Distributed Systems

03r. Python Web Services Programming Tutorial

Paul Krzyzanowski
TA: Long Zhao
Rutgers University
Fall 2017
From Web Browsing to Web Services

• Web browser:
  – Dominant model for user interaction on the Internet

• Not good for programmatic access to data or manipulating data
  – UI is a major component of the content
  – *Site scraping* is a pain!
Web Services

• We wanted:
  – Remotely hosted services – that programs can use
  – Machine-to-machine communication

• Problems
  – Web pages are content-focused
  – Traditional RPC solutions usually used a range of ports
    • And we need more than just RPC sometimes
  – Many RPC systems didn’t work well across languages
  – Firewalls restrict ports & may inspect the protocol
  – No support for load balancing
Web Services

• Set of protocols by which services can be published, discovered, and used in a technology neutral form
  – Language & architecture independent

• Applications will typically invoke multiple remote services
  – Service Oriented Architecture (SOA)
  – SOA = Programming model

• General principles
  – Payloads are text (XML or JSON)
    • Technology-neutral
  – HTTP used for transport
    • Use existing infrastructure: web servers, firewalls, load-balancers
REST

• REST stands for REpresentational State Transfer

• REST was first introduced by Roy Fielding in year 2000

• REST is a web standards based architecture
  – Uses HTTP Protocol for data communication
  – Resource-oriented
    • every component is a resource
    • a resource is accessed by a common interface using HTTP standard methods
REST

• REST Server
  – simply provides access to resources

• REST client
  – accesses and presents the resources

• REST resources
  – each resource is identified by URIs/ Global IDs
  – representations of a resource
    • Text, JSON and XML
    • JSON is now the most popular format
RESTful Web Services

• A web service is:
  – A collection of open protocols
  – Standards used for exchanging data between applications or systems
  – Interoperability between different languages (Java and Python) or platforms (Windows and Linux)

• Web services based on REST Architecture are known as RESTful Web Services
  – Use HTTP methods to implement the concept of REST architecture
  – URI (Uniform Resource Identifier) to define a RESTful service
  – Resources representation: JSON
Everything Is a Resource

• Any interaction of a RESTful API is an interaction with a resource.

• Resources are sources of information, typically documents or services.

• A user can be thought of as resource and thus has an URL such as in the case of GitHub:

  https://api.github.com/users/lrei
Everything Is a Resource

- Resources can have different **representations**. The above mentioned user has the following JSON representation (partial document):

```json
{
  "login": "lrei",
  "created_at": "2008-11-21T14:48:42Z",
  "name": "Luis Rei",
  "email": "me@luisrei.com",
  "id": 35857,
  "blog": "http://luisrei.com"
}
```
Everything Is a Resource

• Resources are Nouns
  – If I want to delete a post whose ID is 233:

    http://api.example.com/posts/delete/233/

  – The correct way:

    Send a **DELETE** HTTP request to the URL:

    http://api.example.com/posts/233/
HTTP Methods

- The following HTTP methods are most commonly used in a REST based architecture.

- **GET** – Provides a read only access to a resource.
- **PUT** – Used to create a new resource.
- **DELETE** – Used to remove a resource.
- **POST** – Used to update an existing resource or create a new resource.
- **OPTIONS** – Used to get the supported operations on a resource.
Implementing RESTful Web APIs with Python & Flask
Flask

- Flask is a microframework for Python based on Werkzeug, a WSGI utility library.

- Flask is a good choice for a REST API because it is:
  - Written in Python;
  - Simple to use;
  - Flexible;
  - Multiple good deployment options;
  - RESTful request dispatching.
RESTful Web APIs with Python & Flask

• To install:

```shell
>> (sudo) pip install flask
```

• We use the `curl` command to make test requests.
  – `curl` is a command that lets you transfer data to or from a server using several protocols, most commonly HTTP
    See https://curl.haxx.se

• Note: the iLab systems already have flask and python installed
Let's begin by making a complete app that responds to requests at the root, /articles and /articles/:id.

```python
from flask import Flask, url_for
app = Flask(__name__)

@app.route('/
def api_root():
    return 'Welcome\n
@app.route('/articles')
def api_articles():
    return 'List of ' + url_for('api_articles') + '\n'
...
```
Let's begin by making a complete app that responds to requests at the root, /articles and /articles/:id.

```python
... 
@app.route('/articles/<articleid>')
def api_article(articleid):
    return 'You are reading ' + articleid + 'n'

if __name__ == '__main__':
    app.run()
```
RESTful Web APIs with Python & Flask

• You can use curl to make the requests using:

```bash
>> curl http://127.0.0.1:5000/
```

