. BRANDICTED. *Nefertari documentation: Why Nefertari: Vision* [online]. 2016 [cit. 2016-04-22]. Dostupný z WWW: ⟨https://nefertari.readthedocs.org/en/stable/why.html#vision⟩.
10. BRANDICTED. *Nefertari documentation: Authentication & Security* [online]. 2016 [cit. 2016-04-22]. Dostupný z WWW: ⟨https://nefertari.readthedocs.org/en/stable/auth.html⟩.
11. BRANDICTED. *Nefertari, queen of APIs* [online]. 2016 [cit. 2016-04-22]. Dostupný z WWW: ⟨https://nefertari.readthedocs.org/⟩.
12. BROOKS, Gray. *Document further query parameters: Issue #30* [online]. 2016 [cit. 2016-04-29]. Dostupný z WWW: ⟨https://github.com/jeffknupp/sandman2/issues/30⟩.
13. BURKE, Kevin; CONROY, Kyle; HORN, Ryan et al. *Flask-RESTful: Flask-RESTful documentation* [online]. 2015 [cit. 2016-04-09]. Dostupný z WWW: ⟨https://flask-restful.readthedocs.org/en/0.3.5/⟩.
14. BURKE, Kevin; CONROY, Kyle; HORN, Ryan et al. *Flask-RESTful documentation: Quickstart: Full Example* [online]. 2015 [cit. 2016-04-09]. Dostupný z WWW: ⟨https://flask-restful.readthedocs.org/en/0.3.5/quickstart.html#full-example⟩.
15. CODECOV. *falconry/falcon@master* [online]. 2016 [cit. 2016-04-08]. Dostupný z WWW: ⟨https://codecov.io/github/falconry/falcon?branch=master⟩.
16. CREATIVE COMMONS. *CC0 1.0 Universal (CC0 1.0): Public Domain Dedication* [online]. [cit. 2016-04-01]. Dostupný z WWW: ⟨https://creativecommons.org/publicdomain/zero/1.0/⟩.
17. CROSLEY, Timothy Edmund. *hug: Exposing our API as an HTTP micro-service* [online]. 2016 [cit. 2016-04-08]. Dostupný z WWW: ⟨http://www.hug.rest/website/quickstart⟩.
18. CROSLEY, Timothy Edmund. *hug: Embrace the APIs of the future* [online]. 2016 [cit. 2016-04-08]. Dostupný z WWW: ⟨http://www.hug.rest/⟩.

19. CROSLEY, Timothy Edmund et al. *hug: Embrace the APIs of the future. Hug aims to make developing APIs as simple as possible, but no simpler* [online]. 2016 [cit. 2016-04-08]. Dostupný z WWW: ⟨https://github.com/timothycrosley/hug⟩.

20. CROSLEY, Timothy Edmund. *hug logo* [online]. 2016 [cit. 2016-04-08]. Dostupný z WWW: ⟨https://github.com/timothycrosley/hug/blob/develop/artwork/logo.png⟩.

21. CROSLEY, Timothy Edmund; WAGNER, Ian et al. *hug/authentication.py at develop: timothycrosley/hug* [online]. 2016 [cit. 2016-04-08]. Dostupný z WWW: ⟨https://github.com/timothycrosley/hug/blob/develop/hug/authentication.py⟩.

22. DANIAL, Al. *CLOC: Count Lines of Code* [online]. 2016 [cit. 2016-04-07]. Dostupný z WWW: ⟨http://cloc.sourceforge.net/⟩.

23. DAVIS, Ian; STEINER, Thomas; HORS, Arnaud J Le. *RDF 1.1 JSON Alternate Serialization (RDF/JSON)* [online]. 2013 [cit. 2016-05-06]. Dostupný z WWW: ⟨https://dvcs.w3.org/hg/rdf/raw-file/default/rdf-json/index.html⟩.

24. DJANGO SOFTWARE FOUNDATION. *Django's release process: Deprecation policy* [online]. 2016 [cit. 2016-04-10]. Dostupný z WWW: ⟨https://docs.djangoproject.com/en/dev/internals/release-process/#deprecation-policy⟩.

