API*, PYTHON*

Проектирование RESTful API с помощью Python и Flask

ПЕРЕВОД

NCNecros 25 декабря 2014 в 18:43 99k

Оригинал: Miguel Grinberg

В последние годы REST (REpresentational State Transfer) стала стандартной архитектурой при дизайне веб-сервисов и веб-API.

В этой статье я покажу вам как просто создавать RESTful веб-сервисы используя Python и микрофреймворк Flask.

Что такое REST?

Характеристика системы REST определяется шестью правилами дизайна:

- **Клиент-Сервер**: Должно быть разделение между сервером, который предлагает сервис и клиентом, который использует ее.
- **Stateless**: Каждый запрос от клиента должен содержать всю информацию, необходимую серверу для выполнения запроса. Другими словами, сервер не обязан сохранять информацию о состоянии клиента.
- **Кэширование**: В каждом запросе клиента должно явно содержаться указание о возможности кэширования ответа и получения ответа из существующего кэша.

- **Уровневая система**: Клиент может взаимодействовать не напрямую с сервером, а с произвольным количеством промежуточных узлов. При этом клиент может не знать о существовании промежуточных узлов, за исключением случаев передачи конфиденциальной информации.
- **Унификация**: Унифицированный программный интерфейс сервера.
- **Код по запросу**: Сервера могут поставлять исполняемый код или скрипты для выполнения их на стороне клиентов.

Что такое RESTful веб-сервис?

Архитектура REST разработана чтобы соответствовать протоколу HTTP используемому в сети Интернет.
Центральное место в концепции RESTful веб-сервисов это понятие ресурсов. Ресурсы представлены URI. Клиенты отправляют запросы к этим URI используя методы представленные протоколом HTTP, и, возможно, изменяют состояние этих ресурсов.
Методы HTTP спроектированы для воздействия на ресурс стандартным способом:

Метод HTTP	Действие	Пример
GET	Получить информацию о ресурсе	example.com/api/orders (получить список заказов)
GET	Получить информацию о ресурсе	example.com/api/orders/123 (получить заказ #123)
POST	Создать новый ресурс	example.com/api/orders (создать новый заказ из данных переданных с запросом)
PUT	Обновить ресурс	example.com/api/orders/123 (обновить заказ #123 данными переданными с запросом)

DELETE	Удалить ресурс	example.com/api/orders/123 (удалить заказ #123)

Дизайн REST не дает рекомендаций каким конкретно должен быть формат данных передаваемых с запросами. Данные переданные в теле запроса могут быть JSON blob, или с помощью аргументов в URL.

Проектируем простой веб-сервис

При проектировании веб-сервиса или API нужно определить ресурсы, которые будут доступны и запросы, с помощью которых эти данные будут доступны, согласно правил REST.
Допустим мы хотим написать приложение To Do List и мы должны спроектировать веб-сервис для него. Первое что мы должны сделать, это придумать кореневой URL для доступа к этому сервису. Например мы могли бы придумать в качестве корневого URL что-то типа:

```
http://[hostname]/todo/api/v1.0/
```

Здесь я решил включить в URL имя приложения и версию API. Добавление имени приложения в URL это хороший способ разделить между собой сервисы запущенные на одном сервере. Добавление версии API в URL может помочь, если вы захотите сделать обновление в будущем и внедрить в новой версии несовместимые функции и не хотите ломать работающие приложения которые работают на старом API.
Следующим шагом мы должны выбрать ресурсы, которые будут

доступны через наш сервис. У нас очень простое приложение, у нас есть только задачи, поэтому нашими ресурсами могут быть только задачи из нашего ToDo листа.
Для доступа к ресурсам будем использовать следующие методы HTTP:

Метод HTTP	URI	Действие
GET	http://[hostname]/todo/api/v1.0/tasks	Получить список задач
GET	http://[hostname]/todo/api/v1.0/tasks/[task_id]	Получить задачу
POST	http://[hostname]/todo/api/v1.0/tasks	Создать новую задачу
PUT	http://[hostname]/todo/api/v1.0/tasks/[task_id]	Обновить существующую задачу
DELETE	http://[hostname]/todo/api/v1.0/tasks/[task_id]	Удалить задачу

Наша задача будет иметь следующие поля:

- **id**: уникальный идентификатор задачи. Тип Numeric.
- **title**: Краткое описание задачи. Тип String.
- **description**: подробное описание задачи. Тип Text.
- **done**: отметка о выполнении. Тип Boolean.

