Building RESTful Services with Python and Flask

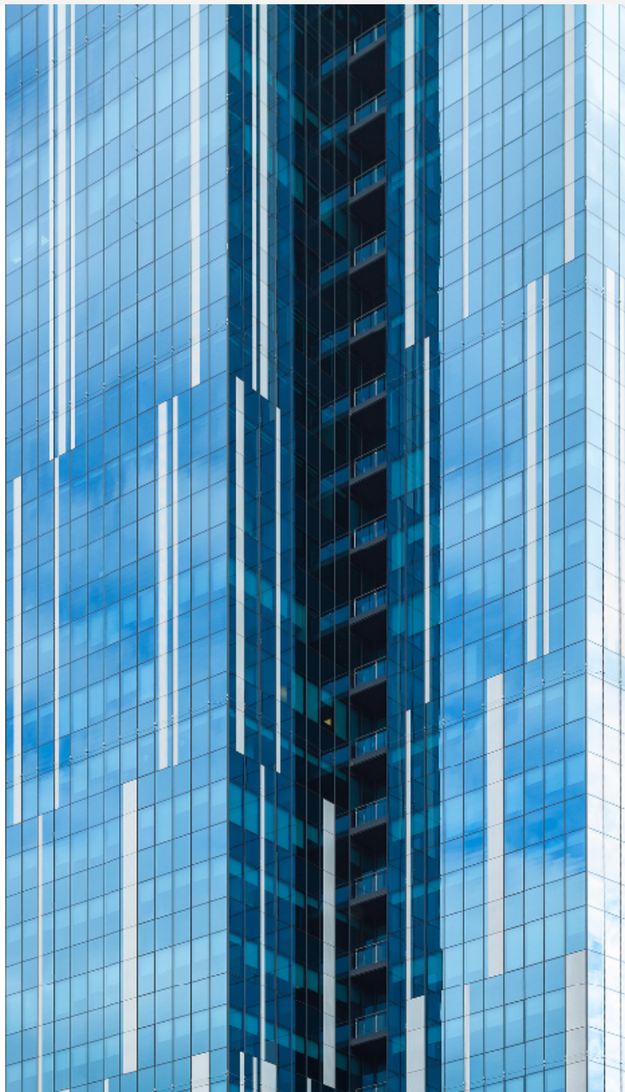
Instructor:

John J Rofrano

Senior Technical Staff Member, DevOps Champion
IBM T.J. Watson Research Center
rofrano@us.ibm.com (@JohnRofrano)

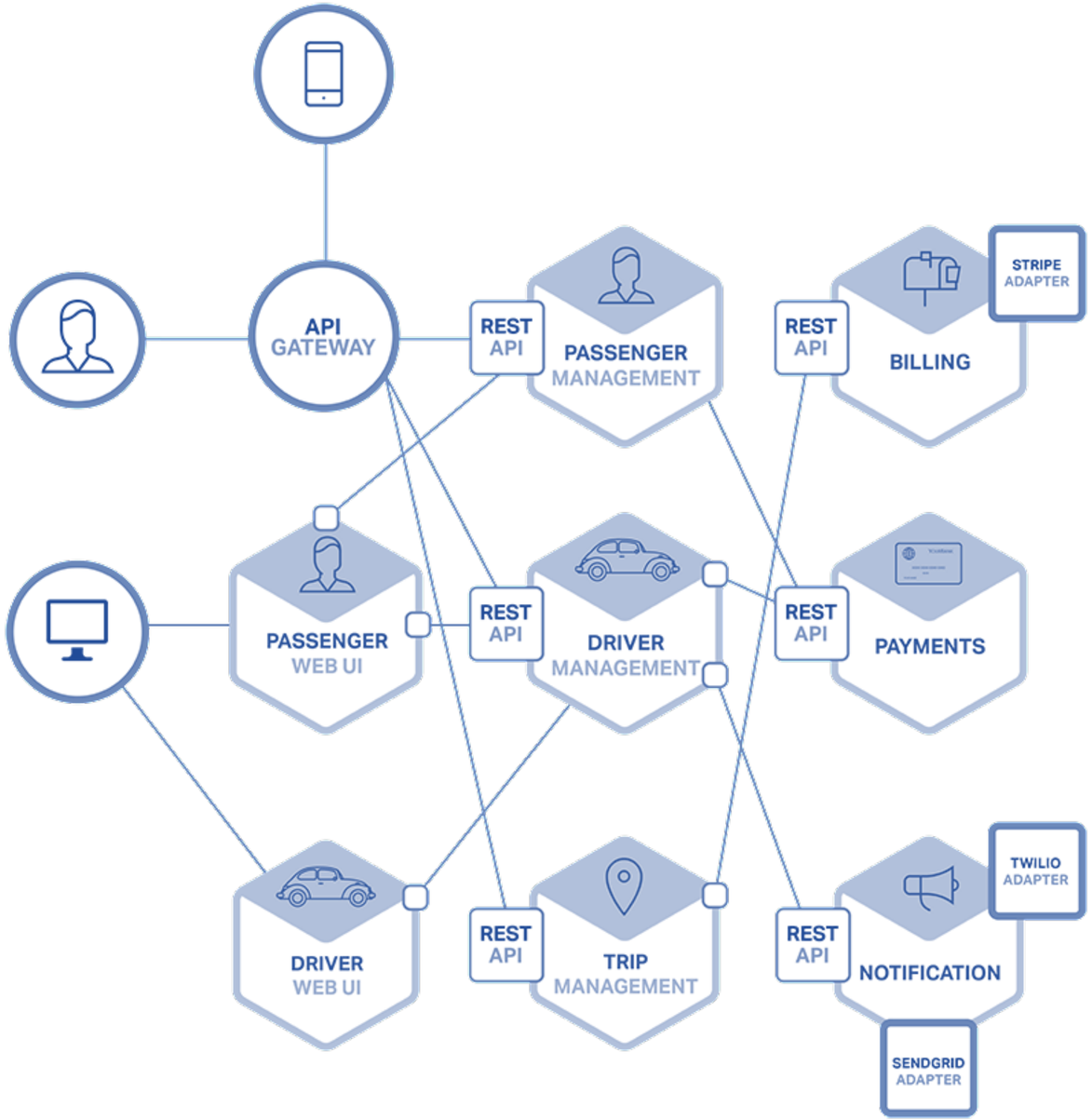
What Will You Learn?