25. DOTMOBO; CHANTREUX, Marc. *Spore: Specification to a POrtable Rest Environment* [online]. 2015 [cit. 2016-04-03]. Dostupný z WWW: ⟨http://spore.github.io/⟩.

26. FAASSEN, Martijn. *REST: Morepath docuemntation* [online]. 2014 [cit. 2016-04-09]. Dostupný z WWW: ⟨https://morepath.readthedocs.org/en/latest/rest.html⟩.

27. FAASSEN, Martijn; BERNSTEIN, Michael R.; KRIENBÜHL, Denis. *Morepath: Super Powered Python Web Framework* [online]. 2014 [cit. 2016-04-09]. Dostupný z WWW: ⟨https://morepath.readthedocs.org/⟩.

28. FAASSEN, Martijn; BERNSTEIN, Michael R.; REES, Reinout van. *Superpowers: Morepath docuemntation* [online]. 2014 [cit. 2016-04-09]. Dostupný z WWW: ⟨https://morepath.readthedocs.org/en/latest/superpowers.html⟩.

29. FAASSEN, Martijn; HULSKI, Henri. *Security: Morepath docuemntation* [online]. 2014 [cit. 2016-04-09]. Dostupný z WWW: ⟨https://morepath.readthedocs.org/en/latest/security.html⟩.

30. FAASSEN, Martijn; KINGMA, Franklin. *History of Morepath: Morepath docuemntation* [online]. 2014 [cit. 2016-04-09]. Dostupný z WWW: ⟨https://morepath.readthedocs.org/en/latest/history.html⟩.

31. FIELDING, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. 2000. Dostupný také z WWW: ⟨http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm⟩. Irvine, 2000. Disertační práce. University of California, Irvine.

32. FLYNN, Charles et al. *Eve-docs* [online]. 2014 [cit. 2016-05-01]. Dostupný z WWW: ⟨https://github.com/charlesflynn/eve-docs⟩.

33. FOWLER, Martin. *Patterns of Enterprise Application Architecture*. Boston, MA: Addison-Wesley Professional, 2002. ISBN 978-0-32112-742-6.

34. FREE SOFTWARE FOUNDATION, Inc. *GNU General Public License* [online]. 1991 [cit. 2016-04-01]. Dostupný z WWW: ⟨http://www.gnu.org/licenses/old-licenses/gpl-2.0.html⟩.

35. FREE SOFTWARE FOUNDATION, Inc. *GNU Affero General Public License* [online]. 2007 [cit. 2016-04-01]. Dostupný z WWW: ⟨http://www.gnu.org/licenses/agpl-3.0.html⟩.

36. FREE SOFTWARE FOUNDATION, Inc. *GNU General Public License* [online]. 2007 [cit. 2016-04-01]. Dostupný z WWW: ⟨http://www.gnu.org/licenses/gpl-3.0.html⟩.

37. FREE SOFTWARE FOUNDATION, Inc. *GNU Lesser General Public License* [online]. 2007 [cit. 2016-04-01]. Dostupný z WWW: ⟨http://www.gnu.org/licenses/lgpl.html⟩.

38. FREE SOFTWARE FOUNDATION, Inc. *What is Copyleft?* [online]. 2015 [cit. 2016-04-01]. Dostupný z WWW: ⟨https://www.gnu.org/licenses/copyleft.html.en⟩.

39. GABORY, Yohann. *Python Rest Api Framework documentation: Representing relations* [online]. 2013 [cit. 2016-04-15]. Dostupný z WWW: ⟨http://python-rest-framework.readthedocs.org/en/latest/tutorial/represent_related.html⟩.

40. GABORY, Yohann. *Python Rest Api Framework documentation: Tutorial: building an adressebook API* [online]. 2013 [cit. 2016-04-15]. Dostupný z WWW: ⟨http://python-rest-framework.readthedocs.org/en/latest/tutorial.html⟩.

41. GABORY, Yohann. *Python Rest Api Framework documentation: Linking ressource together* [online]. 2013 [cit. 2016-04-15]. Dostupný z WWW: ⟨http://python-rest-framework.readthedocs.org/en/latest/tutorial/related_ressources.html⟩.

