RESTful Web API

With Python, Flask and Mongo

Nicola Iarocci

Good Morning.

@nicolaiarocci

Full Disclosure

I'm a .NET guy

20 years in the Microsoft ecosystem
Scary, Yes.

"Life begins at the end of your comfort zone."

- Neale Walsh

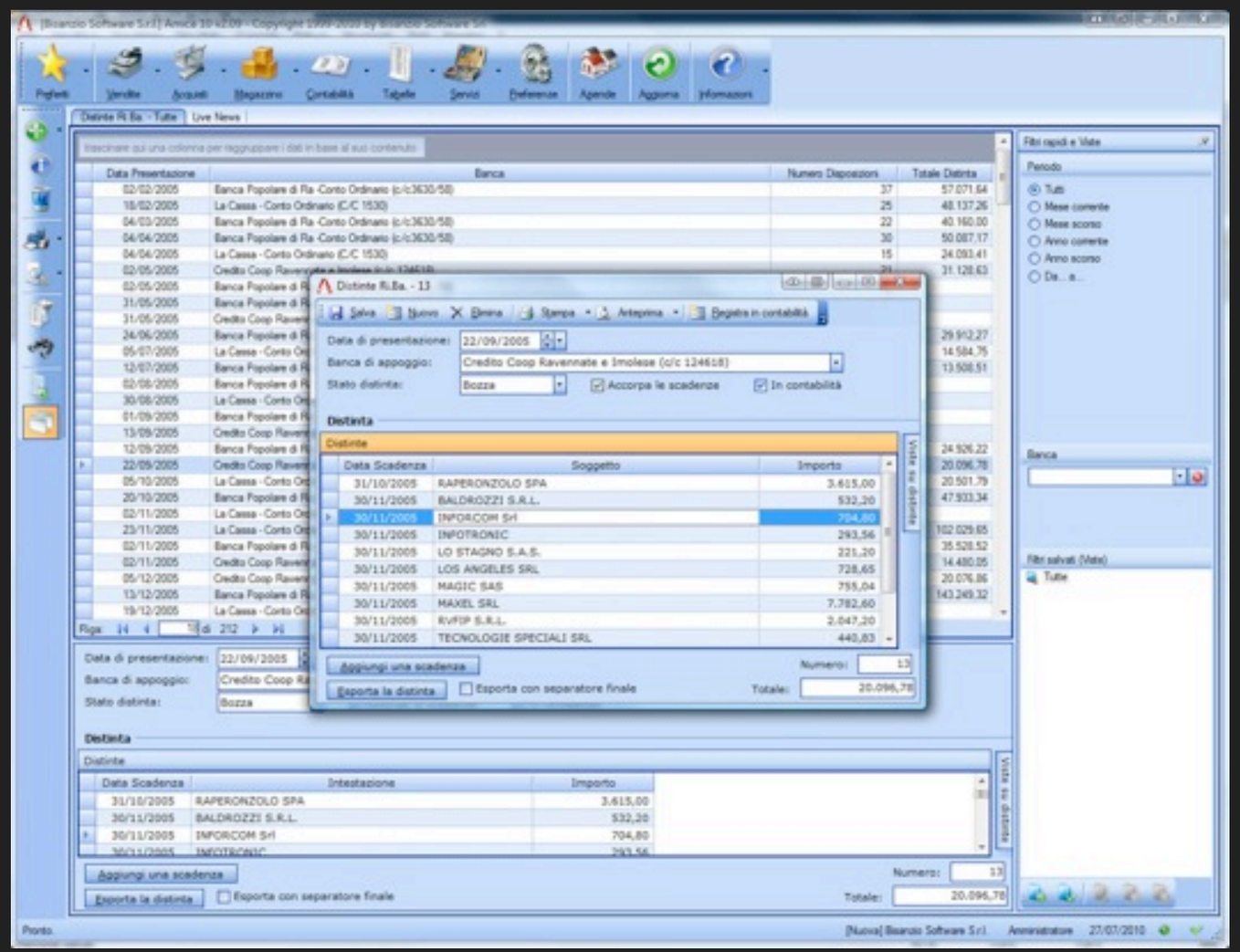
Still with me?

Great.

gestionaleamica.com

invoicing & accounting

Your Typical Old School Desktop App...



... now going web & mobile

Enter Python

Flask and Mongo

REST

So What Is REST All About?

REST is

not a standard

REST is not a protocol

REST is an architectural style for networked applications

REST
defines a set of simple principles

loosely followed by most API implementations

#1

resource

the source of a specific information

A web page is not a resource

rather, the representation of a resource

#2
global
permanent identifier

every resource is uniquely identified
(think a HTTP URI)

#3
standard interface

used to exchange representations of resources
(think the HTTP protocol)

#4

set of constraints

separation of concerns, stateless, cacheability,
layered system, uniform interface...

we'll get to
these later

The World Wide Web is built on REST

and it is meant to be consumed by humans

RESTful Web APIs are built on REST

and are meant to be consumed by machines

Beginners Reading

How I Explained REST to My Wife

by Ryan Tomayko

http://tomayko.com/writings/rest-to-my-wife

The Real Stuff

Representational State Transfer (REST)

by Roy Thomas Fielding

http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

RESTful Web API

Nuts & Bolts

The Tools

or why I picked Flask and Mongo

Flask

web development, one drop at a time

Simple & Elegant

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

@app.route("/")
def hello():
    return "Hello World!"