- Understand what a REST API is
- Understand the Guidelines and Best Practices
- How to build a more RESTful service
- How to use Python with Flask to build a REST API



Cloud Native Applications

- The Twelve-Factor App describes patterns for cloud-native architectures which leverage microservices.
- Applications are designed as a collection of stateless microservices.
- State is maintained in separate databases and persistent object stores.
- Resilience and horizontal scaling is achieved through deploying multiple instances.
- Failing instances are killed and re-spawned, not debugged and patched.
- DevOps pipelines help manage continuous delivery of services.



https://www.nginx.com/blog/introduction-to-microservices/

REST Architecture and RESTful Services

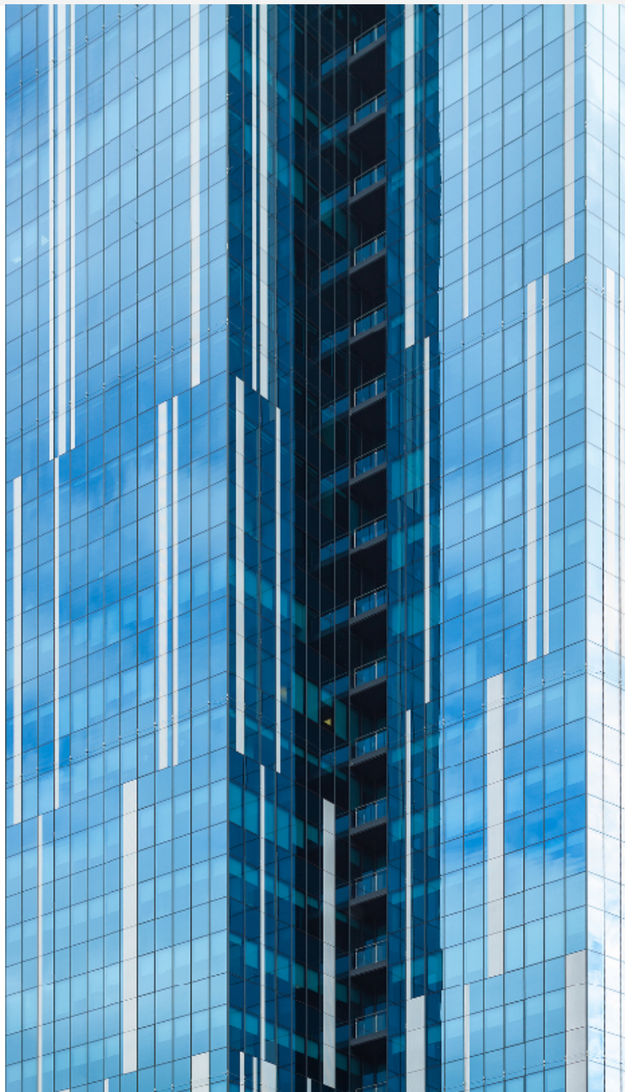
“Microservices need a consistent way of communicating.”

REST Architecture

- REST is a **client-server** architecture
  - The client and the server provide a separation of concerns which allows both the client and the server to evolve independently as it only requires that the interface stays the same
- REST is **stateless**
  - The communication between the client and the server always contains all the information needed to perform the request. There is no session state in the server.
- REST is **cacheable**
  - The client, the server can cache resources in order to improve performance
- REST provides a **uniform interface** between components
  - All components follow the same rules to speak to one another
- REST is a **layered system**
  - Individual components cannot see beyond the immediate layer with which they are interacting

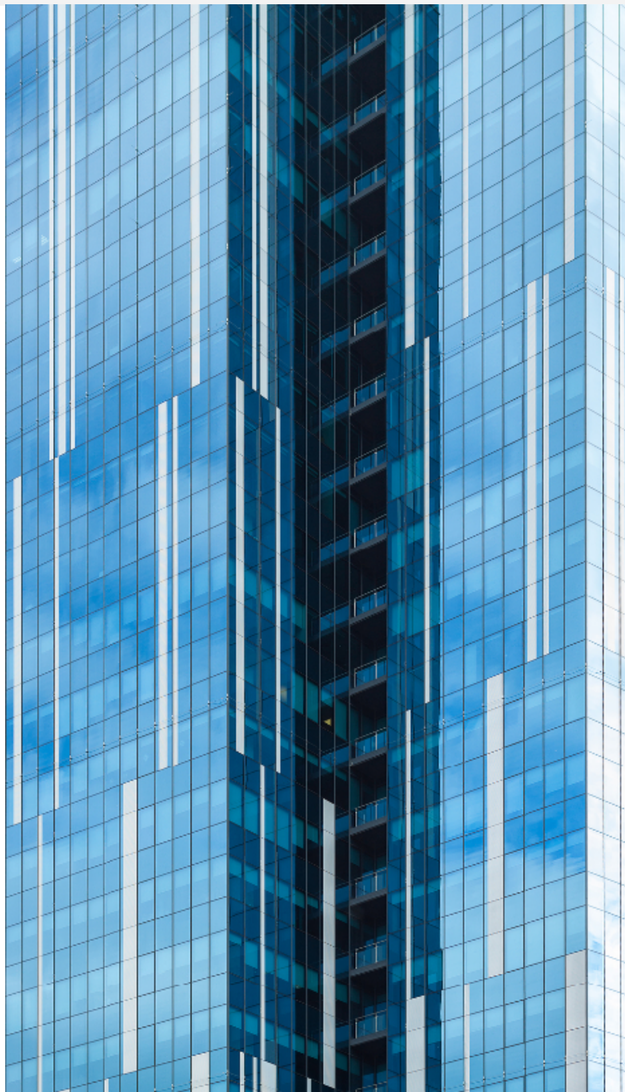
What Does Resource Based Mean?