42. GABORY, Yohann. *Python Rest Api Framework documentation: Authentication and Authorization: Protecting your API* [online]. 2013 [cit. 2016-04-15]. Dostupný z WWW: ⟨http://python-rest-framework.readthedocs.org/en/latest/tutorial/protect_api.html⟩.

43. GABORY, Yohann. *Python Rest Api Framework's documentation* [online]. 2013 [cit. 2016-04-15]. Dostupný z WWW: ⟨https://python-rest-framework.readthedocs.org/⟩.

44. GABORY, Yohann. *What is Python REST API Framework: Architecture* [online]. 2013 [cit. 2016-04-15]. Dostupný z WWW: ⟨http://python-rest-framework.readthedocs.org/en/latest/introduction.html#architecture⟩.

45. GEBHARDT, Dan et al. *JSON API: A specification for building APIs in JSON* [online]. 2016 [cit. 2016-05-06]. Dostupný z WWW: ⟨http://jsonapi.org/⟩.

46. GITHUB, Inc. *Contributors to nicolaiarocci/eve* [online]. 2016 [cit. 2016-04-07]. Dostupný z WWW: ⟨https://github.com/nicolaiarocci/eve/graphs/contributors⟩.

47. GITHUB, Inc. *Contributors to tomchristie/django-rest-framework* [online]. 2016 [cit. 2016-04-05]. Dostupný z WWW: ⟨https://github.com/tomchristie/django-rest-framework/graphs/contributors⟩.

48. GITHUB, Inc. *GitHub help: About Stars* [online]. 2016 [cit. 2016-04-02]. Dostupný z WWW: ⟨https://help.github.com/articles/about-stars/⟩.

49. GRIFFITHS, Kurt et al. *Falcon: The minimalist Python WSGI framework* [online]. 2016 [cit. 2016-04-08]. Dostupný z WWW: ⟨http://falconframework.org/⟩.

50. GRIFFITHS, Kurt. *Falcon Logo* [online]. 2016 [cit. 2016-04-08]. Dostupný z WWW: ⟨http://falconframework-226b.kxcdn.com/img/logo.png⟩.

51. HART, Chris. *Create a REST API in Minutes With Pyramid and Ramses* [online]. 2015 [cit. 2016-04-22]. Dostupný z WWW: ⟨https://realpython.com/blog/python/create-a-rest-api-in-minutes-with-pyramid-and-ramses/⟩.

52. HELLER, Martin. *REST and CRUD: the Impedance Mismatch* [online]. 2007 [cit. 2016-05-06]. Dostupný z WWW: ⟨http://www.infoworld.com/article/2640739/application-development/rest-and-crud--the-impedance-mismatch.html⟩.

53. HERMANNSBLUM. *Extended Documentation: Pull Request #25* [online]. 2015 [cit. 2016-05-01]. Dostupný z WWW: ⟨https://github.com/charlesflynn/eve-docs/pull/25⟩.

54. HOCEVAR, Sam. *DO WHAT THE FUCK YOU WANT TO PUBLIC LICENSE* [online]. 2004 [cit. 2016-04-01]. Dostupný z WWW: ⟨http://www.wtfpl.net/txt/copying/⟩.

55. HOLOVATY, Adrian; KAPLAN-MOSS, Jacob. The MTV (or MVC) Development Pattern. In: *The Definitive Guide to Django: Web Development Done Right*. Berkeley, CA: Apress, 2008. S. 72–74. ISBN 978-1-59059-725-5. Dostupný také z WWW: ⟨http://www.djangobook.com/en/2.0/chapter05.html#the-mtv-or-mvc-development-pattern⟩.

56. HORN, Ryan. *flask-restful.png* [online]. 2012 [cit. 2016-04-09]. Dostupný z WWW: ⟨https://github.com/flask-restful/flask-restful/blob/master/docs/_static/flask-restful.png⟩.

57. CHESNEAU, Benoit et al. *Gunicorn: Python WSGI HTTP Server for UNIX* [online]. 2016 [cit. 2016-05-05]. Dostupný z WWW: ⟨http://gunicorn.org/⟩.

58. CHHANTYAL, Nar et al. *Python 3 Readiness* [online]. 2016 [cit. 2016-04-02]. Dostupný z WWW: ⟨http://py3readiness.org/⟩.

