RASCUNHO

Capítulo 4

Os cinco passos para transformar qualquer site de dados públicos em uma API de dados abertos

Sérgio Souza Costa, Mateus Vitor Duarte Sousa, Micael Lopes da Silva

Abstract

This text presents in a practical way five steps to transform public data sites into open data ports. These steps include modeling and extracting data, developing, documenting, and deploying a REST API. As an example, the Maranhão state transparency portal was used.

Resumo

Esse texto apresenta de maneira prática, cinco passos para transformar sites com dados públicos em portas de dados abertos. Esses passos incluem a modelagem e extração dos dados, o desenvolvimento, documentação e implantação de uma API REST. Como exemplo, foi utilizado o portal da transparência do estado do Maranhão.

4.1. Introdução

No Brasil o acesso à informação sobre os recursos públicos está previsto pela Constituição de 1988 e foi reforçado através Lei de Acesso à Informação (Lei nº 12.527/2011[1]). Em resposta a esta demanda, instituições públicas tem disponibilizado um grande volume de informações públicas através de sites na Web. Contudo, muitos destes sites foram desenvolvidos para serem acessados por um usuário, e não são compreensíveis por algoritmos computacionais, impedindo assim uma busca automatizada das informações. Desta forma a extração de uma informação específica pode ser muito difícil, até impossível em alguns casos. Quando isso ocorre o interessado precisa entrar em contato com a instituição provedora, para requisitar os dados de interesse. Esse processo pode demandar um tempo maior que o esperado pelo solicitante, além de consumir tempo da instituição detentora

[1]A lei pode ser acessada integralmente em http://www.planalto.gov.br/ccivil_03/_ato2011-2014/2011/lei/l12527.htm

dos dados. Dado este cenário, muitas instituições estão migrando para o paradigma de dados abertos.

Nesse paradigma, os dados abertos precisam ser publicados de acordo com alguns princípios. Por exemplo, ser completos, primários, atuais, acessíveis, compreensíveis por máquina, não proprietários e livres de licença [Lessig 2007]. A principal vantagem é possibilitar tanto o desenvolvimento de aplicativos pela comunidade quanto suportar a utilização de algoritmos de análise de dados. Muitas destas análises seriam impraticáveis de serem realizadas sem a utilização de um algoritmo computacional para automatização da recuperação, manipulação e visualização dos dados. Segundo [Vaz 2011], atualmente espera-se que tudo que não for sigiloso esteja disponível na internet no formato de dados abertos, ou seja, não basta disponibilizar quando solicitado. O paradigma de dados abertos é muito importante para a transparência de instituições públicas, onde elas passam a disponibilizar todos os dados de modo bruto e atualizado ao invés de proverem os dados apenas quando solicitados. Deste modo, a comunidade pode usar estes dados tanto para fiscalizar quanto para gerar soluções úteis a população [Keßler and Kauppinen 2012, Daga et al. 2015, Molloy 2011, Dietrich et al. 2009, Zablith et al. 2011]

Existem alguns portais que facilitam a busca por dados. Por exemplo, no Brasil o portal `http://dados.gov.br/dataset` agrupam diversos conjunto de dados por categorias e instituições brasileiras. Não restrito a dados nacionais, a Google possui um portal específico para a busca de dados abertos: `https://toolbox.google.com/datasetsearch`. Porém, muitos dados ainda são acessíveis apenas através de páginas HTML[2] e requerem a interação com um usuário. Eles não foram projetados para serem acessados por aplicativos móveis ou *scripts* de análise de dados.

O objetivo desse texto é apresentar uma solução para expor os dados públicos como dados abertos utilizando para isso a extração de dados (ou *data scraping*). Nesse texto, será apresentado então como realizar o *scraping* dos dados e disponibilizá-los através de uma API implantada na plataforma Heroku (`https://dashboard.heroku.com/login`). Para isso, esse texto está estrutura da seguinte forma. A Seção 4.2 apresenta os pré-requisitos necessários para o desenvolvimento da API. A Seção 4.3 apresenta os 5 passos propostos: (1) identificação e modelagem dos dados, (2) extração dos dados, (3) desenvolvimento, (4) documentação e (5) implantação. A Seção 4.4 mostra algumas experimentações simples e discute algumas possibilidades de evolução. E por fim, a Seção 4.5 apresenta algumas conclusões finais.

4.2. Pré-requisitos

Esse texto tem como objetivo apresentar todos os passos de modo simples e de fácil reprodução. O nível de dificuldade dos passos dependerá dos seus conhecimentos prévios. É esperado um conhecimento básico de programação e alguma familiaridade com a linguagem de programação Python. Conhecimento básico de HTML também é desejável, principalmente para a evolução da API. Além disso, antes de seguir a execução dos cinco passos detalhados na Seção 4.3, será necessário atender os seguintes pré-requisitos:

[2]HTML do inglês HyperText Markup Language, é uma linguagem de marcação utilizada na construção de páginas na Web com diversos metadados ou *tags* e recursos que são úteis apenas para a visualização através dos navegadores.

1. Ter instalado e configurado o interpretador da Linguagem Python e o instalador de pacotes PIP (https://www.python.org/).

2. Ter uma conta na Plataforma Heroku (https://www.heroku.com/).

3. Ter uma conta no Github (https://github.com/).

4. Ter instalado o Git (https://git-scm.com/).

A solução apresentada nesse texto usará a linguagem de programação Python. Ela é uma linguagem de alto nível, interpretada, multiparadigma e dinamicamente tipada e com o enfoque na legibilidade de código e facilidade de desenvolvimento. Criada em 1989, por Guido Von Rossum no Instituto de Pesquisa Nacional para Matemática e Ciências da Computação, nos Países Baixos. Ela é mantida pela organização sem fins lucrativos, Python *Software Foundation* (https://www.python.org/psf/), e está, atualmente, na sua versão 3.7.3, lançada em 25 de março de 2019.

A instalação do Python dependerá do sistema operacional utilizado. No caso do Linux, maioria das distribuições já vem com o Python instalado e configurado para ser executado diretamente pelo terminal. No caso do Windows, é necessário baixar o instalador e prosseguir com a instalação. Durante a instalação, marque a opção que permite ao instalador adicionar o Python na variável de ambiente `PATH`, como na Figura 4.1.