- Things vs Actions
- Nouns vs Verbs
- Not a Remote Procedure Call mechanism
- Everything is Identified by URI’s
  - Multiple URI’s can manipulate the same Resource using different HTTP Verbs



REST Provides a Uniform Interface

- Simplifies and decouples the architecture
- Fundamental to RESTful design is an interface that almost be guessed
  - Perform CRUD on Resources
- Uses HTTP verbs (`POST, GET, PUT, DELETE`)
- Uses URL's to address resources
- Uses HTTP Response (status, body)



Example REST API for USER Resource

- Using the example of a User as a resource the API would be:

```
GET  /users                 <- retrieve a list of Users
GET  /users?name="sam" <- retrieve a list of Users with a
                            Name of "sam"
GET  /users/123             <- retrieves the User with Id 123
POST /users                 <- creates a new User
PUT  /users/123             <- updates the User with Id 123
DELETE /users/123           <- deletes the User with Id 123
```

What REST is Not

- REST is not an RPC call or just a bunch of verbs as URI's
- For example, these are NOT RESTful API's:

```
GET  http://api.myapp.com/getUser/123
POST http://api.myapp.com/addUser
GET  http://api.myapp.com/removeUser/123
```

- These are the RESTful equivalents:

```
GET    http://api.myapp.com/users/123
POST   http://api.myapp.com/users
DELETE http://api.myapp.com/users/123
```

Guidelines for Well Formed URI's

- A plural noun should be used for collection names
  - e.g., `/users` not `/user`

Guidelines for Well Formed URI's

- A plural noun should be used for collection names
  - e.g., `/users` not `/user`
- A singular noun should be used for document names
  - e.g., `/users/123/agreement`

Guidelines for Well Formed URI's

- A plural noun should be used for collection names
  - e.g., `/users` not `/user`
- A singular noun should be used for document names
  - e.g., `/users/123/agreement`
- Variable path segments may be substituted with identity-based values
  - e.g., `/users/123/addresses/2`

Guidelines for Well Formed URI's

- A plural noun should be used for collection names
  - e.g., `/users` not `/user`
- A singular noun should be used for document names
  - e.g., `/users/123/agreement`
- Variable path segments may be substituted with identity-based values
  - e.g., `/users/123/addresses/2`
- CRUD function names should not be used in URIs
  - e.g., never use: `/getUsers/123`

URI Format Guidelines

- Forward slash separator (/) must be used to indicate a hierarchical relationship

```
http://api.myapp.com/users/{id}/addresses
```

URI Format Guidelines

- Forward slash separator (/) must be used to indicate a hierarchical relationship

```
http://api.myapp.com/users/{id}/addresses
```

- A trailing forward slash (/) should not be included in URIs

```
http://api.myapp.com/users/   <- not recommended
http://api.myapp.com/users    <- recommended
```

URI Format Guidelines

- Forward slash separator (/) must be used to indicate a hierarchical relationship

```
http://api.myapp.com/users/{id}/addresses
```

- A trailing forward slash (/) should not be included in URIs

```
http://api.myapp.com/users/   <- not recommended
http://api.myapp.com/users    <- recommended
```

- Hyphens (-) should be used to improve the readability of URIs

```
http://api.myapp.com/userGroups     <- not recommended
http://api.myapp.com/user-groups    <- recommended
```

URI Format Guidelines (cont)

- Underscores (_) should not be used in URIs

```
http://api.myapp.com/user_groups     <- not recommended
http://api.myapp.com/user-groups     <- recommended
```

URI Format Guidelines (cont)

- Underscores (_) should not be used in URIs

```
http://api.myapp.com/user_groups     <- not recommended
http://api.myapp.com/user-groups     <- recommended
```

- Lowercase letters should be preferred in URI paths

```
http://api.myapp.com/Groups/Reset     <- not recommended
http://api.myapp.com/groups/reset     <- recommended
```

URI Format Guidelines (cont)

- Underscores (_) should not be used in URIs

```
http://api.myapp.com/user_groups     <- not recommended
http://api.myapp.com/user-groups     <- recommended
```

- Lowercase letters should be preferred in URI paths

```
http://api.myapp.com/Groups/Reset     <- not recommended
http://api.myapp.com/groups/reset     <- recommended
```

- File extensions should not be included in URIs

```
http://api.myapp.com/users/123/adresses.json     <- not recommended
http://api.myapp.com/users/123/adresses          <- recommended
```

What About Subordinates

- Subordinate resources can be addressed with multiple resources and ids in the URI:

```
GET /resource/{id}/subordinate/{id}
```

- Example: A User has multiple Addresses. Get Address with id = 2

```
GET /users/123/addresses/2
```

Create with POST

- Use POST to Create a Resource
- If a resource has been created on the origin server, the response **SHOULD be 201** (Created) and contain an entity which describes the status of the request and refers to the new resource, and a Location header
- Hypertext Transfer Protocol -- HTTP/1.1
  - https://www.ietf.org/rfc/rfc2616.txt

Idempotence

- **PUT** and **DELETE** operations are **idempotent**
- If you DELETE a resource, it’s gone. If you DELETE it again, it’s still gone (i.e., do not throw a 404 Not Found error!)
- Some systems might want to send a 404 on DELETE if the original resource never existed (but this could be difficult to prove)
- If you use PUT to change the state of a resource, you can resend the PUT request and the resource state should remain the same.

Side-Effects

- GET must NEVER modify the resource
  - Never code: `GET https://api.del.icio.us/posts/delete`
- GET must have no side-effects (*see above*)
- GET should return a representation of the resource(s) it was called on... period!

Safety

- While GET means "Read"... (and is safe)
- POST is neither safe nor idempotent
- Making two identical POST requests to a resource will probably result in creating two resources containing the same information
- With overloaded POST’s that may have different behavior based on data passed in, all bets are off

Use HTTP Headers

- Information about what to send or accept should be expressed in HTTP headers
- Some examples are:

```
{
    'Accept': 'application/json',
    'Content-Type': 'application/json',
    'Authorization': 'bearer %s' % token
}
```

The Root URL

- The root url '/' should return helpful information about the API:

```
200 OK
{
    "url": "https://api.spire.io/",
    "resources": {
        "sessions": {
            "url": "https://api.spire.io/sessions"
        },
        "accounts": {
            "url": "https://api.spire.io/accounts"
        },
        "billing": {
            "url": "https://api.spire.io/billing"
        }
    }
}
```

What About Actions?