59. CHRISTIE, Tom. *Flask API: Browsable Web APIs for Flask* [online]. 2014 [cit. 2016-04-09]. Dostupný z WWW: ⟨http://www.flaskapi.org/⟩.

60. CHRISTIE, Tom. *Kickstarting Django REST framework 3* [online]. 2014 [cit. 2016-04-03]. Dostupný z WWW: ⟨http://www.django-rest-framework.org/topics/kickstarter-announcement/⟩.

61. CHRISTIE, Tom. *Django REST framework: Quickstart* [online]. 2016 [cit. 2016-04-03]. Dostupný z WWW: ⟨http://www.django-rest-framework.org/tutorial/quickstart/⟩.

62. CHRISTIE, Tom. *Django REST framework* [online]. 2016 [cit. 2016-04-03]. Dostupný z WWW: ⟨http://www.django-rest-framework.org/⟩.

63. CHRISTIE, Tom. *Django REST framework: REST, Hypermedia & HATEOAS* [online]. 2016 [cit. 2016-04-03]. Dostupný z WWW: ⟨http://www.django-rest-framework.org/topics/rest-hypermedia-hateoas/⟩.

64. CHRISTIE, Tom. *Django REST framework: Authentication* [online]. 2016 [cit. 2016-04-05]. Dostupný z WWW: ⟨http://www.django-rest-framework.org/api-guide/authentication/⟩.

65. CHRISTIE, Tom. *Django REST framework: Permissions* [online]. 2016 [cit. 2016-04-05]. Dostupný z WWW: ⟨http://www.django-rest-framework.org/api-guide/permissions/⟩.

66. CHRISTIE, Tom. *Django REST framework: Documenting your API* [online]. 2016 [cit. 2016-05-05]. Dostupný z WWW: ⟨http://www.django-rest-framework.org/topics/documenting-your-api/⟩.

67. CHRISTIE, Tom et al. *Flask API* [online]. 2016 [cit. 2016-04-09]. Dostupný z WWW: ⟨https://github.com/tomchristie/flask-api⟩.

68. IAROCCI, Nicola. *Eve-OAuth2* [online]. 2015 [cit. 2016-04-07]. Dostupný z WWW: ⟨https://github.com/nicolaiarocci/eve-oauth2⟩.

69. IAROCCI, Nicola. *Eve: Quickstart* [online]. 2016 [cit. 2016-04-05]. Dostupný z WWW: ⟨http://python-eve.org/quickstart.html⟩.

70. IAROCCI, Nicola. *Eve: Features: HATEOAS* [online]. 2016 [cit. 2016-04-07]. Dostupný z WWW: ⟨http://python-eve.org/features.html#hateoas⟩.

71. IAROCCI, Nicola. *Eve: Python REST API Framework* [online]. 2016 [cit. 2016-04-05]. Dostupný z WWW: ⟨http://python-eve.org/⟩.

72. IAROCCI, Nicola. *Eve: REST API for Humans* [online]. 2016 [cit. 2016-04-07]. Dostupný z WWW: ⟨http://python-eve.org/rest_api_for_humans.html⟩.

73. IAROCCI, Nicola. *Eve: Authentication and Authorization* [online]. 2016 [cit. 2016-04-07]. Dostupný z WWW: ⟨http://python-eve.org/authentication.html⟩.

74. JIRŮTKA, Jakub. *Usermap API* [online]. 2015 [cit. 2016-04-29]. Dostupný z WWW: ⟨https://rozvoj.fit.cvut.cz/Main/usermap-api⟩.

75. JIRŮTKA, Jakub. *OAuth 2.0: Autorizační server* [online]. 2016 [cit. 2016-04-29]. Dostupný z WWW: ⟨https://rozvoj.fit.cvut.cz/Main/oauth2#HAutoriza10DnEDserver⟩.

76. JIRŮTKA, Jakub; HRONČOK, Miroslav. *Rozvrhy ÚTVS (databáze)* [online]. 2014 [cit. 2016-04-22]. Dostupný z WWW: ⟨https://rozvoj.fit.cvut.cz/Main/rozvrhy-utvs-db⟩.