На этом мы заканчиваем часть посвященную дизайну нашего сервиса. Осталось только реализовать это!

Краткое введение в микрофреймворк Flask

Если вы читали серию Мега-Учебник Flask, вы знаете что Flask это простой и достаточно мощный веб-фреймворк на Python.
Прежде чем мы углубимся в специфику веб-сервисов, давайте рассмотрим как обычно реализованы приложения Flask.
Я предполагаю, что вы знакомы с основами работы с Python на

вашей платформе. В примерах я буду использовать Unix-подобную операционную систему. Короче говоря, это означает, что они будут работать на Linux, MacOS X и даже на Windows, если вы будете использовать Cygwin. Команды будут несколько отличаться, если вы будете использовать нативную версию Python для Windows.

Для начала установим Flask в виртуальном окружении. Если в вашей системе не установлен `virtualenv`, вы можете загрузить его из https://pypi.python.org/pypi/virtualenv.

```
$ mkdir todo-api
$ cd todo-api
$ virtualenv flask
New python executable in flask/bin/python
Installing setuptools.............................done.
Installing pip...................done.
$ flask/bin/pip install flask
```

Теперь, когда Flask установлен давайте создадим простое веб приложение, для этого поместим следующий код в `app.py`:

```
#!flask/bin/python
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return "Hello, World!"

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

Чтобы запустить приложение, мы должны запустить `app.py`:

```
$ chmod a+x app.py
$ ./app.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader
```

Теперь вы можете запустить веб-браузер из набрать `localhost:5000` чтобы увидеть наше маленькое приложение в действии.
Просто, не так ли? Теперь мы будем конвертировать наше приложение в RESTful сервис!

Реализация RESTful сервиса на Python и Flask

Создание веб-сервиса на Flask удивительно просто, гораздо проще, чем строить полноценные серверные приложения, вроде того, которое мы делали в серии Мега-Туториал.
Есть пара хороших расширений для Flask, которые могут облегчить создание RESTful сервисов, но наша задача настолько просто, что использование расширений будет излишним.
Клиенты нашего веб-сервиса будут просить сервис добавлять, удалять и модифицировать задачи, поэтому нам нужен простой способ хранить задачи. Очевидный способ сделать это — сделать небольшую базу данных, но, поскольку база данных выходи за рамки темы статьи, мы сделаем всё гораздо проще. Чтобы больше узнать о правильном использовании БД с Flask я снова рекомендую почитать мой Мега-Туториал.

Вместо базы данных мы будем хранить список наших задач в

памяти. Это сработает, только если мы будем работать с сервером в один поток и в один процесс. Хоть для development-сервера это нормально, то для production-сервера это будет очень плохой идеей и будет лучше подумать об использовании базы данных. Сейчас мы готовы реализовать первую точку входа в наш веб-сервис:

```
#!flask/bin/python
from flask import Flask, jsonify

app = Flask(__name__)

tasks = [
    {
        'id': 1,
        'title': u'Buy groceries',
        'description': u'Milk, Cheese, Pizza, Fruit, Tylenol',
        'done': False
    },
    {
        'id': 2,
        'title': u'Learn Python',
        'description': u'Need to find a good Python tutorial on the
web',
        'done': False
    }
]

@app.route('/todo/api/v1.0/tasks', methods=['GET'])
def get_tasks():
    return jsonify({'tasks': tasks})

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

Как вы можете видеть, изменилось немногое. Мы создали в памяти задачи, которые являются не более чем простым массивом словарей. Каждая запись в массиве имеет все поля, которые мы определили выше для наших задач.

Вместо того, чтобы использовать точку входа `index`, у нас теперь есть функция `get_tasks` связанная с URI `/todo/api/v1.0/tasks`, для HTTP метода `GET`.

Вместо текста наша функция отдает JSON, в который Flask с помощью метода `jsonify` кодирует нашу структуру данных.

Использование веб-браузера, для тестирования веб-сервиса, не самая лучшая идея, т.к. с помощью веб-браузера не так просто генерировать все типы HTTP-запросов. Вместо этого мы будем использовать curl. Если `curl` у вас не установлен, лучше сделать это прямо сейчас.