- Actions are more like RPC than REST
- REST doesn’t prescribe how to handle actions but there are best practices
- But some times you need to make an action based call on a resource
  - e.g., Start, Stop, Reboot, Shutdown, Register, Deregister

Actions as A URI

- Let’s say you want to reboot a resource that is running
- If the resource doesn’t have a representation of it’s state that you can change, you could create a URI for the action
- **Example:** Reboot server with id 123

```
PUT http://api.myapp.com/servers/123/reboot
```

Create Useful Error Messages

- The more you can tell the client about what went wrong the better
- If there is a required parameter like "name" that is missing:
  - Rather than returning an error message that says: '`missing parameter`'
  - Return more information like: '`required name parameter is missing`'
- This will give the caller more information about why their call failed

Error Message Example

An example from the twilio api:

```
GET https://api.twilio.com/2010-04-01/Accounts.json
```

results in:

```
401 Unauthorized
WWW-Authenticate: Basic realm="Twilio API"
{
     "code": 20003,
     "detail": "Your AccountSid or AuthToken was incorrect.",
     "message": "Authentication Error - No credentials provided",
     "more_info": "https://www.twilio.com/docs/errors/20003",
     "status": 401
}
```

Use Proper HTTP Return Codes

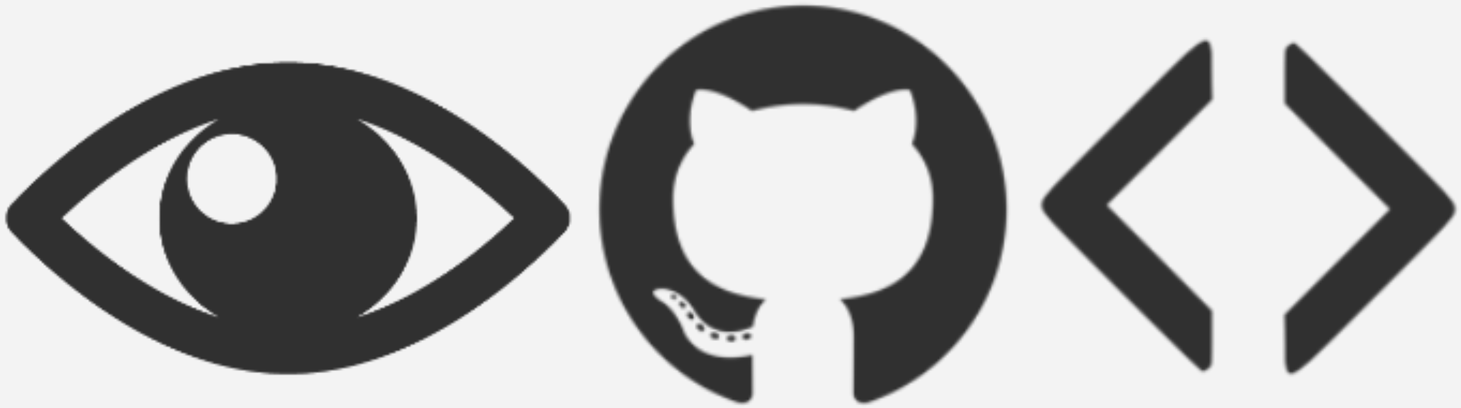
200	Indicates that the request was **completed** successfully. All GET requests that are successful should return 200
201	Indicates that a record was **created** successfully. All POST requests that successfully create a resource should return 201
204	Indicates that a record was **deleted** successfully. All DELETE requests that completed successfully should return 204 and the body should be empty.
40X (401, 404)	Status codes in the 400 range indicate a client error, such as 400 for invalid request syntax and 401 Unauthorized for not having proper credentials are probably the most common.
50X (500, 503)	Status codes in the 500 range indicate that a server error occurred. The client request may have been valid or invalid, but a problem occurred on the server that prevented it from processing the request.

Responses (Happy Path)

- `GET /users`
  - 200 + Array of Users [{...},{...},{...}]
- `GET /users/12345`
  - 200 + User {...}
- `POST /users`
  - 201 + User {...}
  - Location Header for GET
- `PUT /users/12345`
  - 200 + User {...}
- `DELETE /users/12345`
  - 204 + empty body

Common REST API Return Codes

Code: 200 (“OK”) should be used to indicate nonspecific success
Code: 200 (“OK”) must not be used to communicate errors in the response body
Code: 201 (“Created”) must be used to indicate successful resource creation
Code: 202 (“Accepted”) must be used to indicate successful start of an asynchronous action
Code: 204 (“No Content”) should be used when the response body is intentionally empty
Code: 301 (“Moved Permanently”) should be used to relocate resources
Code: 302 (“Found”) should not be used
Code: 303 (“See Other”) should be used to refer the client to a different URI
Code: 304 (“Not Modified”) should be used to preserve bandwidth
Code: 307 (“Temporary Redirect”) should be used to tell clients to resubmit the request to another URI
Code: 400 (“Bad Request”) may be used to indicate nonspecific failure
Code: 401 (“Unauthorized”) must be used when there is a problem with the client’s credentials
Code: 403 (“Forbidden”) should be used to forbid access regardless of authorization state
Code: 404 (“Not Found”) must be used when a client’s URI cannot be mapped to a resource
Code: 405 (“Method Not Allowed”) must be used when the HTTP method is not supported
Code: 406 (“Not Acceptable”) must be used when the requested media type cannot be served
Code: 409 (“Conflict”) should be used to indicate a violation of resource state
Code: 412 (“Precondition Failed”) should be used to support conditional operations
Code: 415 (“Unsupported Media Type”) must be used when the media type of a request’s payload cannot be processed
Code: 422 ("Unprocessable entity") must be used if the server can not process the property, for example, if an image cannot be formatted or if required fields are missing from the payload
Code: 500 (“Internal Server Error”) should be used to indicate API malfunction



Let's look at some RESTful Code!

Hands-On

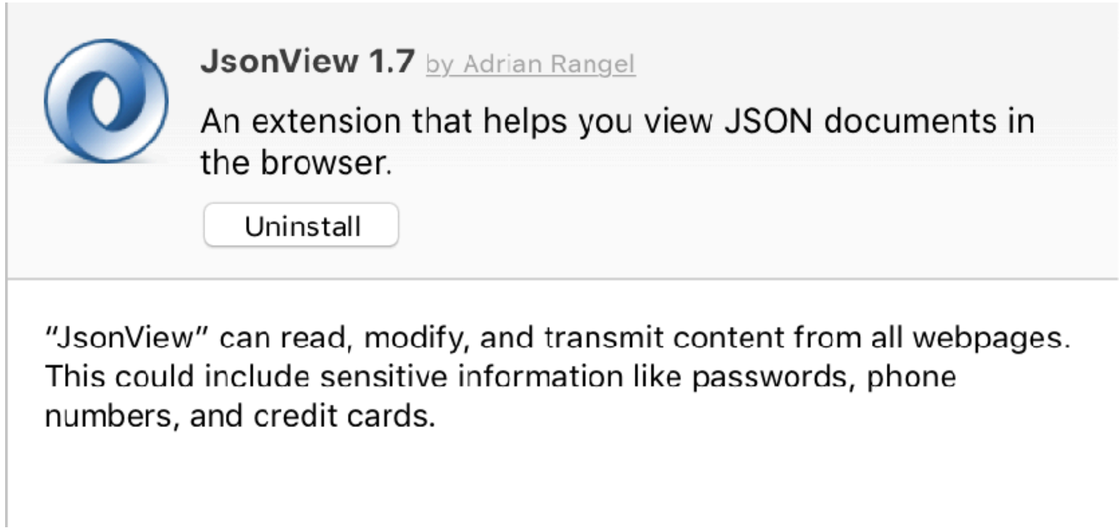
“live session”