77. JONES, Michael B.; HARDT, Dick. *RFC 6750: The OAuth 2.0 Authorization Framework: Bearer Token Usage* [online]. 2012 [cit. 2016-05-02]. Dostupný z WWW: ⟨https://tools.ietf.org/html/rfc6750#section-2.1⟩.

78. JSON-LD COMMUNITY GROUP. *JSON-LD: JSON for Linking Data* [online]. [cit. 2016-04-03]. Dostupný z WWW: ⟨http://json-ld.org/⟩.

79. KELLY, Mike. *HAL: Hypertext Application Language* [online]. 2011 [cit. 2016-04-03]. Dostupný z WWW: ⟨http://stateless.co/hal_specification.html⟩.

80. KNUPP, Jeff. *sandman documentaion: Authentication* [online]. 2013 [cit. 2016-04-22]. Dostupný z WWW: ⟨https://sandman.readthedocs.org/en/latest/authentication.html⟩.

81. KNUPP, Jeff. *sandman documentation* [online]. 2013 [cit. 2016-04-22]. Dostupný z WWW: ⟨https://sandman.readthedocs.org/⟩.

82. KNUPP, Jeff. *admin-view.png* [online]. 2014 [cit. 2016-04-22]. Dostupný z WWW: ⟨http://jeffknupp.com/images/admin-view.png⟩.

83. KNUPP, Jeff. *sandman2 documentation: Quickstart* [online]. 2014 [cit. 2016-04-22]. Dostupný z WWW: ⟨http://pythonhosted.org/sandman2/quickstart.html⟩.

84. KNUPP, Jeff et al. *sandman* [online]. 2016 [cit. 2016-04-22]. Dostupný z WWW: ⟨https://github.com/jeffknupp/sandman⟩.

85. KNUPP, Jeff et al. *sandman2* [online]. 2016 [cit. 2016-04-22]. Dostupný z WWW: ⟨https://github.com/jeffknupp/sandman2⟩.

86. KRAWCZYK, H.; BELLARE, M.; CANETTI, R. *RFC 2104: HMAC: Keyed-Hashing for Message Authentication* [online]. 1997 [cit. 2016-04-07]. Dostupný z WWW: ⟨ftp://ftp.isi.edu/in-notes/rfc2104.txt⟩.

87. LANTHALER, Markus. *Hydra: Hypermedia-Driven Web APIs* [online]. 2013 [cit. 2016-04-03]. Dostupný z WWW: ⟨http://www.markus-lanthaler.com/hydra/⟩.

88. LAWLOR, Chris. *How to show foreign key in django-piston rest output instead of related object data* [online]. 2011 [cit. 2016-04-10]. Dostupný z WWW: ⟨http://stackoverflow.com/a/7615732/1839451⟩.

89. LEWIS, Ian. *Mixins and Python* [online]. 2013 [cit. 2016-05-06]. Dostupný z WWW: ⟨https://www.ianlewis.org/en/mixins-and-python⟩.

90. LINDSLEY, Daniel. *Tastypie: RESTful APIs for Django* [online]. 2013 [cit. 2016-04-20]. Dostupný z WWW: ⟨http://tastypieapi.org/⟩.

91. LINDSLEY, Daniel. *restless* [online]. 2014 [cit. 2016-04-17]. Dostupný z WWW: ⟨https://restless.readthedocs.org/⟩.

92. LINDSLEY, Daniel. *restless documentation: Philosophy Behind Restless* [online]. 2014 [cit. 2016-04-17]. Dostupný z WWW: ⟨http://restless.readthedocs.org/en/latest/philosophy.html⟩.

93. LINDSLEY, Daniel et al. *Tastypie documentation: Getting Started with Tastypie* [online]. 2016 [cit. 2016-04-21]. Dostupný z WWW: ⟨https://django-tastypie.readthedocs.org/en/latest/tutorial.html⟩.

94. LINDSLEY, Daniel et al. *Tastypie documentation: Authentication* [online]. 2016 [cit. 2016-04-21]. Dostupný z WWW: ⟨https://django-tastypie.readthedocs.org/en/latest/authentication.html⟩.

95. LINDSLEY, Daniel et al. *Tastypie documentation: Authorization* [online]. 2016 [cit. 2016-04-21]. Dostupný z WWW: ⟨https://django-tastypie.readthedocs.org/en/latest/authorization.html⟩.