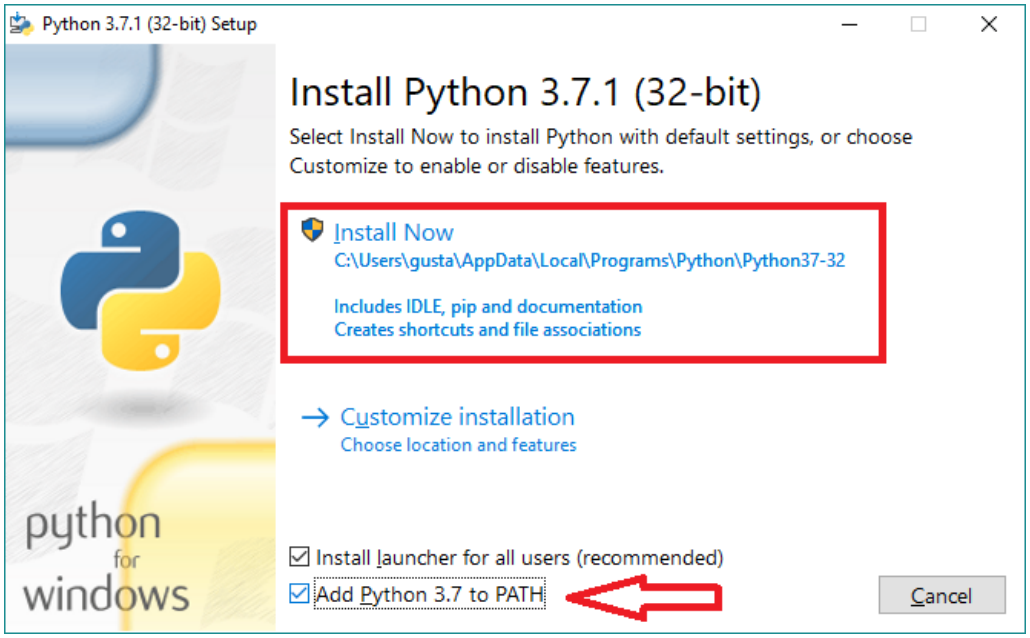

Figura 4.1. Instalando o Python no Windows. Fonte: http://bit.ly/2P0wqNN

Caso não tenha marcado a seleção como na Figura 4.1, será necessário configurar a variável de ambiente `PATH`. Para isso, será necessário identificar a localização do Python em seu computador e adicioná-la para variável de ambiente. Por exemplo, para um dado computador foi executado o seguinte comando:

```
$ set PATH=C:\Users\sergi\AppData\Local\Programs\Python\Python37-32\
    Scripts;C:\Users\sergi\AppData\Local\Programs\Python\Python37-32;%
    PATH%
```

Se está usando o Windows, uma alternativa interessante ao `CMD` é o terminal `git bash` que é instalado juntamente com o `git`. Caso esteja usando esse terminal, o comando a ser executado seria:

```
PATH=/C/Users/sergi/AppData/Local/Programs/Python/Python37-32:/C/Users/
    sergi/AppData/Local/Programs/Python/Python37-32/scripts:$PATH
```

Além do Python, será necessário ter uma conta na plataforma Heroku [3] que será usado para a implantação. Ela é uma plataforma como serviço ou PaaS, do inglês *Platform as a Service*, ou seja, ela provê serviços na nuvem que abstraem a infraestrutura de hardware e software que irá hospedar a aplicação. Heroku permite fácil escalabilidade para as aplicações a partir de *dynos*[4], que são máquinas virtuais que executam um sistema operacional com uma determinada quantidade de memória e processador. Nela as aplicações são executadas em contêineres de modo isolado. Ter uma conta no Heroku é fundamental para a implantação que será o último passo, como detalhado na Seção 4.3.

A plataforma Heroku se integra diretamente com repositórios Github. Então, nesse texto iremos usar tanto o Git quanto o Github para o gerenciamento e a implantação do serviço. O Git é um *software* livre, desenvolvido por Linus Torvalds, para versionamento que foi utilizado durante o desenvolvimento do kernel do linux [Torvalds 2005]. Basicamente o Git monitora todos os arquivos, diretório e subdiretórios dentro do projeto, provendo histórico completo e acompanhamento de revisões sem necessidade de uma rede encarregada pra isso ou uma máquina centralizada [Chacon 2014]. Por outro lado, o Github é uma ferramenta online para armazenamento de projetos gerenciados pelo Git e oferece a opção de hospedagem de projetos *open source* ou privados.

4.3. Mapeando os dados públicos para dados abertos em cinco passos

Nesta seção serão apresentados os cinco passos para transformar os dados públicos de um dado site, para o formato de dados abertos. O objetivo dessa seção é ser bem prática e permitir a fácil replicação por alguém com conhecimentos básicos em programação. Para a execução desses passos são usadas diversas tecnologias. A Figura 4.2 ilustra as principais tecnologias que serão destacadas brevemente em cada passo em que ela for utilizada, sendo que algumas já foram destacadas na Seção 4.2.

4.3.1. Identificação e modelagem dos dados

Como discutido anteriormente, existem diversos sites com dados relevantes, porém, inacessíveis diretamente por algoritmos e aplicativos. Eles apresentam os dados através de páginas web, porém não disponibilizam o acesso direto aos dados em formatos abertos. Então, a identificação e a modelagem dos dados requer a exploração do site e os dados de interesse.

Neste trabalho será usado o site da transparência do governo do Estado do Mara-

[3] https://www.heroku.com/about
[4] https://devcenter.heroku.com/articles/dynos

Figura 4.2. Principais tecnologias utilizadas

nhão (http://www.transparencia.ma.gov.br/). Ele foi lançado em 2015 e apresenta diversas melhorias em relação ao anterior, mas ainda não disponibiliza dados abertos. O acesso a todos os dados requer a interação entre um usuário e um navegador web. Por exemplo, ao acessar o seguinte endereço http://www.transparencia.ma.gov.br/app/despesas/por-funcao/2019#lista o usuário irá visualizar uma página HTML com uma tabela similar a da Figura 4.3, que descreve as despesas de cada função administrativa em 2019.

CÓDIGO	FUNÇÃO	EMPENHADO	LIQUIDADO	PAGO
12	EDUCACAO	1.684.622.426,16	1.390.292.803,31	625.194.524,41
10	SAUDE	1.381.755.880,82	1.153.294.533,10	1.042.897.165,13
09	PREVIDENCIA SOCIAL	1.304.287.119,54	1.275.240.828,87	179.238.233,65
06	SEGURANCA PUBLICA	1.075.738.298,56	1.037.012.947,76	776.244.140,29
28	ENCARGOS ESPECIAIS	902.049.666,77	894.992.848,30	863.561.971,07
02	JUDICIARIA	675.138.826,77	603.905.156,34	556.102.369,34
04	ADMINISTRACAO	529.798.789,14	451.046.595,86	351.247.216,01
03	ESSENCIAL A JUSTICA	417.413.106,68	379.601.831,41	373.304.788,51
01	LEGISLATIVA	328.702.859,43	312.632.771,91	273.589.849,54
14	DIREITOS DA CIDADANIA	272.659.104,45	214.876.489,62	187.946.915,15

Figura 4.3. Despesas por função administrativa em 2019

Observe pela Figura 4.3 que cada linha representa uma função administrativa, com o seu código, nome e valores empenhado, liquidado e pago. No próximo passo, cada um destes dados serão extraídos para depois serem enviados em um formato de dados abertos.

4.3.2. Extração dos dados

No segundo passo, os dados selecionados serão extraídos usando uma biblioteca Python denominada *Beautiful Soup*[5]. Essa biblioteca permite automatizar a obtenção de dados de um *website*, através de um conjunto técnicas denominadas de *Web scraping*. A função de um *web scraper* é verificar certos tipos de informações, extraí-las e agregá-las em novos formatos [Mitcheal 2015].