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

RESTful request dispacthing

```
@app.route('/user/<username>')
def show_user_profile(username):
    return 'User %s' % username
```

```
@app.route('/post/<int:post_id>')
def show_post(post_id):
    return 'Post %d' % post_id
```

Built-in development server & debugger

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

@app.route("/")
def hello():
    return "Hello World!"

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

Explicit & passable application objects

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

@app.route("/")
def hello():
    return "Hello World!"

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

100% WSGI Compliant

e.g., response objects are WSGI applications themselves

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

@app.route("/")
def hello():
    return "Hello World!"

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

Minimal Footprint

Only 800 lines of source code

Heavily Tested

1500 lines of tests

Unittesting Support

one day I will make good use of it

Bring Your Own Batteries

we aim for flexibility

No built-in ORM

we want to be as close to the bare metal as possible

No form validation

we don’t need no freaking form validation

No data validation

Python offers great tools to manipulate JSON,
we can tinker something ourselves

Layered API

built on Werkzeug, Jinja2, WSGI

Built by the Pros

The Pocoo Team did Werkzeug, Jinja2, Sphinx, Pygments, and much more

Excellent Documentation

Over 200 pages, lots of examples and howtos

Active Community

Widely adopted, extensions for everything

“Flask is a sharp tool for building sharp services”

Kenneth Reitz,
DjangoCon 2012

MongoDB

scalable, high-performance, open source NoSQL database

Similarity with RDBMS

made NoSQL easy to grasp (even for a **dumbhead** like me)

Terminology

RDBMS	Mongo
Database	Database
Table	Collection
Rows(s)	JSON Document
Index	Index
Join	Embedding & Linking

JSON-style data store

true selling point for me

JSON & RESTful API

GET

Client

Mongo

maybe we can push directly to client?

JSON & RESTful API

GET

Client

API

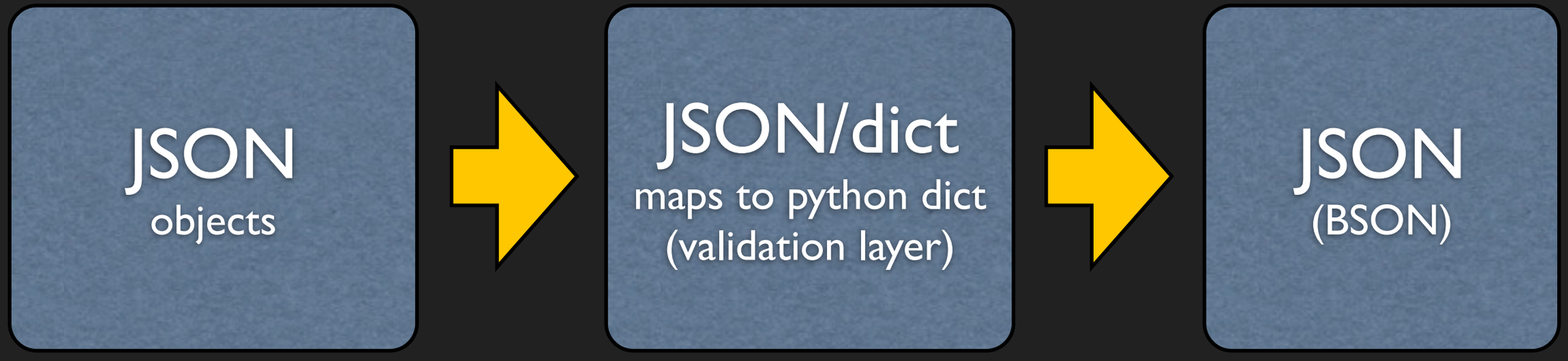
Mongo

almost.

JSON & RESTful API

POST

Client | API | Mongo



also works when posting (adding) items to the database

What about Queries?

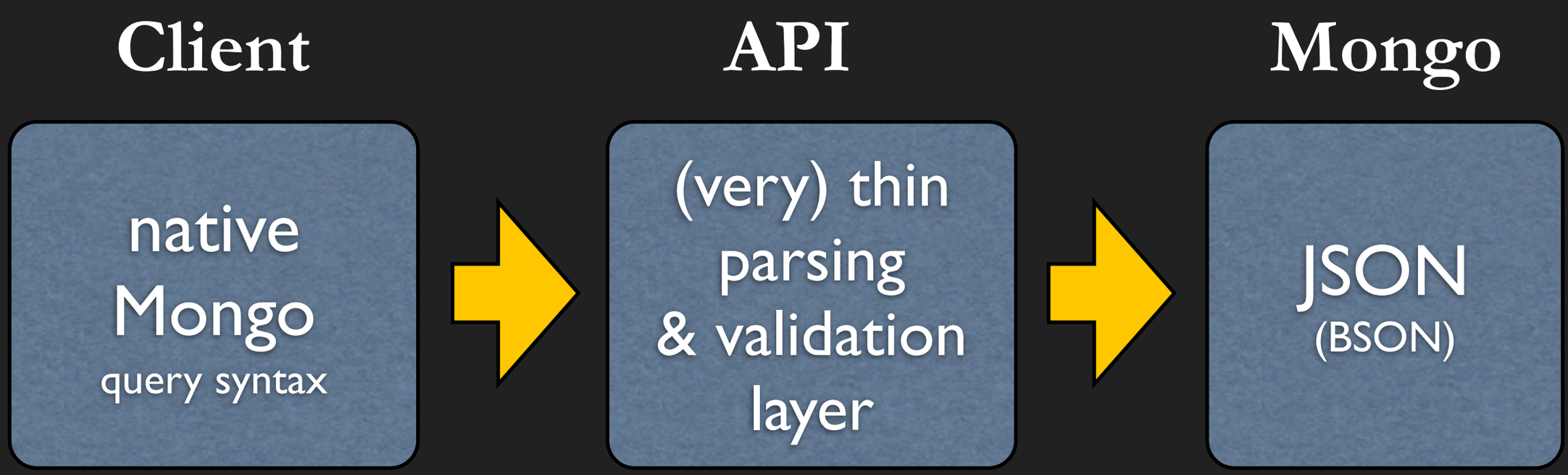
Queries in MongoDB are represented as JSON-style objects

```
// select * from things where x=3 and y="foo"
db.things.find({x: 3, y: "foo"});
```