Запустите веб-сервис тем же самым путем, как и демонстрационное приложение, запустив `app.py`. Теперь откройте новое окно консоли и вводите следующие команды:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 294
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 04:53:53 GMT

{
  "tasks": [
    {
      "description": "Milk, Cheese, Pizza, Fruit, Tylenol",
      "done": false,
      "id": 1,
      "title": "Buy groceries"
    },
    {
      "description": "Need to find a good Python tutorial on the
web",
      "done": false,
      "id": 2,
      "title": "Learn Python"
    }
  ]
}
```

Мы просто вызвали функцию нашего RESTful сервиса!

Сейчас давайте напишем вторую версию метода GET для наших задач. Если вы взгляните на таблицу выше, то следующим будет метод, который возвращает данные из одной задачи:

```
from flask import abort

@app.route('/todo/api/v1.0/tasks/<int:task_id>', methods=['GET'])
def get_task(task_id):
    task = filter(lambda t: t['id'] == task_id, tasks)
    if len(task) == 0:
        abort(404)
    return jsonify({'task': task[0]})
```

Вторая функция немного интересней. Здесь мы передаем через URL id задачи, и с помощью Flask транслируем в аргумент функции `task_id`.
С этим аргументом мы ищем нашу задачу в базе. Если полученный id не найдется в базе, мы вернем ошибку 404, которая по спецификации HTTP означает «Resource Not Found».
Если задача будет найдена, мы просто упакуем ее в JSON с помощью функции `jsonify` и отправим как ответ, так же как поступали раньше, отправляя коллекцию.
Вот так выглядит действие этой функции, когда мы вызываем ее с помощью `curl`:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks/2
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 151
```

```
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 05:21:50 GMT

{
  "task": {
    "description": "Need to find a good Python tutorial on the web",
    "done": false,
    "id": 2,
    "title": "Learn Python"
  }
}
$ curl -i http://localhost:5000/todo/api/v1.0/tasks/3
HTTP/1.0 404 NOT FOUND
Content-Type: text/html
Content-Length: 238
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 05:21:52 GMT

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>404 Not Found</title>
<h1>Not Found</h1>
<p>The requested URL was not found on the server.</p><p>If you
entered the URL manually please check your spelling and try again.
</p>
```

Когда мы запросили ресурс с id #2 мы получили его, но вместо ресурса с id #3 мы получили ошибку 404. Такую странную ошибку внутри HTML вместо JSON мы получили, потому, что Flask по умолчанию генерирует страницу с ошибкой 404. Так как это клиентские приложения будут всегда ожидать он нашего сервера JSON, то нам нужно изменить это поведение:

```
from flask import make_response

@app.errorhandler(404)
def not_found(error):
    return make_response(jsonify({'error': 'Not found'}), 404)
```

Так мы получим более соответствующий нашему API ответ:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks/3
HTTP/1.0 404 NOT FOUND
Content-Type: application/json
Content-Length: 26
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 05:36:54 GMT

{
  "error": "Not found"
}
```

Следующий в нашем списке метод `POST`, который мы будем использовать чтобы добавить новую задачу в нашу базу:

```
from flask import request

@app.route('/todo/api/v1.0/tasks', methods=['POST'])
def create_task():
    if not request.json or not 'title' in request.json:
        abort(400)
    task = {
        'id': tasks[-1]['id'] + 1,
        'title': request.json['title'],
        'description': request.json.get('description', ""),
        'done': False
    }
    tasks.append(task)
    return jsonify({'task': task}), 201
```

Добавление новой задачи тоже реализуется довольно просто. `request.json` содержит данные запроса, но только если они помечены как JSON. Если данных там нет, или данные на месте но отсутствует значение поля `title`, тогда возвращается код 400, который используется чтобы обозначить «Bad Request».

Затем мы создаем словарь с новой задачей, используя id последней задачи плюс 1(простой способ гарантировать уникальность id в нашей простой базе). Мы терпим отсутствие значения в поле `description`, и мы предполагаем что поле `done` при создании задачи всегда будет `False`.

Мы добавляем новую задачу к нашему массиву `tasks`, затем возвращаем клиенту сохраненную задачу и код 201, который в HTTP означает «Created».
Чтобы протестировать новую функцию мы используем следующую команду `curl`:

```
$ curl -i -H "Content-Type: application/json" -X POST -d
'{"title":"Read a book"}' http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 201 Created
Content-Type: application/json
Content-Length: 104
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 05:56:21 GMT

{
  "task": {
    "description": "",
    "done": false,
    "id": 3,
    "title": "Read a book"
  }
}
```

Примечание: если у вас Windows и вы используете Cygwin версию `curl` из `bash` тогда вышеописанная команда сработает как надо. Если вы используете нативную версию `curl` из обычно командной строки, то придется немного подшаманить с двойными кавычками:

```
curl -i -H "Content-Type: application/json" -X POST -d "
{"""title""":"""Read a book"""}"
http://localhost:5000/todo/api/v1.0/tasks
```

В Windows вы используете двойные кавычки чтобы отделить тело запроса, и внутри запроса двойные кавычки чтобы экранировать третью кавычку.
Конечно, после выполнения этого запроса мы можем получим обновленный список задач:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 423
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 05:57:44 GMT

{
  "tasks": [
    {
      "description": "Milk, Cheese, Pizza, Fruit, Tylenol",
      "done": false,
      "id": 1,
      "title": "Buy groceries"
    },
    {
      "description": "Need to find a good Python tutorial on the
web",
      "done": false,
      "id": 2,
      "title": "Learn Python"
    },
    {
      "description": "",
      "done": false,
      "id": 3,
      "title": "Read a book"
    }
  ]
}
```

Оставшиеся две функции нашего веб-сервиса будут выглядеть так:

```
@app.route('/todo/api/v1.0/tasks/<int:task_id>', methods=['PUT'])
def update_task(task_id):
    task = filter(lambda t: t['id'] == task_id, tasks)
    if len(task) == 0:
        abort(404)
    if not request.json:
        abort(400)
    if 'title' in request.json and type(request.json['title']) !=
unicode:
        abort(400)
    if 'description' in request.json and
type(request.json['description']) is not unicode:
        abort(400)
    if 'done' in request.json and type(request.json['done']) is not
bool:
        abort(400)
    task[0]['title'] = request.json.get('title', task[0]['title'])
    task[0]['description'] = request.json.get('description', task[0]
['description'])
    task[0]['done'] = request.json.get('done', task[0]['done'])
    return jsonify({'task': task[0]})

@app.route('/todo/api/v1.0/tasks/<int:task_id>', methods=['DELETE'])
def delete_task(task_id):
    task = filter(lambda t: t['id'] == task_id, tasks)
    if len(task) == 0:
        abort(404)
    tasks.remove(task[0])
    return jsonify({'result': True})
```

Функция `delete_task` без сюрпризов. Для функции `update_task` мы стараемся предотвратить ошибки делая тщательную проверку входных аргументов. Мы должны убедиться, что предоставленные клиентом данные в надлежащем формате, прежде чем запишем их в базу.

Вызов функци обновляющей задачу с id #2 будет выглядеть примерно так:

```
$ curl -i -H "Content-Type: application/json" -X PUT -d
'{"done":true}' http://localhost:5000/todo/api/v1.0/tasks/2
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 170
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 07:10:16 GMT

{
  "task": [
    {
      "description": "Need to find a good Python tutorial on the
web",
      "done": true,
      "id": 2,
      "title": "Learn Python"
    }
  ]
}
```

Улучшаем интерфейс нашего сервиса

Сейчас основная проблема дизайна нашего сервиса в том, что клиенты вынуждены строить URI самостоятельно исходя из ID задач. Этот легко, но дает знание клиенту как строятся URI для доступа к данным, что может помешать в будущем, если мы захотим внести изменения в URI.
Вместо id задачи мы вернем полный URI, через который будет осуществляться выполнение всех действий с задачей. Для этого мы напишем маленькую функцию-хелпер, которая будет генерировать «публичную» версию задачи, отправляемую клиенту:

```
from flask import url_for

def make_public_task(task):
    new_task = {}
    for field in task:
        if field == 'id':
            new_task['uri'] = url_for('get_task',
task_id=task['id'], _external=True)
        else:
            new_task[field] = task[field]
    return new_task
```

Все что мы делаем здесь это берем задачу из нашей базы данных и создаем новую задачу в которой все поля идентичны, за исключением поля `id`, которое заменено полем `uri`, сгенерированным функцией `url_for` предоставляемой Flask. Когда мы возвращаем список задач мы прогоняем все задачи через эту функцию, прежде чем отослать клиенту:

```
@app.route('/todo/api/v1.0/tasks', methods=['GET'])
def get_tasks():
    return jsonify({'tasks': map(make_public_task, tasks)})
```

Теперь клиент получает вот такой список задач:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 406
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 18:16:28 GMT