Some Assembly Required

- Tools you will need to complete this lab:
  - Github Account (github.com)
  - Git Client
  - Text Editor (...i like atom.io)
  - Vagrant and VirtualBox
  - (Optional) JsonView, Simple REST Client

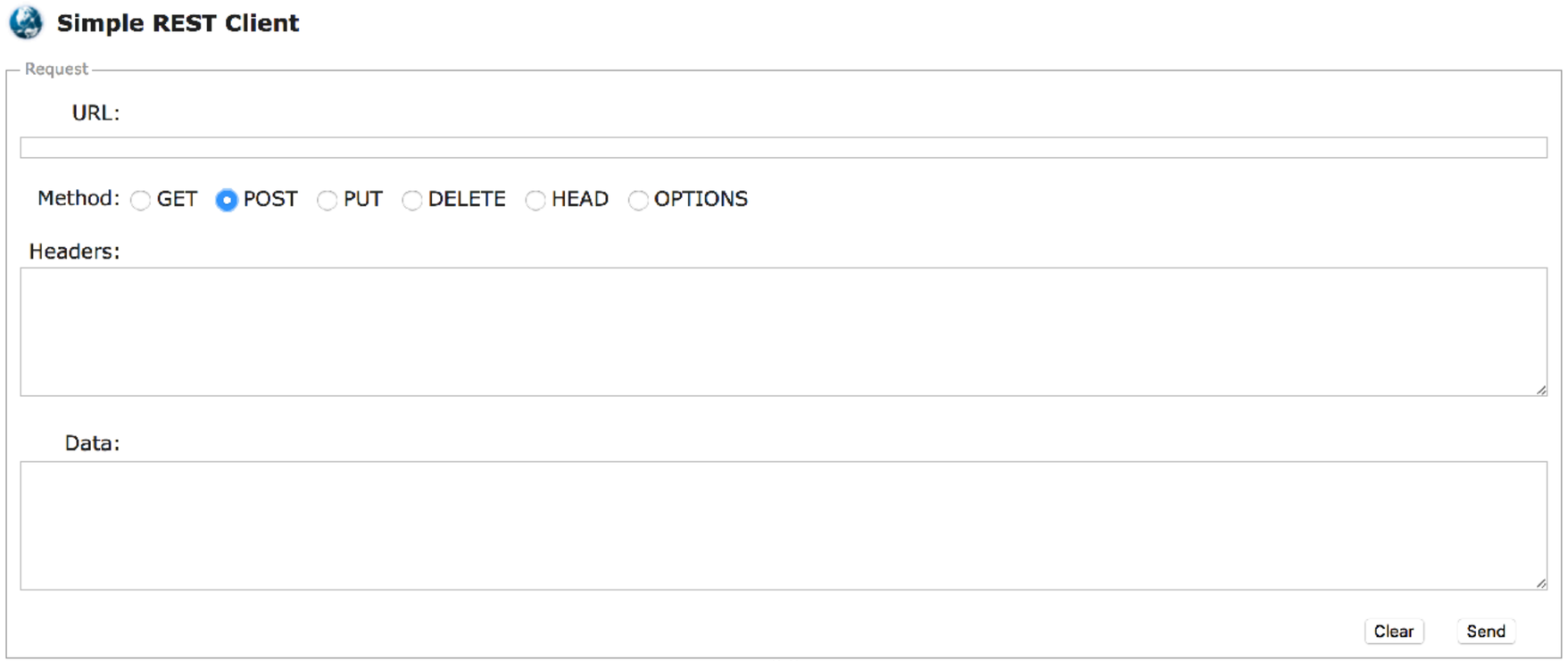
Optional Browser Plugin

- **JsonView** — will format Json output so that even humans can read it
- Browser extensions for Safari, Chrome, Firefox
- When working with REST APIs it comes in handy