96. LINDSLEY, Daniel; FARWELL, Corey; LIAO, Mission. *restless* [online]. 2015 [cit. 2016-04-17]. Dostupný z WWW: ⟨https://github.com/toastdriven/restless⟩.

97. MARTIN, Tim. *ripozo-logo.png* [online]. 2015 [cit. 2016-04-19]. Dostupný z WWW: ⟨https://github.com/vertical-knowledge/ripozo/blob/master/logos/ripozo-logo.png⟩.

98. MARTIN, Tim. *ripozo* [online]. 2016 [cit. 2016-04-19]. Dostupný z WWW: ⟨https://ripozo.readthedocs.org/⟩.

99. MARTIN, Tim. *ripozo documentation: Preprocessors and Postprocessors* [online]. 2016 [cit. 2016-04-19]. Dostupný z WWW: ⟨https://ripozo.readthedocs.org/en/latest/topics/processors.html⟩.

100. MOZILLA FOUNDATION. *Mozilla Public License, version 2.0* [online]. 2012 [cit. 2016-04-03]. Dostupný z WWW: ⟨https://www.mozilla.org/en-US/MPL/2.0/⟩.

101. MOZILLA SERVICES. *Cornice: A REST framework for Pyramid* [online]. 2011 [cit. 2016-04-03]. Dostupný z WWW: ⟨https://cornice.readthedocs.org/⟩.

102. MOZILLA SERVICES. *Cornice Documentation: QuickStart for people in a hurry* [online]. 2011 [cit. 2016-04-03]. Dostupný z WWW: ⟨https://cornice.readthedocs.org/en/latest/quickstart.html⟩.

103. MOZILLA SERVICES. *Cornice Documentation: SPORE support* [online]. 2011 [cit. 2016-04-03]. Dostupný z WWW: ⟨https://cornice.readthedocs.org/en/latest/spore.html⟩.

104. NØHR, Jesper. *New Piston logo* [online]. 2009 [cit. 2016-04-09]. Dostupný z WWW: ⟨https://bitbucket.org/jespern/django-piston/wiki/piston.png⟩.

105. NØHR, Jesper. *Piston* [online]. 2013 [cit. 2016-04-09]. Dostupný z WWW: ⟨https://bitbucket.org/jespern/django-piston/wiki/Home⟩.

106. NØHR, Jesper et al. *Piston Documentation: Authentication* [online]. 2014 [cit. 2016-04-09]. Dostupný z WWW: ⟨https://bitbucket.org/jespern/django-piston/wiki/Documentation#!authentication⟩.

107. NØHR, Jesper et al. *Piston Documentation: Anonymous Resources* [online]. 2014 [cit. 2016-04-09]. Dostupný z WWW: ⟨https://bitbucket.org/jespern/django-piston/wiki/Documentation#!anonymous-resources⟩.

108. NULLISM et al. *Framework Benchmarks* [online]. 2015 [cit. 2016-04-13]. Dostupný z WWW: ⟨https://github.com/nullism/pycnic/tree/master/benchmark⟩.

109. NULLISM. *Pycnic* [online]. 2015 [cit. 2016-04-13]. Dostupný z WWW: ⟨http://pycnic.nullism.com/images/pycnic-head.png⟩.

110. NULLISM. *Pycnic: A JSON API web framework for lazy people* [online]. 2016 [cit. 2016-04-13]. Dostupný z WWW: ⟨http://pycnic.nullism.com/⟩.

111. NULLISM. *Pycnic Docs: Example of the request object* [online]. 2016 [cit. 2016-04-13]. Dostupný z WWW: ⟨http://pycnic.nullism.com/docs/example-request.html⟩.

112. NULLISM. *Pycnic Docs: Full Authentication Example* [online]. 2016 [cit. 2016-04-13]. Dostupný z WWW: ⟨http://pycnic.nullism.com/docs/example-full-auth.html⟩.

113. OPEN SOURCE INITIATIVE. *The BSD 2-Clause License* [online]. [cit. 2016-04-01]. Dostupný z WWW: ⟨https://opensource.org/licenses/BSD-2-Clause⟩.