JSON & RESTful API

FILTERING & SORTING

```
?where={x: 3, y: "foo”}
```



JSON
all along the pipeline

mapping to and from the database feels more natural

Schema-less

dynamic objects allow for a painless evolution of our schema
(because yes, a schema exists at any point in time)

ORM

Where we're going we don't need ORMs.

PyMongo

official Python driver
all we need to interact with the database

Also in MongoDB

- setup is a breeze
- lightweight
- fast inserts, updates and queries
- excellent documentation
- great support by 10gen
- great community

A Great Introduction To MongoDB

The Little MongoDB Book

by Karl Seguin

http://openmymind.net/2011/3/28/The-Little-MongoDB-Book/

Shameless Plug

Il Piccolo Libro di MongoDB

by Karl Seguin, traduzione di

Nicola Iarocci

http://nicolaiarocci.com/il-piccolo-libro-di-mongodb-edizione-italiana/

MongoDB Interactive Tutorial

http://tutorial.mongly.com/tutorial/index

RESTful Web APIs are really just collection of resources

accesible through to a uniform interface

#1
each resource is identified by a persistent identifier

We need to properly implement Request Dispatching

Collections

API's entry point + plural nouns

http://api.example.com/v1/contacts

Collections

Flask URL dispatcher allows for variables

```
@app.route('/<collection>')
def collection(collection):
    if collection in DOMAIN.keys():
        (...)
abort(404)
```

api.example.com/**contacts**
api.example.com/**invoices**
etc.

Collections

Flask URL dispatcher allows for variables

```
@app.route('/<collection>')
def collection(collection):
    if collection in DOMAIN.keys():
        (...)
    abort(404)
```

validation
dictonary

Collections

Flask URL dispatcher allows for variables

```
@app.route('/<collection>')
def collection(collection):
    if collection in DOMAIN.keys():
        (...)
    abort(404)
```

we don't know
this collection,
return a **404**

RegEx

by design, collection URLs are plural nouns

```
@app.route('/<regex("[\w]*[Ss]"):collection>')
def collection(collection):
    if collection in DOMAIN.keys():
        (...)
    abort(404)
```

regular expressions can be used to better narrow a variable part URL.
However...

RegEx

We need to build our own Custom Converter

```
class RegexConverter(BaseConverter):
    def __init__(self, url_map, *items):
        super(RegexConverter, self).__init__(url_map)
        self.regex = items[0]

app.url_map.converters['regex'] = RegexConverter
```

subclass **BaseConverter** and
pass the new converter to
the url_map

Document

Documents are identified by ObjectID

http://api.example.com/v1/contacts/4f46445fc88e201858000000

And eventually by an alternative lookup value

http://api.example.com/v1/contacts/CUST12345

Document

```
@app.route('/<regex("[\w]*[Ss]"):collection>/<lookup>')
@app.route('/<regex("[\w]*[Ss]"):collection>'
           '/<regex("[a-f0-9]{24}"):object_id>')
def document(collection, lookup=None, object_id=None):
    (...)
```

URL dispatcher handles **multiple variables**

http://api.example.com/v1/contacts/CUST12345

Document

```
@app.route('/<regex("[\w]*[Ss]"):collection>/<lookup>')
@app.route('/<regex("[\w]*[Ss]"):collection>'
           '/<regex("[a-f0-9]{24}"):object_id>')
def document(collection, lookup=None, object_id=None):
    (...)
```

and of course it also handles **multiple RegEx** variables

http://api.example.com/v1/contacts/4f46445fc88e201858000000

Document

```
@app.route('/<regex("[\w]*[Ss]"):collection>/<lookup>')
@app.route('/<regex("[\w]*[Ss]"):collection>'
           '/<regex("[a-f0-9]{24}"):object_id>')
def document(collection, lookup=None, object_id=None):
    (...)
```

Different URLs can be dispatched to the same function just by piling up **@app.route** decorators.