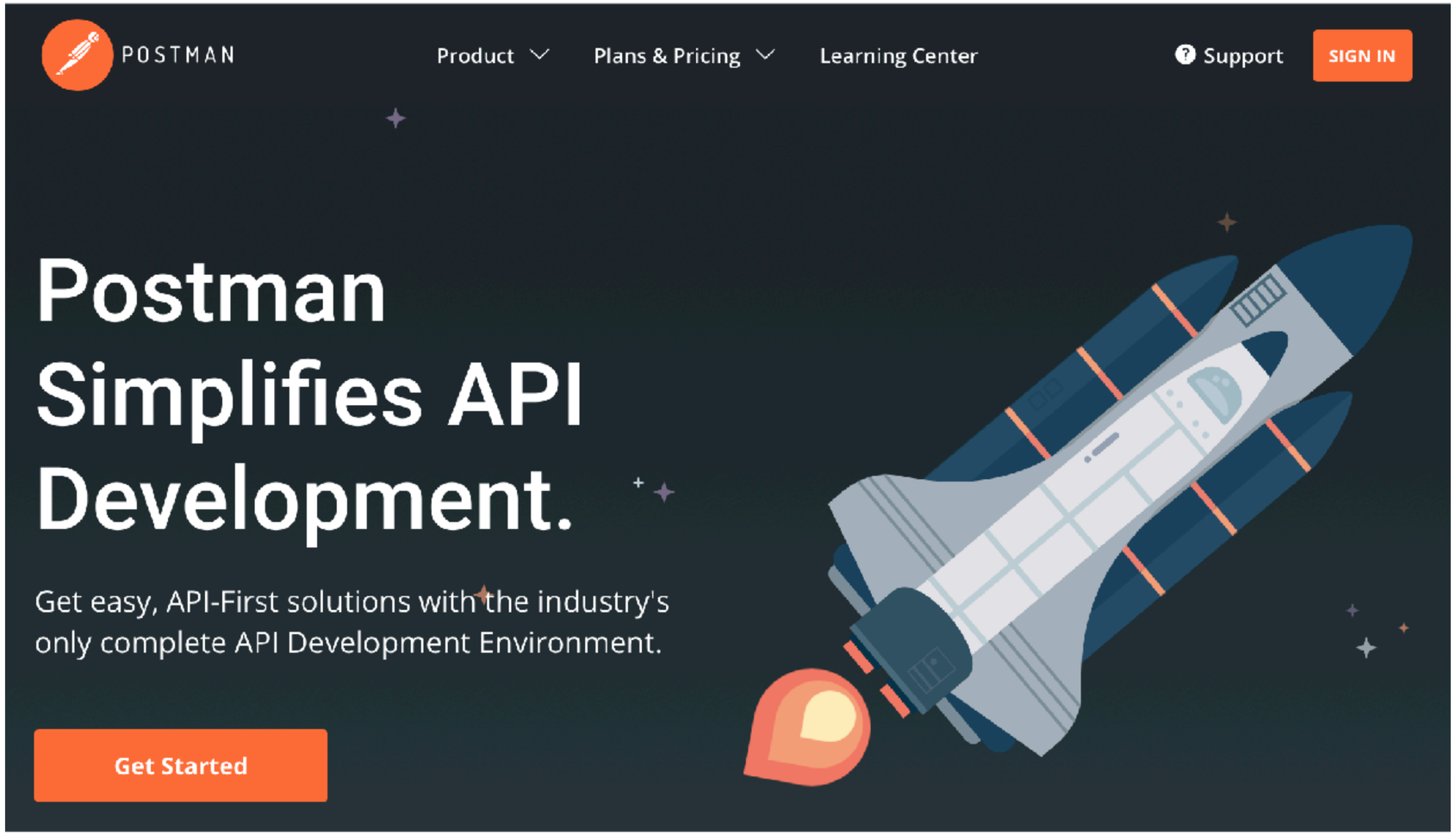
Optional Browser Plugin

- **Simple REST Client** allows you to make POST, PUT, DELETE calls in addition to GET (which your browser can do)



Postman REST Client

https://www.getpostman.com



Hello Flask

- Flask is a light-weight web framework that is idea for microservices

pastebin.com/WmzQ6FVT

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return 'Hello Flask!'

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

Hello Flask

- Flask is a light-weight web framework that is idea for microservices

pastebin.com/WmzQ6FVT

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return 'Hello Flask!'

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

Import Flask module

Hello Flask

- Flask is a light-weight web framework that is idea for microservices

pastebin.com/WmzQ6FVT

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return 'Hello Flask!'

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

Import Flask module

Create an instance of Flask app

Hello Flask

- Flask is a light-weight web framework that is idea for microservices

pastebin.com/WmzQ6FVT

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return 'Hello Flask!'

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

Import Flask module

Create an instance of Flask app

Create a route (url)

Hello Flask

- Flask is a light-weight web framework that is idea for microservices

pastebin.com/WmzQ6FVT

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return 'Hello Flask!'

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

Import Flask module

Create an instance of Flask app

Create a route (url)

Define a function to handle the route

Hello Flask

- Flask is a light-weight web framework that is idea for microservices

pastebin.com/WmzQ6FVT

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return 'Hello Flask!'

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

Import Flask module

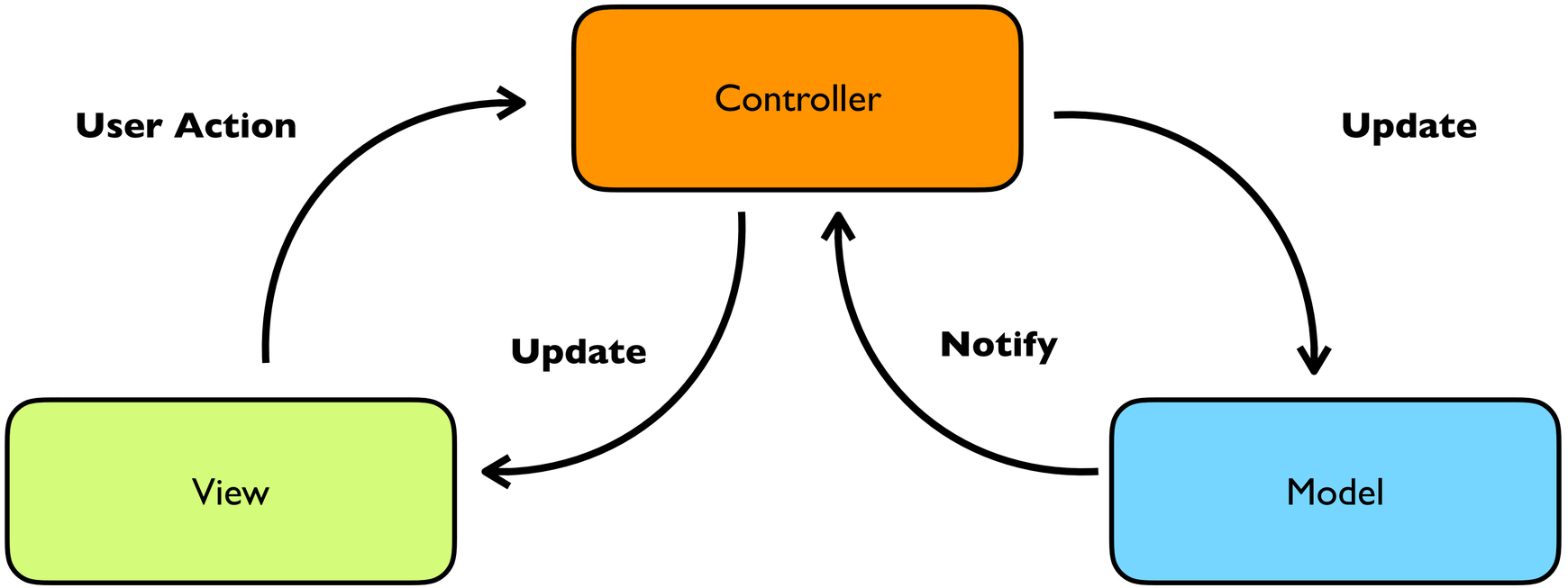
Create an instance of Flask app

Create a route (url)