Então, antes de mais nada será necessário já ter instalado e configurado o Python, como discutido na Seção 4.2. Com o Python configurado, já é possível acessá-lo diretamente pelo terminal, incluso o instalador de pacotes `pip`. Com ele instalado, pode-se instalar o `pipenv` através do seguinte comando:

```
$ pip install pipenv
```

O Pipenv atua como criador e gerenciador de dependências em ambientes de desenvolvimento e de produção. Em resumo, o pipenv integra em um mesmo utilitários as tarefas que eram usualmente desempenhados através do *pip* e do *virtualenv* (`https://virtualenv.pypa.io/`. Ou seja, ele gerencia tanto as dependências quanto se encarrega de criar e carregar um ambiente isolado Python para a aplicação [VanTol 2018].

Com o pipenv instalado, crie a pasta para o projeto chamada `transparencia-ma` e em um terminal Linux ou Windows (CMD ou Git Bash), execute os seguintes comandos:

```
$ pipenv --three
$ pipenv install requests beautifulsoup4 lxml
```

O primeiro comando irá iniciar um ambiente isolado para a versão 3 do Python, e o segundo irá instalar as três bibliotecas básicas usadas durante esse passo. A biblioteca `requests` é usada para fazer requisições HTTPs, `beautifulsoup4` para a extração de dados e a `lxml` é usada pelo `Beautiful Soup` para analisar as estruturas em HTML ou XML [Richardson 2019].

> **Fica a dica**
>
> Use um ambiente de desenvolvimento integrado (IDE) que tenha acesso a terminais de comandos dentro do própria aplicação, como o Visual Studio Code (`https://code.visualstudio.com/`).

Usando uma IDE, ou diretamente pelo terminal, crie um arquivo denominado (`scrapper.py`) com o Código 4.1.

```
1 import requests
2 from bs4 import BeautifulSoup as BS
3
4 url = "http://www.transparencia.ma.gov.br/app/despesas/por-funcao/2019#
    lista"
5 page_response = requests.get(url)
```

[5] `https://www.crummy.com/software/BeautifulSoup/bs4/doc/` - Mais informações sobre o Beautiful Soup

```
6 page = BS(page_response.text, 'lxml')
7 table = page.find ('table')
8 print (table)
```

Código 4.1. Arquivo scrapper.py: Um simples teste para verificar a correta instalação

O Código 4.1 vai ser usado apenas para testar a instalão do ambiente e bibliotecas. Na linha 5 é feito a requisição da página HTML usando a bibioteca `requests`. Na linha 6, usa-se uma função da biblioteca `Beautiful Soup` para transformar o conteúdo requisitado em um objeto que irá permitir a busca de elementos específicos. Como ocorre na linha 7, que pegou a primeira tabela (`table`) do resultado da requisição. Para testar esse código, em um terminal execute os seguintes comandos:

```
$ pipenv shell
$ python scrapper.py
```

O primeiro comando ativa o ambiente, enquanto o segundo executa o *script*. A execução dos comandos acima, deverá imprimir na tela a tabela HTML que foi encontrada.

CheckPoint

Se chegou até aqui, significa que já está com o Python, Pipenv e as bibliotecas funcionando corretamente.

Antes de extrair os dados de interesse é necessário analisar como eles estão apresentados através de *tags* HTML. Usualmente, essa tarefa será executada explorando os recursos presentes nos navegadores web atuais que permitem visualizar o código fonte das páginas. Neste texto, pegou-se uma linha qualquer da tabela impressa pela execução do Código 4.1, tendo como saída o Código 4.2.

```
100 <tr>
101 <td>19</td>
102 <td class="secondLeft">
103 <a href="http://www.transparencia.ma.gov.br/app/despesas/por-funcao
    /2019/funcao/19?">
104             CIENCIA E TECNOLOGIA
105             </a>
106 </td>
107 <td>22.570.913,72</td>
108 <td>17.746.719,47</td>
109 <td>13.433.197,71</td>
110 </tr>
```

Código 4.2. Trecho de código HTML da tabela de a ser extraída

No Código 4.2, a primeira coluna é o código da função administrativa. A segunda coluna possui o nome da função administrativa e uma URL para mais detalhes sobre as despesas dessa função. As demais colunas apresentam respectivamente os valores empenhado, liquidado e pago. Então, para extrair os dados, apague o código escrito anteriormente e substitua-o pelo Código 4.3.

```
1 import requests
2 from bs4 import BeautifulSoup as BS
```

```
3
4  def despesas_total ():
5      url = "http://www.transparencia.ma.gov.br/app/despesas/por-funcao
       /2019#lista"
6      response = requests.get(url)
7      page = BS(response.text, 'lxml')
8      table = page.find ('table')
9      rows = table.find_all('tr')
10     despesas = []
11     for row in rows[1:3]: # testando apenas com 3 linhas
12         cols =row.find_all("td")
13         despesa = {}
14         despesa["nome"] = cols[1].find("a").get_text().strip()
15         despesa["url_detalhe"] = cols[1].find("a").get('href')
16         despesa["empenhado"] = cols[2].get_text().strip()
17         despesa["liquidado"] = cols[3].get_text().strip()
18         despesa["pago"] = cols[4].get_text().strip()
19         despesas.append(despesa)
20
21     return despesas
22
23 # remover o codigo abaixo
24 print (despesas_total())
```

Código 4.3. Arquivo scrapper.py: Extraindo os dados de despesas

O Código 4.3 pode ser facilmente compreendido por alguém que tenha familiaridade com a linguagem Python. Na linha 8 está sendo recuperada a tabela através do método `find` que irá retornar a primeira tabela existente. Nesse caso específico, é exatamente a tabela que possui os dados a serem extraídos. Em alguns casos, pode ser necessário passar algumas informações adicionais para que se possa recuperar a tabela de interesse.

Com a tabela recuperada o comando `find_all` irá retornar todas as linhas dessa tabela, como isso incluir a linha de cabeçalho da tabela na linha 11 ela será ignorada. Então, para cada linha é recuperada as colunas, e depois é construído um objeto `despesa` com as chaves `nome`, `url_detalhe`, `empenhado`, `liquidado` e `pago`. Por fim, esse código já pode ser testado com o comando a seguir.

```
 $ python scrapper.py
[{'nome': 'ADMINISTRACAO', 'url_detalhe': 'http://www.transparencia.ma.
   gov.br/app/despesas/por-funcao/2019/funcao/04?', 'empenhado':
   '530.070.090,32', 'liquidado': '452.420.550,87', 'pago':
   '351.633.728,08'}, {'nome': 'AGRICULTURA', 'url_detalhe': 'http://
   www.transparencia.ma.gov.br/app/despesas/por-funcao/2019/funcao
   /20?', 'empenhado': '69.816.420,71', 'liquidado': '58.739.278,15',
   'pago': '45.119.980,63'}]
```

CheckPoint

Neste ponto, já está finalizado um código para extração dos dados. A execução desse código deverá produzir uma saída similar a apresentada anteriormente.

Com o código executado corretamente, no próximo passo será desenvolvida a API

de dados abertos.

4.3.3. Desenvolvimento

No terceiro passo é desenvolvido uma API REST que irá disponibilizar os dados extraídos em um formato acessível para algoritmos computacionais. O REST (*Representational State Transfer*), ou em português Estado Representacional de Transferência, é uma abstração da arquitetura web proposta em [Fielding 2000]. Ela consiste em um conjunto de princípios/regras/*constraints* baseadas no protocolo HTTP para a criação de APIs com interfaces bem definidas. Atualmente as diversas linguagens de programação possuem *frameworks* que auxiliam na construção desse tipo de APIs.