#2
representation of resources via media types

JSON, XML or any other valid internet media type

depends on the request and not the identifier

Accepted Media Types

mapping supported media types to corresponding renderer functions

```
mime_types = {'json_renderer': ('application/json',),
              'xml_renderer': ('application/xml', 'text/xml',
                               'application/x-xml',)}
```

JSON rendering function

Accepted Media Types

mapping supported media types to corresponding renderer functions

```
mime_types = {'json_renderer': ('application/json',),
              'xml_renderer': ('application/xml', 'text/xml',
                               'application/x-xml',)}
```

corresponding JSON
internet **media type**

Accepted Media Types

mapping supported media types to corresponding renderer functions

```
mime_types = {'json_renderer': ('application/json',),
              'xml_renderer': ('application/xml', 'text/xml',
                               'application/x-xml',)}
```

XML rendering function

Accepted Media Types

mapping supported media types to corresponding renderer functions

```
mime_types = {'json_renderer': ('application/json',),
              'xml_renderer': ('application/xml', 'text/xml',
                               'application/x-xml',)}
```

corresponding XML
internet **media types**

JSON Render

datetimes and ObjectIDs call for further tinkering

```
class APIEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, datetime.datetime):
            return date_to_str(obj)
        elif isinstance(obj, ObjectId):
            return str(obj)
        return json.JSONEncoder.default(self, obj)


def json_renderer(**data):
    return json.dumps(data, cls=APIEncoder)
```

renderer function mapped to
the **appication/json**
media type

JSON Render

datetimes and ObjectIDs call for further tinkering

```
class APIEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, datetime.datetime):
            return date_to_str(obj)
        elif isinstance(obj, ObjectId):
            return str(obj)
        return json.JSONEncoder.default(self, obj)


def json_renderer(**data):
    return json.dumps(data, cls=APIEncoder)
```

standard json encoding is not enough, we need a specialized **JSONEncoder**

JSON Render

datetimes and ObjectIDs call for further tinkering

```
class APIEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, datetime.datetime):
            return date_to_str(obj)
        elif isinstance(obj, ObjectId):
            return str(obj)
        return json.JSONEncoder.default(self, obj)


def json_renderer(**data):
    return json.dumps(data, cls=APIEncoder)
```

Python datetimes are encoded as **RFC 1123** strings: “Wed, 06 Jun 2012 14:19:53 UTC”

JSON Render

datetimes and ObjectIDs call for further tinkering

```
class APIEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, datetime.datetime):
            return date_to_str(obj)
        elif isinstance(obj, ObjectId):
            return str(obj)
        return json.JSONEncoder.default(self, obj)


def json_renderer(**data):
    return json.dumps(data, cls=APIEncoder)
```

Mongo **ObjectId** data types are encoded as strings: “4f46445fc88e201858000000”

JSON Render

datetimes and ObjectIDs call for further tinkering

```
class APIEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, datetime.datetime):
            return date_to_str(obj)
        elif isinstance(obj, ObjectId):
            return str(obj)
        return json.JSONEncoder.default(self, obj)


def json_renderer(**data):
    return json.dumps(data, cls=APIEncoder)
```

we let **json/simplejson**
handle the other data types

Rendering

Render to JSON or XML and get WSGI response object

```
def prep_response(dct, status=200):
    mime, render = get_best_mime()
    rendered = globals()[render](**dct)
    resp = make_response(rendered, status)
    resp.mimetype = mime
    return resp
```

best match between request **Accept** header and media types supported by the service

Rendering

Render to JSON or XML and get WSGI response object

```
def prep_response(dct, status=200):
    mime, render = get_best_mime()
    rendered = globals()[render](**dct)
    resp = make_response(rendered, status)
    resp.mimetype = mime
    return resp
```

call the appropriate render function and retrieve the encoded JSON or XML

Rendering

Render to JSON or XML and get WSGI response object

```
def prep_response(dct, status=200):
    mime, render = get_best_mime()
    rendered = globals()[render](**dct)
    resp = make_response(rendered, status)
    resp.mimetype = mime
    return resp
```

flask's **make_response()** returns a WSGI response object wich we can use to attach headers

Rendering

Render to JSON or XML and get WSGI response object

```
def prep_response(dct, status=200):
    mime, render = get_best_mime()
    rendered = globals()[render](**dct)
    resp = make_response(rendered, status)
    resp.mimetype = mime
    return resp
```

and finally, we set the appropriate **mime type** in the response header

Flask-MimeRender

“Python module for RESTful resource representation using MIME Media-Types and the Flask Microframework”

```
pip install flask-mimerender
```

Flask-MimeRender

Render Functions

```
render_json = jsonify
render_xml = lambda message: '<message>%s</message>' % message
render_txt = lambda message: message
render_html = lambda message: '<html><body>%s</body></html>' % \
    message
```

Flask-MimeRender

then you just decorate your end-point function

```
@app.route('/')
@mimerender(
        default = 'html',
        html = render_html,
        xml  = render_xml,
        json = render_json,
        txt  = render_txt
)
def index():
    if request.method == 'GET':
        return {'message': 'Hello, World!'}
```

Flask-MimeRender

Requests

```
$ curl -H "Accept: application/html" example.com/
<html><body>Hello, World!</body></html>
```

```
$ curl -H "Accept: application/xml" example.com/
<message>Hello, World!</message>
```

```
$ curl -H "Accept: application/json" example.com/
{'message':'Hello, World!'}
```

```
$ curl -H "Accept: text/plain" example.com/
Hello, World!
```

#3
resource manipulation through HTTP verbs

"GET, POST, PUT, DELETE and all that mess"