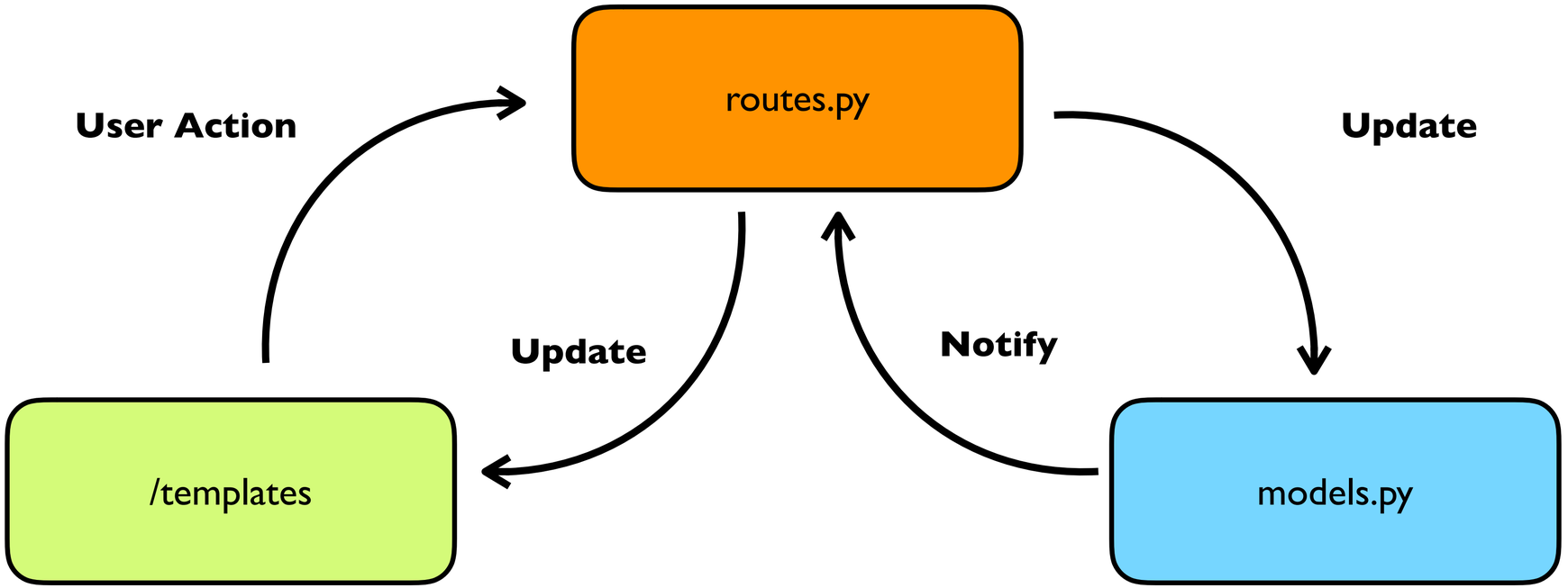
Define a function to handle the route

Run the web server

Model / View / Controller



MVC with Flask



Simple RESTful Pet Service

Allows lifecycle operations on a collection of pets

Config from Environment

- The main routine gets it’s configuration information like what port to bind to from the environment in true 12-factor style

```
# Pull options from environment
DEBUG = (os.getenv('DEBUG', 'False') == 'True')
PORT = os.getenv('PORT', '5000')
HOST = os.getenv('HOST', '0.0.0.0')

. . . CODE HERE . . .

if __name__ == "__main__":
   app.run(host=HOST, port=int(PORT), debug=DEBUG)
```

Root URL Request

- This code is called on the root URL

```
@app.route('/')
def index():
    return jsonify(name='Pet Demo REST API Service',
                   version='1.0',
                   url=url_for('list_pets', _external=True)), status.HTTP_200_OK
```

- It returns the following JSON

```
{
  "name": "Pet Demo REST API Service",
  "url": "http://localhost:5000/pets",
  "version": "1.0"
}
```

LIST ALL PETS

- This code is called on the URL GET /pets

```
@app.route('/pets', methods=['GET'])
def list_pets():
    results = []
    category = request.args.get('category')
    if category:
        app.logger.info('Getting Pets for category: {}'.format(category))
        results = Pet.find_by_category(category)
    else:
        app.logger.info('Getting all Pets')
        results = Pet.all()

    return jsonify([pet.serialize() for pet in results]), status.HTTP_200_OK
```

LIST ALL PETS

- This code is called on the URL GET /pets

```
@app.route('/pets', methods=['GET'])
def list_pets():
    results = []
    category = request.args.get('category')
    if category:
        app.logger.info('Getting Pets for category: {}'.format(categor
        results = Pet.find_by_category(category)
    else:
        app.logger.info('Getting all Pets')
        results = Pet.all()

    return jsonify([pet.serialize() for pet in results]), status.HTTP_200_OK
```

```
200
[
  {
    "id": 2,
    "category": "cat",
    "name": "kitty"
  },
  {
    "id": 1,
    "category": "dog",
    "name": "fido"
  }
]
```

RETRIEVE A PET

- This code is called on the URL GET /pets/<id>

```
@app.route('/pets/<int:pet_id>', methods=['GET'])
def get_pets(pet_id):
    app.logger.info('Getting Pet with id: {}'.format(pet_id))
    pet = Pet.find(pet_id)
    if not pet:
        raise NotFound('Pet with id: {} was not found'.format(pet_id))

    return jsonify(pet.serialize()), status.HTTP_200_OK
```

RETRIEVE A PET

- This code is called on the URL GET /pets/<id>

```
@app.route('/pets/<int:pet_id>', methods=['GET'])
def get_pets(pet_id):
    app.logger.info('Getting Pet with id: {}'.format(pet_id))
    pet = Pet.find(pet_id)
    if not pet:
        raise NotFound('Pet with id: {} was not found'.format(pet_id))

    return jsonify(pet.serialize()), status.HTTP_200_OK
```

```
200
{
  "id": 2,
  "category": "cat",
  "name": "kitty"
}
```

Create a Pet

- This code is called on the URL POST /pets

```
@app.route('/pets', methods=['POST'])
def create_pets():
    app.logger.info('Create Pet requested')
    pet = Pet()
    pet.deserialize(request.get_json())
    pet.save()
    app.logger.info('Created Pet with id: {}'.format(pet.id))
    return make_response(jsonify(pet.serialize()),
                         status.HTTP_201_CREATED,
                         {'Location': url_for('get_pets', pet_id=pet.id, _external=True)})
```