Para esse texto, foi usado o *micro framework* denominado Flask[6], com a extensão flask-restplus[7]. Uma de suas maiores vantagens é a possibilidade de desenvolver APIs simples que poderão ser extendidas através de bibliotecas a medida que surgem novas necessidades A instalação dessa biblioteca é similar as demais:

```
$ pipenv install flask-restplus
```

Após a instalação já é possível escrever uma API bem simples que para uma única rota retorne os dados extraídos na seção anterior, como apresentada no Código 4.4.

```
1  from flask import Flask
2  from flask_restplus import Resource, Api, fields
3  from scrapper import despesas_total
4
5  app = Flask(__name__)
6  api = Api(app)
7
8  @api.route('/despesas')
9  class Despesas(Resource):
10     def get(self):
11         return despesas_total()
12
13 if __name__ == '__main__':
14     app.run(debug=True)
```

Código 4.4. Arquivo app.py: Primeira versão da API

As rotas são implementadas com o decorator `@api.route`, que recebe como parâmetro a rota, que neste caso é `despesas`. Para essa rota só está implementado o método GET, que irá retornar todos os dados extraídos pela função `despesas_total`. A função retorna os dados como uma lista de dicionários, e será convertida e retornada para o cliente no formato JSON (*JavaScript Object Notation*). Esse um formato de texto de tipo chave-valor muito utilizado para a serialização de dados abertos, por ser autodescritivo e de fácil interpretação e geração, tanto por agentes humanos, quanto por agentes computacionais (`https://www.json.org/`).

Para testar a primeira versão do servidor, basta executar o Código 4.4:

```
$ python app.py
```

[6]Site oficial http://flask.pocoo.org/

[7]Documentação: https://flask-restplus.readthedocs.io/en/stable/

```
* Serving Flask app "app" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production
    deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Restarting with stat
* Debugger is active!
* Debugger PIN: 230-864-203
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

A saída acima mostra que o servidor foi iniciado e esta aguardando requisições através do endereço local (127.0.0.1) e na porta 500. Então, já é possível acessar os dados através de um cliente qualquer, por exemplo, entrar com o seguinte endereço `http://localhost:5000/despesas` em um navegador web.

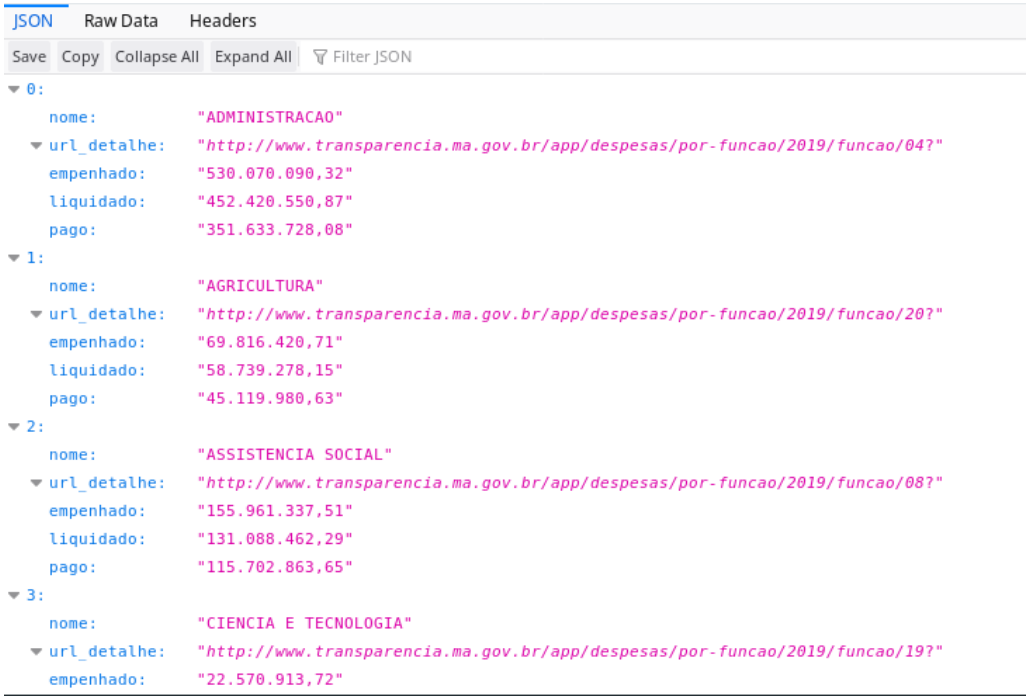

Figura 4.4. Visualizando os dados de despesas, testando o Código 4.4

Tendo uma saída similar a da Figura 4.4 significa que a API está funcionando. Porém, é necessário fazer algumas melhorias para torná-la mais útil, por exemplo, permitindo especificar quais os dados que serão retornados. Ao verificar o Código 4.3, é possível identificar pela URL que os dados que estão sendo retornados são referentes apenas ao ano de 2019. Porém, é possível acessar dados de diferentes anos apenas mudando o ano na URL. Lembrando, que este site só disponibiliza os dados a partir de 2015.

Com essa característica pode-se adaptar a função `despesas_total` para retornar os dados de um determinado ano, incluindo-o como parâmetro e usando-o para formar a URL, como na linha 6 do Código 4.5.

```
4 def despesas_total (ano):
5     url_base = "http://www.transparencia.ma.gov.br/app"
```

```
    url = url_base + "/despesas/por-funcao/"+ano
    response = requests.get(url)
    ...
```

Código 4.5. Arquivo `scrapper.py`: Adicionando o ano como parâmetro da função despesas_total

Tendo alterado a função `despesas_total`, deve-se alterar a rota para incluir o ano, e então passá-lo para a função despesas total, como demonstrado no Código 4.6.

```
@api.route('/despesas/<string:ano>')
class Despesas(Resource):
    def get(self, ano):
        return despesas_total(ano)
```

Código 4.6. Arquivo `app.py`: Adicionando o ano para a rota da API

Com essas alterações é possível acessar os dados através de uma rota que inclui o ano de referência para os dados. Por exemplo, os dados do ano de 2016 poderão ser acessados através da seguinte URL: `http://localhost:5000/despesas/2016`.

O site da transparência do Governo do Maranhão permite visualizar os detalhes das despesas de uma dada função ou órgão administrativo. Por exemplo, o código da função administrativa **educação** é 12. Então, a `http://www.transparencia.ma.gov.br/app/despesas/por-funcao/2018/funcao/12#lista` detalha como a despesa com a educação foi distribuída para cada orgão.

Acessando este endereço, percebe-se que a estrutura dos dados é muito similar, permitindo reutilizar o código escrito anteriormente. A principal diferença é na formação da URL que requer o código da função administrativa. Então, pode-se adaptar o código do arquivo `scrapper.py` incluindo uma função que irá extrair o total das despesas e outra detalhada por função administrativa, Código 4.7.