{
  "tasks": [
    {
      "title": "Buy groceries",
```

```
      "done": false,
      "description": "Milk, Cheese, Pizza, Fruit, Tylenol",
      "uri": "http://localhost:5000/todo/api/v1.0/tasks/1"
    },
    {
      "title": "Learn Python",
      "done": false,
      "description": "Need to find a good Python tutorial on the
web",
      "uri": "http://localhost:5000/todo/api/v1.0/tasks/2"
    }
  ]
}
```

Применив эту технику к остальным функциям мы сможем гарантировать, что клиент всегда получит URI, вместо id.

Защита RESTful веб-сервиса

Вы думали мы уже закончили? Конечно, мы закончили с функциональностью нашего сервиса, но у нас есть проблема. Наш сервис открыт для всех, а это не очень хорошо.
У нас есть законченый веб-сервис, который управляет нашим списком дел, но сервис, в текущем его состоянии, доступен каждому. Если незнакомец выяснит как работает наше API он или она может написать новый клиент и навести беспорядок в наших данных.
Многие руководства для начинающих игнорируют безопасность и заканчиваются здесь. По-моему это серьезная проблема, которая всегда должна быть решена.
Простой путь защитить наш веб-сервис это пускать клиентов после авторизации по логину и паролю. В обычном веб-приложении вы должны сделать форму логина, которая отправляет данные авторизации, сервер обрабатывает их и делает новую сессию, а

браузер пользователя получает куки с идентификатором сессии. К сожаление здесь мы такое сделать не можем, **stateless** — одно из правил построения REST веб-сервисов и мы должны просить клиентов отправлять свои регистрационные данные при каждом запросе.
С REST мы всегда стараемся придерживаться протокола HTTP настолько, насколько сможем. Сейчас нам нужно реализовать аутентификацию пользователя в контексте HTTP, который предоставляет нам 2 варианта — Basic и Digest.

Существует маленькое расширение Flask написанное вашим покорным слугой. Давайте установим Flask-HTTPAuth:

```
$ flask/bin/pip install flask-httpauth
```

Теперь скажем нашего веб-сервису отдавать данные только пользователю с логином `miguel` и паролем `python`. Для начала настроим Basic HTTP authentication как показано ниже:

```
from flask.ext.httpauth import HTTPBasicAuth
auth = HTTPBasicAuth()

@auth.get_password
def get_password(username):
    if username == 'miguel':
        return 'python'
    return None

@auth.error_handler
def unauthorized():
    return make_response(jsonify({'error': 'Unauthorized access'}),
401)
```

Функция `get_password` будет по имени пользователя возвращать пароль. В более сложных системах такая функцию должна будет лезть в базу, но для одного пользователя это не обязательно. Функция `error_handler` будет использоваться чтобы отправить ошибку авторизации, при неправильных данных. Так же как мы поступили с другими ошибками мы должны настроить функцию на отправку JSON, вместо HTML.

После настройки системы аутентификаци, осталось только добавить декоратор`@auth.login_required` для всех функций, которые должны быть защищены. Например:

```
@app.route('/todo/api/v1.0/tasks', methods=['GET'])
@auth.login_required
def get_tasks():
    return jsonify({'tasks': tasks})
```

Если мы попробуем запросить эту функцию с помощью `curl` мы получим примерно следующее:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 401 UNAUTHORIZED
Content-Type: application/json
Content-Length: 36
WWW-Authenticate: Basic realm="Authentication Required"
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 06:41:14 GMT

{
  "error": "Unauthorized access"
}
```

Для того, чтобы вызвать эту функцию, мы должны подтвердить наши полномочия:

```
$ curl -u miguel:python -i http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 316
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 06:46:45 GMT

{
  "tasks": [
    {
      "title": "Buy groceries",
      "done": false,
      "description": "Milk, Cheese, Pizza, Fruit, Tylenol",
      "uri": "http://localhost:5000/todo/api/v1.0/tasks/1"
    },
    {
      "title": "Learn Python",
      "done": false,
      "description": "Need to find a good Python tutorial on the
web",
      "uri": "http://localhost:5000/todo/api/v1.0/tasks/2"
    }
  ]
}
```

Расширение с аутентификацией дает нам свободу выбирать какие функции будут в общем доступе, а какие защищены.
Для защиты регистрационной информации наш веб-сервис должен быть доступен через HTTP Secure server (...) который шифрует траффик между клиентом и сервером и предотвращает получение конфиденциальной информаци третьей стороной.
К сожалению веб-браузеры имеют дурную привычку показывать