Create a Pet

- This code is called on the URL POST /pets

```
@app.route('/pets', methods=['POST'])
def create_pets():
    app.logger.info('Create Pet requested')
    pet = Pet()
    pet.deserialize(request.get_json())
    pet.save()
    app.logger.info('Created Pet with id: {}'.format(pet.id))
    return make_response(jsonify(pet.serialize()),
                         status.HTTP_201_CREATED,
                         {'Location': url_for('get_pets', pet_id=pet.id, _external=True)})
```

```
201
{
  "id": 3,
  "category": "lion",
  "name": "leo"
}
```

Update an Existing pet

- This code is called on the URL PUT /pets

```
@app.route('/pets/<int:pet_id>', methods=['PUT'])
def update_pets(pet_id):
    app.logger.info('Updating with id: {}'.format(pet_id))
    pet = Pet.find(pet_id)
    if not pet:
        raise NotFound('Pet with id: {} was not found'.format(pet_id))

    # process the update request
    pet.deserialize(request.get_json())
    pet.id = pet_id # make id matches request
    pet.save()
    app.logger.info('Pet with id {} has been updated'.format(pet_id))
    return jsonify(pet.serialize()), status.HTTP_200_OK
```

Update an Existing pet

- This code is called on the URL PUT /pets

```
@app.route('/pets/<int:pet_id>', methods=['PUT'])
def update_pets(pet_id):
    app.logger.info('Updating with id: {}'.format(pet_id))
    pet = Pet.find(pet_id)
    if not pet:
        raise NotFound('Pet with id: {} was not found'.format(pet_id))

    # process the update request
    pet.deserialize(request.get_json())
    pet.id = pet_id # make id matches request
    pet.save()
    app.logger.info('Pet with id {} has been updated'.format(pet_id))
    return jsonify(pet.serialize()), status.HTTP_200_OK
```

```
200
{
  "id": 2,
  "category": "tabby",
  "name": "kitty"
}
```

Delete a Pet

- This code is called on the URL POST /pets

```
@app.route('/pets/<int:pet_id>', methods=['DELETE'])
def delete_pets(pet_id):
    app.logger.info('Request to delete Pet with id: {}'.format(pet_id))
    pet = Pet.find(pet_id)
    if pet:
        pet.delete()
    return make_response('', status.HTTP_204_NO_CONTENT)
```

Delete a Pet

- This code is called on the URL POST /pets

```
@app.route('/pets/<int:pet_id>', methods=['DELETE'])
def delete_pets(pet_id):
    app.logger.info('Request to delete Pet with id: {}'.format(pet_id))
    pet = Pet.find(pet_id)
    if pet:
        pet.delete()
    return make_response('', status.HTTP_204_NO_CONTENT)
```

```
204
''
```

Delete a Pet

- This code is called on the URL POST /pets

```
@app.route('/pets/<int:pet_id>', methods=['DELETE'])
def delete_pets(pet_id):
    app.logger.info('Request to delete Pet with id: {}'.format(pet_id))
    pet = Pet.find(pet_id)
    if pet:
        pet.delete()
    return make_response('', status.HTTP_204_NO_CONTENT)
```

```
204
''
```

What should we do if there is no Pet with that ID?

Pet Model

- This is a summary of the methods in the Pet Model

```
class Pet(object):
    def __init__(self):
    def __repr__(self):
    def save(self):
    def delete(self):
    def serialize(self):
    def deserialize(self, data):

    @classmethod
    def all():
    def find(id):
    def find_by_category(category):
```

Pet Model

- Data Attributes

```
class Pet(db.Model):
    """ Represents a single pet """
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(63))
    category = db.Column(db.String(63))

    def __repr__(self):
        return '<Pet %r>' % (self.name)
```

Pet Model

- Persistence Methods

```
    def save(self):
        """ Saves an existing Pet in the database """
        # if the id is None it hasn't been added to the database
        if not self.id:
            db.session.add(self)
        db.session.commit()

     def delete(self):
        """ Deletes a Pet from the database """
        db.session.delete(self)
        db.session.commit()
```

Pet Model

- Conversions for transmission

```
    def serialize(self):
        return { "id": self.id, "name": self.name, "category": self.category }

    def deserialize(self, data):
        try:
            self.name = data['name']
            self.category = data['category']
        except KeyError as e:
            raise DataValidationError('Invalid pet: missing ' + e.args[0])
        except TypeError as e:
            raise DataValidationError('Invalid pet: body of request contained bad or no data')
        return self
```

Pet Model

- Query Methods

```
    @classmethod
     def all(cls):
         """ Returns all of the Pets in the database """
         return cls.query.all()

     @classmethod
     def find(cls, pet_id):
         """ Finds a Pet by it's ID """
         return cls.query.get(pet_id)

     @classmethod
     def find_by_category(cls, category):
         """ Returns all of the Pets in a category """
         return cls.query.filter(cls.category == category)
```

What About testing?

- It is critical to have test cases for all of your REST APIs

```
class TestPetService(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        """ Run once before all tests """
       app.config['SQLALCHEMY_DATABASE_URI'] = DATABASE_URI

    def setUp(self):
        """ Runs before each test """
        db.drop_all()    # clean up the last tests
        db.create_all()  # create new tables
        Pet(name='fido', category='dog').save()
        Pet(name='kitty', category='cat').save()
        self.app = app.test_client()

    def test_index(self):
        """ Test the Home Page """
        resp = self.app.get('/')
        self.assertEqual(resp.status_code, status.HTTP_200_OK)
        data = resp.get_json()
        self.assertEqual(data['name'], 'Pet Demo REST API Service')
```

Summary

- You should now have a good overview what a REST API is
- You now know the guidelines for creating a good RESTful API
- You should also be able to create a REST API for any Resource using Python and Flask

Additional Reading

- **RESTful Web Services Cookbook** by Subbu Allamaraju, Publisher: O'Reilly Media, Inc.
- **REST API Design Rulebook** by Mark Masse, Publisher: O'Reilly Media, Inc.
- **REST in Practice**, by Savas Parastatidis, Jim Webber, Ian Robinson,Publisher: O'Reilly Media, Inc.
- **RESTful Web Services**, by Sam Ruby, Leonard Richardson, Publisher: O'Reilly Media, Inc.
- **Microservice Architecture**, by Martin Fowler (https://martinfowler.com/articles/microservices.html)