• The responses will be, respectively,

```bash
>> curl http://127.0.0.1:5000/
GET /
Welcome

>> curl http://127.0.0.1:5000/articles
GET /articles
List of /articles

>> curl http://127.0.0.1:5000/articles/123
GET /articles/123
You are reading 123
GET Parameters

• Let’s begin by making a complete app that responds to requests at /hello and handles an optional GET parameter

```python
from flask import request

@app.route('/hello')
def api_hello():
    if 'name' in request.args:
        return 'Hello ' + request.args['name'] + '
    else:
        return 'Hello John

if __name__ == '__main__':
    app.run()
```
GET Parameters

• The server will reply in the following manner:

```plaintext
>> curl http://127.0.0.1:5000/hello
GET /hello
Hello John

>> curl http://127.0.0.1:5000/hello?name=Peter
GET /hello?name=Peter
Hello Peter
```
Request Methods (HTTP Verbs)

- Let’s modify the to handle different HTTP verbs:

```python
@app.route('/echo', methods=['GET', 'POST', 'PUT', 'DELETE'])
def api_echo():
    if request.method == 'GET':
        return "ECHO: GET\n"

    elif request.method == 'POST':
        return "ECHO: POST\n"

    elif request.method == 'PUT':
        return "ECHO: PUT\n"

    elif request.method == 'DELETE':
        return "ECHO: DELETE\n"
```
Request Methods (HTTP Verbs)

• To curl the -X option can be used to specify the request type:

  ```shell
  >> curl -X POST http://127.0.0.1:5000/echo
  ```

• The replies to the different request methods will be:

  ```
  GET /echo
  ECHO: GET

  POST /echo
  ECHO: POST
  ...
  ```
Request Data & Headers

- Usually POST is accompanied by data. And sometimes that data can be in one of multiple formats: plain text, JSON, XML, your own data format, a binary file.

- Accessing the HTTP headers is done using the `request.headers` dictionary ("dictionary-like object") and the request data using the `request.data` string. If the mimetype is `application/json`, `request.json` will contain the parsed JSON.
• Usually POST is accompanied by data. And sometimes that data can be in one of multiple formats: plain text, JSON, XML, your own data format, a binary file.

```python
from flask import json

@app.route('/messages', methods = ['POST'])
def api_message():
    if request.headers['Content-Type'] == 'text/plain':
        return "Text Message: " + request.data + '
'
    elif request.headers['Content-Type'] == 'application/json':
        return json.dumps(request.json)

    else:
        return "415 Unsupported Media Type ;)"
```
To specify the content type with `curl`:

```
>> curl -H "Content-type: application/json" -X POST \ 
   http://127.0.0.1:5000/messages -d '{"message": "Hello Data"}"
```

The replies to the different content types will be:

POST /messages "Hello Data"
Content-type: text/plain
Text Message: Hello Data

POST /messages {"message": "Hello Data"}
Content-type: application/json
{"message": "Hello Data"}
Responses

• Responses are handled by Flask's `Response` class:

```python
from flask import Response

@app.route('/hello', methods = ['GET'])
def api_hello():
    data = { 'hello': 'world', 'number': 3 }
    js = json.dumps(data)
    resp = Response(js, status=200, mimetype='application/json')
    return resp
```
Responses

• To view the response HTTP headers using curl, specify the `-i` option:

   >> curl -i http://127.0.0.1:5000/hello

• The response returned by the server, with headers included, will be:

  GET /hello
  HTTP/1.0 200 OK
  Content-Type: application/json
  Content-Length: <…>
  Server: <…>
  Date: <…>
  { "hello": "world", "number": 3 }
Status Codes & Errors

• 200 is the default status code reply for GET requests, in both of these examples. There are certain cases where overriding the defaults is necessary: error handling.
@app.errorhandler(404)
def not_found(error=None):
    message = {
        'status': 404,
        'message': 'Not Found:' + request.url
    }
    resp = jsonify(message)
    resp.status_code = 404
    return resp

@app.route('/users/<userid>', methods=['GET'])
def api_users(userid):
    users = {'1': 'john', '2': 'steve', '3': 'bill'}
    if userid in users:
        return jsonify({userid: users[userid]})
    else:
        return not_found()
Status Codes & Errors

• This produces:

GET /users/2
HTTP/1.0 200 OK { "2": "steve" }

GET /users/4
HTTP/1.0 404 NOT FOUND
{
    "status": 404,
    "message": "Not Found: http://127.0.0.1:5000/users/4"
}
Other Useful Links

- iLab: [https://www.cs.rutgers.edu/resources/instructional-lab](https://www.cs.rutgers.edu/resources/instructional-lab)


- Flask Quick Start: [http://flask.pocoo.org/docs/0.12/quickstart/](http://flask.pocoo.org/docs/0.12/quickstart/)

The end