HTTP Methods

Verbs are handled along with URL routing

```
@app.route('/<collection>', methods=['GET', 'POST'])
def collection(collection):
    if collection in DOMAIN.keys():
        if request.method == 'GET':
            return get_collection(collection)
        elif request.method == 'POST':
            return post(collection)
    abort(404)
```

accepted HTTP verbs
a PUT will throw a
405 Command Not Allowed

HTTP Methods

Verbs are handled along with URL routing

```
@app.route('/<collection>', methods=['GET', 'POST'])
def collection(collection):
    if collection in DOMAIN.keys():
        if request.method == 'GET':
            return get_collection(collection)
        elif request.method == 'POST':
            return post(collection)
    abort(404)
```

the global **request object** provides access to clients' request headers

HTTP Methods

Verbs are handled along with URL routing

```
@app.route('/<collection>', methods=['GET', 'POST'])
def collection(collection):
    if collection in DOMAIN.keys():
        if request.method == 'GET':
            return get_collection(collection)
        elif request.method == 'POST':
            return post(collection)
    abort(404)
```

we respond to a **GET** request
for a ‘collection’ resource

HTTP Methods

Verbs are handled along with URL routing

```
@app.route('/<collection>', methods=['GET', 'POST'])
def collection(collection):
    if collection in DOMAIN.keys():
        if request.method == 'GET':
            return get_collection(collection)
        elif request.method == 'POST':
            return post(collection)
    abort(404)
```

and here we respond to a **POST** request. Handling HTTP methods is easy!

CRUD via REST

Acttion	HTTP Verb	Context
Get	GET	Collection/ Document
Create	POST	Collection
Update	PATCH*	Document
Delete	DELETE	Document

* WTF?

GET

Retrieve Multiple Documents (accepting Queries)

http://api.example.com/v1/contacts?where={“age”: {“$gt”: 20}}

Collection GET

http://api.example.com/v1/contacts?where={“age”: {“$gt”: 20}}

```
def get_collection(collection):
    where = request.args.get('where')
    if where:
        args['spec'] = json.loads(where, object_hook=datetime_parser)
    (...)
    response = {}
    documents = []

    cursor = db(collection).find(**args)
    for document in cursor:
        documents.append(document)

    response[collection] = documents
    return prep_response(response)
```

request.args returns the original URI’s query definition, in our example:

where = {“age”: {“$gt”: 20}}

Collection GET

http://api.example.com/v1/contacts?where={“age”: {“$gt”: 20}}

```
def get_collection(collection):
    where = request.args.get('where')
    if where:
        args['spec'] = json.loads(where, object_hook=datetime_parser)
    (...)
    response = {}
    documents = []

    cursor = db(collection).find(**args)
    for document in cursor:
        documents.append(document)

    response[collection] = documents
    return prep_response(response)
```

as the query already comes in as a Mongo expression:

{“age”: {“$gt”: 20}}

we simply convert it to JSON.

Collection GET

http://api.example.com/v1/contacts?where={“age”: {“$gt”: 20}}

```
def get_collection(collection):
    where = request.args.get('where')
    if where:
        args['spec'] = json.loads(where, object_hook=datetime_parser)
    (...)
    response = {}
    documents = []

    cursor = db(collection).find(**args)
    for document in cursor:
        documents.append(document)

    response[collection] = documents
    return prep_response(response)
```

String-to-datetime conversion is obtained via the **object_hook** mechanism

Collection GET

http://api.example.com/v1/contacts?where={“age”: {“$gt”: 20}}

```
def get_collection(collection):
    where = request.args.get('where')
    if where:
        args['spec'] = json.loads(where, object_hook=datetime_parser)
    (...)
    response = {}
    documents = []

    cursor = db(collection).find(**args)
    for document in cursor:
        documents.append(document)

    response[collection] = documents
    return prep_response(response)
```

find() accepts a python dict as query expression, and returns a cursor we can iterate

Collection GET

http://api.example.com/v1/contacts?where={“age”: {“$gt”: 20}}

```
def get_collection(collection):
    where = request.args.get('where')
    if where:
        args['spec'] = json.loads(where, object_hook=datetime_parser)
    (...)
    response = {}
    documents = []

    cursor = db(collection).find(**args)
    for document in cursor:
        documents.append(document)

    response[collection] = documents
    return prep_response(response)
```

finally, we encode the response dict with the requested MIME media-type

Interlude

On encoding JSON dates

On encoding JSON dates

- We don't want to force metadata into JSON representation:
  ("updated": "$date: Thu 1, ..")
- Likewise, epochs are not an option
- We are aiming for a broad solution not relying on the knoweldge of the current domain

the guy behind Redis

Because, you know

Parsing JSON dates

this is what I came out with

```
>>> source = '{"updated": "Thu, 1 Mar 2012 10:00:49 UTC"}'
>>> dct = json.loads(source, object_hook=datetime_parser)
>>> dct
{u'updated': datetime.datetime(2012, 3, 1, 10, 0, 49)}
```

```
def datetime_parser(dct):
    for k, v in dct.items():
        if isinstance(v, basestring) and re.search("\ UTC", v):
            try:
                dct[k] = datetime.datetime.strptime(v, DATE_FORMAT)
            except:
                pass
    return dct
```

object_hook is usually used to deserialize JSON to classes (rings a ORM bell?)