```
import requests
from bs4 import BeautifulSoup as BS
def despesas_total (ano):
    url_base = "http://www.transparencia.ma.gov.br/app"
    url = url_base + "/despesas/por-funcao/"+ano
    return extrai_despesas (url)

def despesas_por_funcao (cod, ano):
    url_base = "http://www.transparencia.ma.gov.br/app"
    url = url_base + "/despesas/por-funcao/"+ano+"/funcao/"+cod
    return extrai_despesas (url)
```

Código 4.7. Arquivo `scrapper.py`: Incluindo a extração por função administrativa.

Ambas funções do Código 4.7 apenas montam a URL e repassam para a função `extrai_despesas`, apresentada pelo Código 4.8.

```
def extrai_despesas (url):
    response = requests.get(url)
    page = BS(response.text, 'lxml')
    table = page.find ('table')
    rows = table.find_all('tr')
    despesas = []
```

```
18      for row in rows[1:]:
19          cols =row.find_all("td")
20          despesa = {}
21          despesa["codigo"]  = cols[0].get_text().strip()
22          despesa["nome"] = cols[1].find("a").get_text().strip()
23          despesa["url_detalhe"] = cols[1].find("a").get('href')
24          despesa["empenhado"] = cols[2].get_text().strip()
25          despesa["liquidado"] = cols[3].get_text().strip()
26          despesa["pago"] = cols[4].get_text().strip()
27          despesas.append(despesa)
28      return despesas
```

Código 4.8. Arquivo `scrapper.py`: Detalhe da função extrai_despesas.

Para suportar essa nova rota, será necessário adaptar o arquivo `app.py` para incluí-la. O Código 4.9 demonstra a inclusão dessa nova rota, que inclui o código da função administrativa e o ano de referência.

```
15 @api.route('/<string:cod_funcao>/<string:ano>')
16 class DespesasPorFuncao(Resource):
17     def get(self, cod_funcao, ano):
18         return despesas.despesas_por_funcao(cod_funcao, ano)
```

Código 4.9. Arquivo `app.py`: Inclusão de nova rota para retornar as despesas detalhadas por função administrativa

Com essas alterações, pode-se testar o servidor através de um navegador web ou pelo terminal:

```
$ curl -X GET "http://localhost:5000/despesas/12/2018" -H  "accept:
    application/json"
[  {       "codigo": "170203",    "nome": "FUNDACAO NICE LOBAO",
...
```

CheckPoint

Nesse ponto, já está finalizado o código básico da API. A execução acima deverá retornar os dados em JSON de diferentes órgãos. O mesmo teste poderá ser feito diretamente no navegador, caso não tenha o aplicativo `curl`.

4.3.4. Documentação

Uma parte importante em qualquer API é uma boa documentação. Então, no quarto passo será utilizado a biblioteca Swagger[8] para a construção automatizada de documentação. A biblioteca `flask-restplus` vem com o suporte para o Swagger e já cria uma documentação básica ao acessar o endereço raiz da API. Porém, através de uma coleção de *decorators* e parâmetros é possível adicionar novas informações ao código, gerando uma documentação mais detalhada como no Código 4.10.

```
1 from flask import Flask
2 from flask_restplus import Resource, Api, fields
3 from scrapper import despesas_total, despesas_por_funcao
4
```

[8] Site oficial https://swagger.io/

```
5  app = Flask(__name__)
6  api = Api(app = app,
7        version = "1.0",
8        title = "Transparência Maranhão",
9            description = "Uma API não oficial com os dados sobre as
       receitas e despesas do Governo do Maranhão")
10
11 ns = api.namespace('despesas', description='Dados de despesas')
12
13 @ns.route('/<string:ano>')
14 class Despesas(Resource):
15     def get(self, ano):
16         return despesas_total(ano)
17 @ns.route('/<string:cod_funcao>/<string:ano>')
18 class DespesasPorFuncao(Resource):
19     def get(self, cod_funcao, ano):
20         return despesas_por_funcao(cod_funcao, ano)
21
22 if __name__ == '__main__':
23     app.run(debug=True)
```

Código 4.10. Arquivo `app.py`: Adicionando novas informações para o Swagger

As linhas 7, 8 e 9 adicionaram a versão, o nome e a descrição como informações principais da API. Além disso, uma API pode ter diferentes rotas, por exemplo, poderia ter rotas especificas para despesas e outras para receitas. Essas rotas poderiam estar agrupadas por dois diferentes *namespaces*. Aqui foi então criado na linha 11, um *namespace* para as despesas, incluindo sua descrição, e as linhas 13 e 17 foram adaptadas para usá-lo. O resultado da documentação pode ser observado na Figura 4.5.

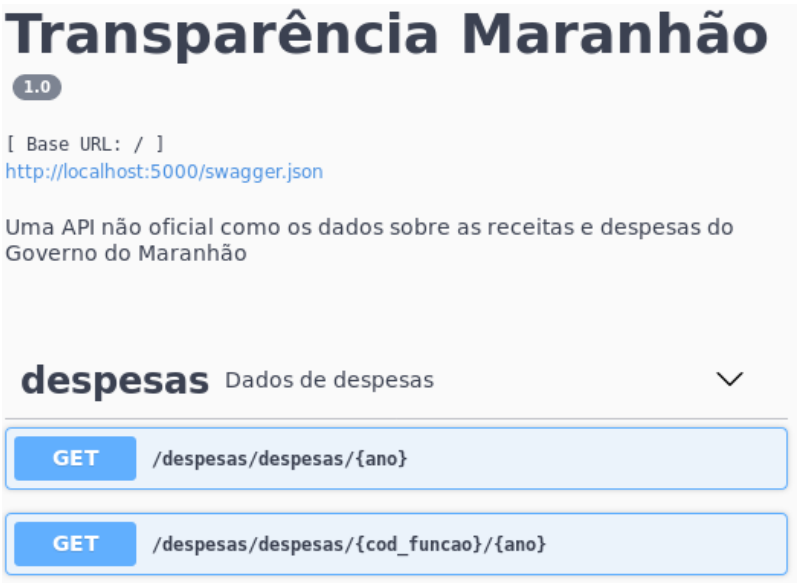

Figura 4.5. Visualização da documentação em Swagger

Além das informações para a API e *namespace*, é possível adicionar informações diretamente aos métodos e parâmetros, como apresentado no Código 4.11.