114. OPEN SOURCE INITIATIVE. *The BSD 3-Clause License* [online]. [cit. 2016-04-01]. Dostupný z WWW: ⟨https://opensource.org/licenses/BSD-3-Clause⟩.

115. OPEN SOURCE INITIATIVE. *The MIT License (MIT)* [online]. [cit. 2016-04-01]. Dostupný z WWW: ⟨https://opensource.org/licenses/MIT⟩.

116. PASINI, Roberto. *Eve* [online]. 2013 [cit. 2016-04-05]. Dostupný z WWW: ⟨http://python-eve.org/_static/eve-sidebar.png⟩.

117. PENG, Luo. *RESTArt documentation: Quickstart* [online]. 2015 [cit. 2016-04-16]. Dostupný z WWW: ⟨https://restart.readthedocs.org/en/latest/quickstart.html⟩.

118. PENG, Luo. *RESTArt documentation: Testing* [online]. 2015 [cit. 2016-04-17]. Dostupný z WWW: ⟨https://restart.readthedocs.org/en/latest/testing.html⟩.

119. PENG, Luo. *RESTArt documentation: Framework Integration* [online]. 2015 [cit. 2016-04-17]. Dostupný z WWW: ⟨https://restart.readthedocs.org/en/latest/framework-integration.html⟩.

120. PENG, Luo. *RESTArt* [online]. 2016 [cit. 2016-04-17]. Dostupný z WWW: ⟨https://github.com/RussellLuo/restart⟩.

121. PETERS, Tim. *PEP 20: The Zen of Python* [online]. 2004 [cit. 2016-04-01]. Dostupný z WWW: ⟨https://www.python.org/dev/peps/pep-0020/⟩.

122. PITCHER, Brad et al. *django-tastypie-oauth* [online]. 2016 [cit. 2016-04-21]. Dostupný z WWW: ⟨https://github.com/orcasgit/django-tastypie-oauth⟩.

123. PYLONS PROJECT. *Pylons Project: Pyramid* [online]. 2016 [cit. 2016-05-06]. Dostupný z WWW: ⟨http://www.pylonsproject.org/⟩.

124. PYTHON SOFTWARE FOUNDATION. *Python 3.0 Release* [online]. 2008 [cit. 2016-04-02]. Dostupný z WWW: ⟨https://www.python.org/download/releases/3.0/⟩.

125. PYTHON SOFTWARE FOUNDATION. *PSF license agreement for Python 3.5.1* [online]. 2015 [cit. 2016-04-01]. Dostupný z WWW: ⟨https://docs.python.org/3/license.html#psf-license-agreement-for-python-release⟩.

126. PYTHON SOFTWARE FOUNDATION. *PyPI: The Python Package Index* [online]. 2015 [cit. 2016-04-02]. Dostupný z WWW: ⟨https://pypi.python.org/pypi⟩.

127. REITZ, Kenneth et al. *Virtual Environments* [online]. 2015 [cit. 2016-04-02]. Dostupný z WWW: ⟨http://docs.python-guide.org/en/latest/dev/virtualenvs/⟩.

128. RONACHER, Armin. *Modular Applications with Blueprints: Flask Documentation* [online]. 2013 [cit. 2016-04-09]. Dostupný z WWW: ⟨http://flask.pocoo.org/docs/0.10/blueprints/⟩.

129. RONACHER, Armin. *Werkzeug: The Python WSGI Utility Library* [online]. 2014 [cit. 2016-05-06]. Dostupný z WWW: ⟨http://werkzeug.pocoo.org/⟩.

130. RONACHER, Armin. *Flask: A Python Microframework* [online]. 2016 [cit. 2016-05-06]. Dostupný z WWW: ⟨http://flask.pocoo.org/⟩.

131. RONACHER, Armin. *Micropackages and Open Source Trust Scaling* [online]. 2016 [cit. 2016-04-02]. Dostupný z WWW: ⟨http://lucumr.pocoo.org/2016/3/24/open-source-trust-scaling/⟩.

132. ROUSE, Margaret. *What is lightweight?* [online]. 2005 [cit. 2016-04-01]. Dostupný z WWW: ⟨http://whatis.techtarget.com/definition/lightweight⟩.