страшное диалоговое окно, когда запрос возвращается с ошибкой 401. Это происходит даже для фоновых запросов, так что если бы мы реализовали клиента для веб-браузера, нам пришлось бы прыгать через обручи, чтобы не давать браузеру показывать свои окна.
Простой путь обмануть браузер — возвращать любой другой код, вместо 401. Любимая всеми альтернатива это код 403, который означает ошибку «Forbidden». Хоть это достаточно близкая по смыслу ошибка, это нарушает стандарт HTTP, так что это неправильно. В частности будет хорошим решением не использовать веб-браузер в качестве клиентского приложения. Но в случаях, когда сервер и клиент разрабатываются совместно это спасает от многих неприятностей. Чтобы провернуть этот трюк нам нужно просто заменить код ошибки с 401 на 403:

```
@auth.error_handler
def unauthorized():
    return make_response(jsonify({'error': 'Unauthorized access'}),
403)
```

В клиентском приложении нужно тоже отлавливать ошибку 403.

Возможные улучшения

Есть несколько возможностей улучшить разработанный нами сегодня веб-сервис.
Для начала, настоящий веб-сервис должен общаться с настоящей базой данных. Структура данных в памяти очень ограниченный способ хранения данных и он не должен использоваться в реальных приложениях.
Другой способ улучшить приложение это поддержка нескольких

пользователей. Если система поддерживает несколько пользователей, то данные аутентификации могут использоваться чтобы возвращать персональные списки пользователям. В такой системе пользователи станут вторым ресурсом. Запрос `POST` будет регистрировать нового пользователя в системе. Запрос `GET` может возвращать информацию о пользователе. Запрос `PUT` может обновлять информацию о пользователе, например email. Запрос `DELETE` будет удалять пользователя из системы.

Запрос `GET`, который возвращает список задач, может быть расширен несколькими способами. Для начала это запрос может иметь опциональные агрументы, такие как количество задач на страницу. Другой путь сделать функцию более удобной это добавить критерии фильтрации. Например клиент может запросить только выполненые задачии или задачи, заголовок которых начинается с определенной буквы. Все эти элементы могут быть добавлены в URL как аргументы.

Вывод

Законченый код для веб-сервиса To Do List вы можете взять здесь: https://gist.github.com/miguelgrinberg/5614326.

Я верю что это было простое и дружелюбное введение в RESTful API. Если есть достаточный инетерес я мог бы написать вторую часть этой статьи, в которой мы разработаем простой веб-клиент для нашего сервиса.

Я сделал клиента для нашего сервиса:Writing a Javascript REST client.

Статья о таком же сервере, но с использованием Flask-RESTfulDesigning a RESTful API using Flask-RESTful.

Miguel

Проголосовать:

+26

Поделиться:

Сохранить:

Комментарии (31)

Похожие публикации

Реализация REST API на Symfony2: правильный путь

25

ПЕРЕВОД

Fesor • 3 августа 2012 в 16:24

Создание RESTful API в Google App Engine на основе Flask

15

xSkyFoXx • 9 апреля 2012 в 17:06

Важные аспекты RESTful API для вашего проекта

47

matriks • 29 ноября 2010 в 14:25

Популярное за сутки

Яндекс открывает Алису для всех разработчиков. Платформа Яндекс.Диалоги (бета) 69

BarakAdama • вчера в 10:52

Почему следует игнорировать истории основателей успешных стартапов 20

ПЕРЕВОД

m1rko • вчера в 10:44

Как получить телефон (почти) любой красотки в Москве, или интересная особенность MT_FREE 24

ИЗ ПЕСОЧНИЦЫ

cab404 • вчера в 20:27

Java и Project Reactor 10

zealot_and_frenzy • вчера в 10:56

Пользовательские агрегатные и оконные функции в PostgreSQL и Oracle 6

erogov • вчера в 12:46

Лучшее на Geektimes

Как фермеры Дикого Запада организовали телефонную сеть на колючей проволоке

31

NAGru • вчера в 10:10

Энтузиаст сделал новую материнскую плату для ThinkPad X200s

49

alizar • вчера в 15:32

Кто-то посылает секс-игрушки с Amazon незнакомцам. Amazon не знает, как их остановить

85

Pochtoycom • вчера в 13:06

Илон Маск продолжает убеждать в необходимости создания колонии людей на Марсе

139

marks • вчера в 14:19

Дела шпионские (часть 1)

16

TashaFridrih • вчера в 13:16