```
...
@ns.route('/<string:ano>')
class Despesas(Resource):
    @api.doc(responses={ 200: 'OK', 400: 'Despesas não encontradas' },
     params={ 'ano': 'Ano de referência para as despesas' })
    def get(self, ano):
        return despesas.despesas_total(ano)

@ns.route('/<string:cod_funcao>/<string:ano>')
class DespesasPorFuncao(Resource):
    @api.doc(responses={ 200: 'OK', 400: 'Despesas não encontradas' },
      params={ 'ano': 'Ano de referência para as despesas',
    'cod_funcao' : 'Código da função (educação, saúde ...) de referê
   ncia para as despesas'})
    def get(self, cod_funcao, ano):
        return despesas.despesas_por_funcao(cod_funcao, ano)
...
```

Código 4.11. Arquivo `app.py`: Adicionando novas informações para os métodos HTTP.

No código 4.11 foi utilizado o *decorator* `@api.doc` para passar algumas informações para o método GET, tendo como resultado a Figura 4.6.

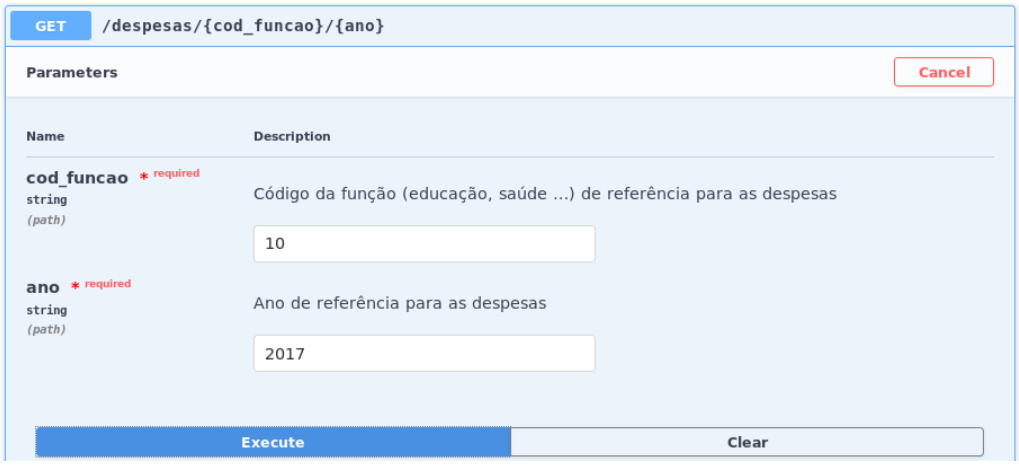

Figura 4.6. Documentação para o métogo GET no Swagger

Por fim, pode-se criar os metadados dos dados providos pela API, que incluem os tipos e as descrições dos dados. Os tipos de dados para os valores liquidados, pagos e empenhados são números, no entanto, os dados extraídos estão no formato de texto e usando a representação brasileira. Então, a conversão para número deverá considerar essa representação. Existe uma biblioteca denominada *babel* que possui já implementada essa funcionalidade e pode ser instalada com o seguinte comando:

```
$ pipenv install babel
```

Com a biblioteca *babel* instalada, será necessário algumas atualização no arquivo `scrapper.py`. Primeiro será necessário importar a função `parse_decimal`.

```
from babel.numbers import parse_decimal
```

E depois, aplicá-la para os valores empenhado, liquidado e pago:

```
despesa["empenhado"] = parse_decimal(cols[2].get_text().strip(), locale
    ='pt_BR')
```

Após a alteração do arquivo `scraper.py`, será necessário descrever os metadados através da função `api.model` como no Código 4.12. Além do tipo e descrição, foi incluído um valor de exemplo que será utilizado também para a documentação.

```
model = api.model('Dados sobre uma função ou orgão', {
    'codigo': fields.String(description='Código da função ou orgão',
    example="04"),
    'nome': fields.String(description='Nome da função ou orgão',
    example="ADMINISTRACAO"),
    'url_detalhe': fields.String(description='Endereço para mais
    detalhes', example="http://www.transparencia.ma.gov.br/app/despesas
    /por-funcao/2016/funcao/04?"),
    'empenhado': fields.Float(description='Valor empenhado', example
    =821854500.93),
    'liquidado': fields.Float(description='Valor liquidado', example
    =794738131.95),
    'pago': fields.Float(description='Valor pago', example=775701742.7)
    ,
})
```

Código 4.12. Arquivo `app.py`: Adicionando o metadado a API

Para associar o metadado aos dados retornados, será usado o *decorator* `@api.marshal_with` em ambas rotas:

```
@ns.route('/<string:ano>')
class Despesas(Resource):
    @api.marshal_with(model, mask='*')
    @api.doc ...
```

Após a atualização da API, pode-se visualizar os metadados como na Figura 4.7.

4.3.5. Implantação

O quinto e último passo tem como objetivo implantar a API desenvolvida. Aqui será usada a plataforma Heroku[9]. Então, antes de prosseguir será necessário criar uma conta gratuita nesse portal. Para a implantação será usado também o gerenciador de configuração Git e o repositório Github (`http://github.com/`.

Antes de mais nada, vamos instalar o *gunicorn* (`https://gunicorn.org/`), um servidor WSGI (Web Server Gateway Interface) necessário para executar os *scripts* Python do lado do servidor. De modo similar as instalações anteriores, basta executar o seguinte comando:

```
$ pipenv install gunicorn
```

Depois de instalado, será necessário criar um arquivo denominado Procfile[10] que é utilizado pelo Heroku para a inicialização do serviço. Nesse caos ele indicará o WSGI

[9] Site oficial: `https://www.heroku.com/`

[10] Mais informações sobre o Procfile `https://devcenter.heroku.com/articles/procfile`

Figura 4.7. Visualização dos metadados

e o nome do aplicativo. Neste caso o nome do aplicativo é `app`, e está localizado no módulo (ou arquivo) `app.py`.

```
web: gunicorn app:app
```

Como será usado o repositório *Github*, garanta que já tenha uma conta no *github* e o aplicativo *git* instalado e configurado no computador. Com isso, será necessário logar na sua conta, criar um repositório denominado `transparencia-ma` e executar os seguintes comandos porém usando o repositório que foi criado.