Parsing JSON dates

this is what I came out with

```
>>> source = '{"updated": "Thu, 1 Mar 2012 10:00:49 UTC"}'
>>> dct = json.loads(source, object_hook=datetime_parser)
>>> dct
{u'updated': datetime.datetime(2012, 3, 1, 10, 0, 49)}
```

```
def datetime_parser(dct):
    for k, v in dct.items():
        if isinstance(v, basestring) and re.search("\ UTC", v):
            try:
                dct[k] = datetime.datetime.strptime(v, DATE_FORMAT)
            except:
                pass
    return dct
```

the resulting dct now has **datetime values** instead of string representations of dates

Parsing JSON dates

this is what I came out with

```
>>> source = '{"updated": "Thu, 1 Mar 2012 10:00:49 UTC"}'
>>> dct = json.loads(source, object_hook=datetime_parser)
>>> dct
{u'updated': datetime.datetime(2012, 3, 1, 10, 0, 49)}
```

```
def datetime_parser(dct):
    for k, v in dct.items():
        if isinstance(v, basestring) and re.search("\ UTC", v):
            try:
                dct[k] = datetime.datetime.strptime(v, DATE_FORMAT)
            except:
                pass
    return dct
```

the function receives a dict representing the **decoded JSON**

Parsing JSON dates

this is what I came out with

```
>>> source = '{"updated": "Thu, 1 Mar 2012 10:00:49 UTC"}'
>>> dct = json.loads(source, object_hook=datetime_parser)
>>> dct
{u'updated': datetime.datetime(2012, 3, 1, 10, 0, 49)}
```

```
def datetime_parser(dct):
    for k, v in dct.items():
        if isinstance(v, basestring) and re.search("\ UTC", v):
            try:
                dct[k] = datetime.datetime.strptime(v, DATE_FORMAT)
            except:
                pass
    return dct
```

strings matching the **RegEx** (which probably should be better defined)...

Parsing JSON dates

this is what I came out with

```
>>> source = '{"updated": "Thu, 1 Mar 2012 10:00:49 UTC"}'
>>> dct = json.loads(source, object_hook=datetime_parser)
>>> dct
{u'updated': datetime.datetime(2012, 3, 1, 10, 0, 49)}
```

```
def datetime_parser(dct):
    for k, v in dct.items():
        if isinstance(v, basestring) and re.search("\ UTC", v):
            try:
                dct[k] = datetime.datetime.strptime(v, DATE_FORMAT)
            except:
                pass
    return dct
```

...are converted to **datetime** values

Parsing JSON dates

this is what I came out with

```
>>> source = '{"updated": "Thu, 1 Mar 2012 10:00:49 UTC"}'
>>> dct = json.loads(source, object_hook=datetime_parser)
>>> dct
{u'updated': datetime.datetime(2012, 3, 1, 10, 0, 49)}
```

```
def datetime_parser(dct):
    for k, v in dct.items():
        if isinstance(v, basestring) and re.search("\ UTC", v):
            try:
                dct[k] = datetime.datetime.strptime(v, DATE_FORMAT)
            except:
                pass
    return dct
```

if conversion fails we assume that we are dealing a normal, legit string

PATCH

Editing a Resource

Why not PUT?

- PUT means resource creation or replacement at a given URL
- PUT does not allow for partial updates of a resource
- 99% of the time we are updating just one or two fields
- We don't want to send complete representations of the document we are updating
- Mongo allows for atomic updates and we want to take advantage of that

‘atomic’ PUT updates are ok when each field is itself a resource

http://api.example.com/v1/contacts/<id>/address

Enter PATCH

"This specification defines the new method, PATCH, which is used to apply partial modifications to a resource."

RFC5789

PATCH

- send a “patch document” with just the changes to be applied to the document
- saves bandwidth and reduces traffic
- it’s been around since 1995
- it is a RFC Proposed Standard
- Widely adopted (will replace PUT in Rails 4.0)
- clients not supporting it can fallback to POST with ‘X-HTTP-Method-Override: PATCH’ header tag

PATCHing

```
def patch_document(collection, original):
    docs = parse_request(request.form)
    if len(docs) > 1:
        abort(400)

    key, value = docs.popitem()
    response_item = {}
    object_id = original[ID_FIELD]

    # Validation
    validate(value, collection, object_id)
    response_item['validation'] = value['validation']

    if value['validation']['response'] != VALIDATION_ERROR:
        # Perform the update
        updates = {"$set": value['doc']}
        db(collection).update({"_id": ObjectId(object_id)}, updates)
        response_item[ID_FIELD] = object_id
    return prep_response(response_item)
```

request.form returns a dict with request form data.

PATCHing

```
def patch_document(collection, original):
    docs = parse_request(request.form)
    if len(docs) > 1:
        abort(400)

    key, value = docs.popitem()
    response_item = {}
    object_id = original[ID_FIELD]

    # Validation
    validate(value, collection, object_id)
    response_item['validation'] = value['validation']

    if value['validation']['response'] != VALIDATION_ERROR:
        # Perform the update
        updates = {"$set": value['doc']}
        db(collection).update({"_id": ObjectId(object_id)}, updates)
        response_item[ID_FIELD] = object_id
    return prep_response(response_item)
```

we aren't going to accept more than one document here

PATCHing

```
def patch_document(collection, original):
    docs = parse_request(request.form)
    if len(docs) > 1:
        abort(400)

    key, value = docs.popitem()
    response_item = {}
    object_id = original[ID_FIELD]

    # Validation
    validate(value, collection, object_id)
    response_item['validation'] = value['validation']

    if value['validation']['response'] != VALIDATION_ERROR:
        # Perform the update
        updates = {"$set": value['doc']}
        db(collection).update({"_Id": ObjectId(object_id)}, updates)
        response_item[ID_FIELD] = object_id
    return prep_response(response_item)
```