133. RVBA. *Django 1.6 incompatibility: Issue #235* [online]. 2009 [cit. 2016-04-10]. Dostupný z WWW: ⟨https://bitbucket.org/jespern/django-piston/issues/235/⟩.

134. SOSIGN INTERACTIVE. *Ramses API framework* [online]. 2016 [cit. 2016-04-15]. Dostupný z WWW: ⟨http://ramses.tech/⟩.

135. SPOLSKY, Joel. *The Law of Leaky Abstractions* [online]. 2002 [cit. 2016-05-02]. Dostupný z WWW: ⟨http://www.joelonsoftware.com/articles/LeakyAbstractions.html⟩.

136. SWIBER, Kevin et al. *Siren: a hypermedia specification for representing entities* [online]. 2015 [cit. 2016-04-19]. Dostupný z WWW: ⟨https://github.com/kevinswiber/siren⟩.

137. VOGEL, Christian et al. *RAML* [online]. 2016 [cit. 2016-04-22]. Dostupný z WWW: ⟨http://raml.org/⟩.

Dodatek A

Seznam použitých zkratek

AGPL	Affero General Public License
API	Application programming interface
ASCII	American Standard Code for Information Interchange
BSD	Berkeley Software Distribution
CLI	Command Line Interface
CRUD	Create, Retrieve, Update, Delete
CRUD+L	Create, Retrieve, Update, Delete, and List
CSRF	Cross-site Request Forgery
ČVUT	České vysoké učení technické
DRF	Django REST framework
FIT	Fakulta informačních technologií
GNU	GNU's Not Unix!
GPL	General Public License
HAL	Hypertext Application Language
HATEOAS	Hypermedia as the Engine of Application State
HMAC	Hash Message Authentication Code
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
KOS	KOmponenta Studium
LGPL	Lesser General Public License

MIT	Massachusetts Institute of Technology
MTV	Model-template-view
MVC	Model-view-controller
OAAS	OAuth 2.0 authorization server
ORM	Object-relational mapping
PDF	Portable Document Format
PEP	Python Enhancement Proposal
PRAF	Python REST API framework
RAML	RESTful API Modeling Language
RDF	Resource Description Framework
REST	Representational State Transfer
RFC	Request for Comments
SPORE	Specification to a POrtable Rest Environment
SQL	Structured Query Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
ÚTVS	Ústav tělesné výchovy a sportu
WTFPL	Do What The Fuck You Want To Public License
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

Dodatek B

Seznam nahlášených chyb a navržených úprav

Django REST framework

https://github.com/tomchristie/django-rest-framework/pull/4097

https://github.com/seebass/drf-hal-json/issues/3

Eve

https://github.com/RedTurtle/eve-sqlalchemy/issues/111

https://github.com/nicolaiarocci/eve/issues/859

https://groups.google.com/d/msg/python-eve/YFu1Xlc-4ys/RTvW4haRDwAJ

Pycnic

https://github.com/nullism/pycnic/pull/4

ripozo

https://github.com/vertical-knowledge/ripozo/issues/64

https://github.com/vertical-knowledge/ripozo-sqlalchemy/issues/9

https://github.com/vertical-knowledge/ripozo-sqlalchemy/issues/8

sandman2

`https://github.com/jeffknupp/sandman2/pull/35`

`https://github.com/jeffknupp/sandman2/issues/36`

Dodatek C

Obsah přiloženého média

- `README.md` ..... stručný popis obsahu média
- `src`
  - `utvsapi-benchmark` ..... skripty pro měření rychlosti
    - `logs` ..... záznamy z měření rychlosti
  - `utvsapi-django` ..... implementace ukázkové služby v DRF
  - `utvsapi-eve` ..... implementace ukázkové služby v Eve
  - `utvsapi-ripozo` ..... implementace ukázkové služby v ripozu
  - `utvsapi-sandman` ..... implementace ukázkové služby v sandmanu2
  - `utvsapitoken` ..... společný modul pro práci s tokenem
  - `diplomka` ..... zdrojová forma práce ve formátu Markdown a XeLaTeX
- `DP_Hroncok_Miroslav_2016.pdf` ..... text práce ve formátu PDF

Adresářová struktura C.1: Obsah přiloženého média