```
git init
git add *
git commit -m "first commit"
git remote add origin https://github.com/profsergiocosta/transparencia-
    ma.git
git push -u origin master
```

Fica a dica

Nesse exemplo está sendo um repositótio HTTP, então será necessário entrar com o nome do usuário e senha para enviar os dados para o repositório, ou seja, executar o comando *push*. Uma dica é trabalhar com repositório SSH ao invés de HTTP, caso esteja usando esses procedimento em um computador pessoal.

A Figura 4.8 mostra os três passos para implantar uma aplicação no Heroku. O passo 1 cria uma aplicação Heroku com o nome `transparencia-ma`, o 2 conecta-se a aplicação ao repositório *Github* e o 3 executa a implantação (*deploy*) manual. Na Figura

4.8 (4) apenas mostra as mensagens que são impressas durante a implantação. Além do *deploy* manual, é possível marcar implantação (*deploy*) automática, que irá ocorrer a cada novo *commit* ao repositório github.

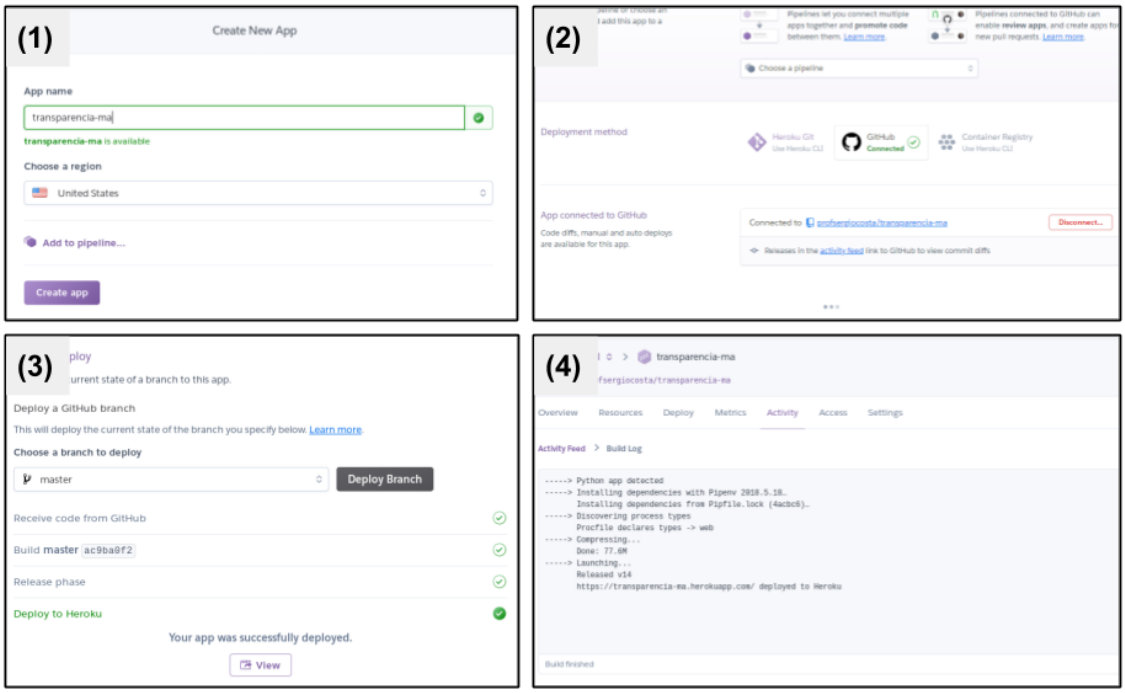

Figura 4.8. Implantação no Heroku em 3 passos

Depois de implantado, é possível testar diretamente em um navegador ou usando o comando `curl`.

```
$ curl -X GET "https://transparencia-ma.herokuapp.com/despesas/2018" -H
     "accept: application/json"
[{"codigo": "04", "nome": "ADMINISTRACAO", "url_detalhe": "http://www.
   transparencia.ma.gov.br/app/despesas/por-funcao/2018/funcao/04?"
...
```

Na próxima Seção, serão apresentados outros exemplos de como usar a API desenvolvida.

4.4. Experimentando e evoluindo

A disponibilização de dados abertos cria muitas oportunidades de soluções que podem ser desenvolvidas pela comunidade. Nesta Seção, serão apresentados alguns exemplos simples de como uma API pode ser utilizada por algoritmos externos.

4.4.1. Um teste simples de análise de dados

O portal da transparência do Maranhão já disponibiliza diversos dados tabelados. Porém, um usuário pode precisar de um dado que não esteja disponível diretamente. Por exemplo, o usuário poderia querer um gráfico que mostrasse a evolução temporal dos gastos por uma dada função administrativa. Sem uma API, seria necessário entrar em diversas

páginas e manualmente anotar os gastos, ou salva-los e manipulá-los através de outro software. Contudo, com a API desenvolvida aqui é possível gerar este gráfico escrevendo um pouco de código em linguagens como Python, R ou JavaScript.

Para exemplificar, foi escrito um código Python diretamente no Google Colab [11]. O código completo da Figura 4.9 pode ser acessado em http://bit.ly/304z5XF.

```python
import json, requests
import matplotlib.pyplot as plt

url = "https://transparencia-ma.herokuapp.com/despesas/"
valores =[]
anos = []
codigo_funcao = "19" # 19 é o código para ciência e tecnologia
for ano in range (2015, 2020):
  r = requests.get(url+str(ano))
  anos.append (ano)
  dados = r.json()
  for dado in dados:
    if dado['codigo'] == codigo_funcao:
      valores.append(dado['empenhado'])
      break
plt.bar (anos,valores)
plt.show()
```

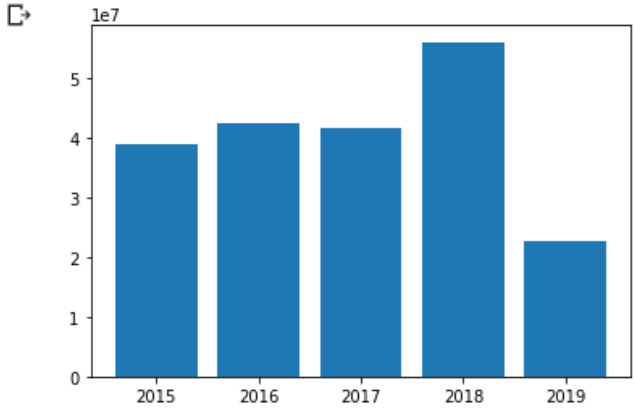

Figura 4.9. Código e gráfico gerado no Google Colab

4.4.2. Um teste simples de aplicação

Além de ser usado para análise, os dados abertos podem ser acessados por aplicativos web e móveis. Nesse caso, uma possibilidade seria desenvolver aplicativos híbridos usando a linguagem JavaScript, através de *frameworks* como o Angular, React e Electron.

Como o objetivo é fazer um teste simples, foi escrito um código usando a biblioteca *JQuery* que executa uma requisição e imprime no console o resultado, Código 4.13. O código completo pode ser acessado em https://repl.it/@SergioSouza1/testeapi.

```
22 <script>
23 $(document).ready(() =>
24     $("button").click( () =>  $.ajax({
25         url: "https://transparencia-ma.herokuapp.com/despesas/10/2018",
```

[11] Google Colab pode ser acessado em: https://colab.research.google.com/

```
        success: (result) => console.log (result) })
))
</script>
```

Código 4.13. Testando a API através de requisições assíncronas

Antes de executar o teste, modifique a URL para referenciar o serviço que foi criado no Heroku. Porém, esse simples teste irá falhar e apresentar seguinte mensagem no navegador:

```
Cross-Origin Request Blocked: The Same Origin Policy disallows reading
    the remote resource at https://transparencia-ma.herokuapp.com/
    despesas/10/2018. (Reason: CORS header  Access -Control-Allow-
    Origin missing).
```

Isso ocorre porque, por padrão, o JavaScript não permite que aplicações de domínios distintos compartilhem recursos. Porém é possível habilitar o que é conhecido como CORS (que em português seria Compartilhamento de Recursos de Origem Cruzada). Então, para tudo funcionar corretamente, será necessário retornar ao ambiente de desenvolvimento da API.

Primeiro passo será instalar a extensão para dar esse suporte, que é feito pela biblioteca `flask_cors`:

```
pipenv install flask_cors
```

Depois no arquivo `app.py`, sera adicionado um novo *import*:

```
from flask_cors import CORS
```

E então, será executado a função CORS ao aplicativo:

```
CORS(app)
```

O código completo pode ser acessado em: http://bit.ly/2PcfHXZ.

Atualizado esse código, deve-se enviar as modificações para o repositório *Github* e, por fim, executar a implantação novamente no Heroku. Com o serviço implantado e atualizado, basta executar o código novamente para ser impresso os dados em JSON no console. Lembrando, que esse é um exemplo extremamente simples, porém a sua execução mostra que a API está pronta para ser usada por aplicações em JavaScript.

4.4.3. Evolução

O site do governo do Maranhão possui diversos outros dados que poderão ser extraídos. Por exemplo, ao acessar um dado órgão, pela seguinte http://www.transparencia.ma.gov.br/app/despesas/por-funcao/2019/funcao/12/orgao/240201 é possível ter acesso aos nomes, CPF ou CNPJ dos credores, e também os valores empenhado, liquidado e pago, como na Figura 4.10.

CREDOR	EMPENHADO	LIQUIDADO	PAGO
A. L. T TRINDADE CPF/CNPJ: 30.865.611/0001-63	134.999,70	0,00	0,00
A.E. MENDES CPF/CNPJ: 41.472.655/0001-40	13.650,00	0,00	0,00
ABILIO NEIVA MOREIRA CPF/CNPJ: 00.002.513/2433-82	459,00	459,00	459,00

Figura 4.10. Credores de um dado orgão público

Observa-se que a tabela é muito similar a que foi extraída na Seção 4.3, com a diferença que cada credor possui um CPF ou CNPJ em vez de um código. Então o Código 4.14 será também muito similar ao escrito anteriormente na Seção 4.3

```
30
31 def despesas_por_orgao (orgao, funcao, ano):
32     url = "http://www.transparencia.ma.gov.br/app/despesas/por-funcao/"
    +ano+"/funcao/"+funcao+"/orgao/"+orgao+"?#lista"
33     response = requests.get(url)
34     page = BS(response.text, 'lxml')
35     table = page.find ('table')
36     rows = table.find_all('tr')
37     despesas = []
38     for row in rows[1:]:
39         cols =row.find_all("td")
40         despesa = {}
41         despesa["nome"] = cols[0].find("a").get_text().strip()
42         despesa["url_detalhe"] = cols[0].find("a").get('href')
43         despesa["cpf/cnpj"] = cols[0].find("small").get_text().strip().
    replace ("CPF/CNPJ: ","")
44         despesa["empenhado"] =  parse_decimal(cols[1].get_text().strip
    (), locale='pt_BR')
45         despesa["liquidado"] =  parse_decimal(cols[2].get_text().strip
    (), locale='pt_BR')
46         despesa["pago"] = parse_decimal (cols[3].get_text().strip(),
    locale='pt_BR')
47         despesas.append(despesa)
48     return despesas
```

Código 4.14. Extraindo os dados por orgão público

Após criar a função do Código 4.14, é possível criar uma nova rota que inclui o código do órgão administrativo, por exemplo: `https://transparencia-ma.herokuapp.com/despesas/240206/12/2018`. Então, será necessário atualizar a rota, fazer um novo *commit*, enviar as atualizações e executar um novo *deploy*.

O processo de transformação de dados públicos em dados abertos requer muita exploração do site de dados públicos. Por exemplo, aqui seria possível extrair ainda diversas outras informações como:

- Ainda sobre despesas, é possível extrair informações detalhadas por credor.

- Além dos dados de despesas, pode-se extrair os dados de receitas de modo muito similar.
- Dados sobre servidores públicos, como salário e diárias.
- Transferência de valores, por exemplo, repasse a municípios.

Além desses dados é possível evoluir bastante essa API, permitindo extrair cada vez mais informações relevantes.

4.5. Conclusões

A web continua a crescer e evoluir desde o primeiro site desenvolvido e publicado por Tim Berners-Lee em 06 de agosto de 1991 [12]. Atualmente, com o seu grande volume de dados, a web demanda ferramentas automatizáveis para a extração e estruturação das informações. Foi neste contexto, que surgiu o paradigma de dados abertos que hoje é muito demandado pela sociedade.

A disponibilização de dados abertos por órgãos públicos pode fomentar o desenvolvimento de soluções bem como a participação das pessoas no controle e na compreensão da gestão dos recursos públicos. Nesse texto, foi apresentada uma solução parcial que permitiu disponibilizar dados que já estavam públicos como dados abertos, através do desenvolvimento de uma API. Essa solução é interessante, pois atualmente existem diversos portais com muitos dados públicos mas que não os disponibilizam em formato legível por computador. Muitos desses portais não têm previsão para o lançamento de uma API de dados abertos. Porém, a solução proposta aqui tem a desvantagem de ter um alto custo computacional, pois requer carregar uma página web e extrair as informações antes de enviar os dados. Uma possibilidade para contornar o problema de custo computacional, é salvar os dados extraídos das páginas em um banco de dados.

Acredita-se que os passos detalhados aqui, irão permitir a qualquer programador transformar qualquer site de dados públicos em uma API de dados abertos, sendo apenas necessário um pouco de experiência na linguagem Python. Uma possibilidade seria continuar a extrair mais informações do site da transparência do governo do Maranhão, ou de outro site que tenha interesse.

Por fim, as tecnologias utilizadas nesse texto não são exclusivas para a criação de portais de dados abertos. Na verdade, o estilo arquitetura REST é muito utilizado ainda por diversas soluções. Principalmente para a comunicação de serviços e aplicativos móveis. Essa abordagem permite que um mesmo serviço seja consumido por diferentes clientes, e as redes sociais são excelentes exemplos de sucesso dessa abordagem.

Referências

[Chacon 2014] Chacon, S. (2014). *Git Pro*. Apress, 2nd edition.

[Daga et al. 2015] Daga, E., d'Aquin, M., Adamou, A., and Brown, S. (2015). The open university linked data - data.open.ac.uk. *Semantic Web Journal*, 7(2):183–191.

[12] Esse site ainda pode ser acessado em http://info.cern.ch/hypertext/WWW/TheProject.html

RASCUNHO

[Dietrich et al. 2009] Dietrich, D., Gray, J., McNamara, T., Poikola, A., Pollock, P., Tait, J., and Zijlstra, T. (2009). Open data handbook.

[Fielding 2000] Fielding, R. T. (2000). Rest: architectural styles and the design of network-based software architectures. *Doctoral dissertation, University of California*.

[Keßler and Kauppinen 2012] Keßler, C. and Kauppinen, T. (2012). Linked open data university of münster–infrastructure and applications. In *Extended Semantic Web Conference*, pages 447–451. Springer.

[Lessig 2007] Lessig, L. (2007). 8 principles of open government data.

[Mitcheal 2015] Mitcheal, R. (2015). *Web Scraping with Python - Collecting data from the modern web*. O'Reilly, 1nd edition.

[Molloy 2011] Molloy, J. C. (2011). The open knowledge foundation: open data means better science. *PLoS Biol*, 9(12):e1001195.

[Richardson 2019] Richardson, L. (2019). Beautiful soup documentation.

[Torvalds 2005] Torvalds, L. (2005). Kernel scm saga..

[VanTol 2018] VanTol, A. (2018). Pipenv: A guide to the new python packaging tool.

[Vaz 2011] Vaz, J. C.; Ribeiro, M. M. M. R. E. (2011). Dados governamentais abertos e seus impactos sobre os conceitos. *Cadernos PPG-AU/UFBA*, pages 45–62.

[Zablith et al. 2011] Zablith, F., Fernandez, M., and Rowe, M. (2011). The ou linked open data: production and consumption. In *Extended Semantic Web Conference*, pages 35–49. Springer.