retrieve the original document ID, will be used by the update command

PATCHing

```
def patch_document(collection, original):
    docs = parse_request(request.form)
    if len(docs) > 1:
        abort(400)

    key, value = docs.popitem()
    response_item = {}
    object_id = original[ID_FIELD]

    # Validation
    validate(value, collection, object_id)
    response_item['validation'] = value['validation']

    if value['validation']['response'] != VALIDATION_ERROR:
        # Perform the update
        updates = {"$set": value['doc']}
        db(collection).update({"_Id": ObjectId(object_id)}, updates)
        response_item[ID_FIELD] = object_id
    return prep_response(response_item)
```

validate the updates

PATCHing

```
def patch_document(collection, original):
    docs = parse_request(request.form)
    if len(docs) > 1:
        abort(400)

    key, value = docs.popitem()
    response_item = {}
    object_id = original[ID_FIELD]

    # Validation
    validate(value, collection, object_id)
    response_item['validation'] = value['validation']

    if value['validation']['response'] != VALIDATION_ERROR:
        # Perform the update
        updates = {"$set": value['doc']}
        db(collection).update({"_id": ObjectId(object_id)}, updates)
        response_item[ID_FIELD] = object_id
    return prep_response(response_item)
```

add validation results to the response dictionary

PATCHing

```
def patch_document(collection, original):
    docs = parse_request(request.form)
    if len(docs) > 1:
        abort(400)

    key, value = docs.popitem()
    response_item = {}
    object_id = original[ID_FIELD]

    # Validation
    validate(value, collection, object_id)
    response_item['validation'] = value['validation']

    if value['validation']['response'] != VALIDATION_ERROR:
        # Perform the update
        updates = {"$set": value['doc']}
        db(collection).update({"_Id": ObjectId(object_id)}, updates)
        response_item[ID_FIELD] = object_id
    return prep_response(response_item)
```

$set accepts a dict
with the updates for the db
eg: {“active”: False}.

PATCHing

```
def patch_document(collection, original):
    docs = parse_request(request.form)
    if len(docs) > 1:
        abort(400)

    key, value = docs.popitem()
    response_item = {}
    object_id = original[ID_FIELD]

    # Validation
    validate(value, collection, object_id)
    response_item['validation'] = value['validation']

    if value['validation']['response'] != VALIDATION_ERROR:
        # Perform the update
        updates = {"$set": value['doc']}
        db(collection).update({"_id": ObjectId(object_id)}, updates)
        response_item[ID_FIELD] = object_id
    return prep_response(response_item)
```

mongo **update()** method commits updates to the database. Updates are atomic.

PATCHing

```
def patch_document(collection, original):
    docs = parse_request(request.form)
    if len(docs) > 1:
        abort(400)

    key, value = docs.popitem()
    response_item = {}
    object_id = original[ID_FIELD]

    # Validation
    validate(value, collection, object_id)
    response_item['validation'] = value['validation']

    if value['validation']['response'] != VALIDATION_ERROR:
        # Perform the update
        updates = {"$set": value['doc']}
        db(collection).update({"_id": ObjectId(object_id)}, updates)
        response_item[ID_FIELD] = object_id
    return prep_response(response_item)
```

udpate() takes the unique Id of the document andthe update expression ($set)

PATCHing

```
def patch_document(collection, original):
    docs = parse_request(request.form)
    if len(docs) > 1:
        abort(400)

    key, value = docs.popitem()
    response_item = {}
    object_id = original[ID_FIELD]

    # Validation
    validate(value, collection, object_id)
    response_item['validation'] = value['validation']

    if value['validation']['response'] != VALIDATION_ERROR:
        # Perform the update
        updates = {"$set": value['doc']}
        db(collection).update({"_Id": ObjectId(object_id)}, updates)
        response_item[ID_FIELD] = object_id
    return prep_response(response_item)
```

as always, our response dictionary is returned with proper encding

POST

Creating Resources

POSTing

```
def post(collection):
    docs = parse_request(request.form)
    response = {}
    for key, item in docs.items():
        response_item = {}
        validate(item, collection)
        if item['validation']['response'] != VALIDATION_ERROR:
            document = item['doc']
            response_item[ID_FIELD] = db(collection).insert(document)
            response_item['link'] = get_document_link(collection,
                                                      response_item[ID_FIELD])
        response_item['validation'] = item['validation']
        response[key] = response_item
    return {'response': response}
```

we accept multiple documents
(remember, we are at
collection level here)

POSTing

```
def post(collection):
    docs = parse_request(request.form)
    response = {}
    for key, item in docs.items():
        response_item = {}
        validate(item, collection)
        if item['validation']['response'] != VALIDATION_ERROR:
            document = item['doc']
            response_item[ID_FIELD] = db(collection).insert(document)
            response_item['link'] = get_document_link(collection,
                                                      response_item[ID_FIELD])
        response_item['validation'] = item['validation']
        response[key] = response_item
    return {'response': response}
```

we loop through the documents to be inserted

POSTing

```
def post(collection):
    docs = parse_request(request.form)
    response = {}
    for key, item in docs.items():
        response_item = {}
        validate(item, collection)
        if item['validation']['response'] != VALIDATION_ERROR:
            document = item['doc']
            response_item[ID_FIELD] = db(collection).insert(document)
            response_item['link'] = get_document_link(collection,
                                                      response_item[ID_FIELD])
        response_item['validation'] = item['validation']
        response[key] = response_item
    return {'response': response}
```

perform validation on the document

POSTing

```
def post(collection):
    docs = parse_request(request.form)
    response = {}
    for key, item in docs.items():
        response_item = {}
        validate(item, collection)
        if item['validation']['response'] != VALIDATION_ERROR:
            document = item['doc']
            response_item[ID_FIELD] = db(collection).insert(document)
            response_item['link'] = get_document_link(collection,
                                                      response_item[ID_FIELD])
        response_item['validation'] = item['validation']
        response[key] = response_item
    return {'response': response}
```

push document and get its **ObjectId** back from Mongo. like other CRUD operations, inserting is trivial in mongo.

POSTing

```
def post(collection):
    docs = parse_request(request.form)
    response = {}
    for key, item in docs.items():
        response_item = {}
        validate(item, collection)
        if item['validation']['response'] != VALIDATION_ERROR:
            document = item['doc']
            response_item[ID_FIELD] = db(collection).insert(document)
            response_item['link'] = get_document_link(collection,
                                                      response_item[ID_FIELD])
        response_item['validation'] = item['validation']
        response[key] = response_item
    return {'response': response}
```

a **direct link** to the resource we just created is added to the response

POSTing

```
def post(collection):
    docs = parse_request(request.form)
    response = {}
    for key, item in docs.items():
        response_item = {}
        validate(item, collection)
        if item['validation']['response'] != VALIDATION_ERROR:
            document = item['doc']
            response_item[ID_FIELD] = db(collection).insert(document)
            response_item['link'] = get_document_link(collection,
                                                      response_item[ID_FIELD])
        response_item['validation'] = item['validation']
        response[key] = response_item
    return {'response': response}
```

validation result is always returned to the client, even if the doc has not been inserted

POSTing

```
def post(collection):
    docs = parse_request(request.form)
    response = {}
    for key, item in docs.items():
        response_item = {}
        validate(item, collection)
        if item['validation']['response'] != VALIDATION_ERROR:
            document = item['doc']
            response_item[ID_FIELD] = db(collection).insert(document)
            response_item['link'] = get_document_link(collection,
                                                      response_item[ID_FIELD])
        response_item['validation'] = item['validation']
        response[key] = response_item
    return {'response': response}
```

standard response enconding applied

Data Validation

We still need to validate incoming data

Data Validation

```
DOMAIN = {}
DOMAIN['contacts'] = {
                        'secondary_id': 'name',
                        'fields': {
                            'name': {
                                'data_type': 'string',
                                'required': True,
                                'unique': True,
                                'max_length': 120,
                                'min_length': 1
                    },
```

DOMAIN is a Python dict containing our validation rules and schema structure

Data Validation

```
DOMAIN = {}
DOMAIN['contacts'] = {
                        'secondary_id': 'name',
                        'fields': {
                            'name': {
                                'data_type': 'string',
                                'required': True,
                                'unique': True,
                                'max_length': 120,
                                'min_length': 1
                        },
```

every **resource** (collection) maintained by the API has a key in DOMAIN

Data Validation

```
DOMAIN = {}
DOMAIN['contacts'] = {
                        'secondary_id': 'name',
                        'fields': {
                            'name': {
                                'data_type': 'string',
                                'required': True,
                                'unique': True,
                                'max_length': 120,
                                'min_length': 1
                        },
```

if the resource allows for a **secondary lookup field**, we define it here

Data Validation

```
DOMAIN = {}
DOMAIN['contacts'] = {
                        'secondary_id': 'name',
                        'fields': {
                            'name': {
                                'data_type': 'string',
                                'required': True,
                                'unique': True,
                                'max_length': 120,
                                'min_length': 1
                    },
```

known fields go in the **fields** dict

Data Validation

```
DOMAIN = {}
DOMAIN['contacts'] = {
                        'secondary_id': 'name',
                        'fields': {
                            'name': {
                                'data_type': 'string',
                                'required': True,
                                'unique': True,
                                'max_length': 120,
                                'min_length': 1
                            },
```

validation rules for ‘name’ field. data_type is mostly needed to process datetimes and currency values

Data Validation

```
(...)
'iban': {
    'data_type': 'string',
    'custom_validation': {
        'module': 'customvalidation',
        'function': 'validate_iban'
    }
}
(...)
```

we can define **custom validation** functions when the need arises

Data Validation

```
(...)
'contact_type': {
                'data_type': 'array',
                'allowed_values': [
                    'client',
                    'agent',
                    'supplier',
                    'area manager',
                    'vector'
                ]
}
(...)
```

or we can define our own
custom data types...

Data Validation

```
(...)
'contact_type': {
                'data_type': 'array',
                'allowed_values': [
                    'client',
                    'agent',
                    'supplier',
                    'area manager',
                    'vector'
                ]
}
(...)
```

... like the **array**, which allows us to define a list of accepted values for the field

I will spare you the validation function

It’s pretty simple really

Hey but!
You're building your own ORM!

Just a thin validation layer on which I have total control

AKA
So What?

#4
Caching and concurrency control

resource representation describes how
when and if it can be used, discarded or re-fetched

Driving conditional requests

Servers use **`Last-Modified`** and **`ETag`** response headers to drive conditional requests

Last-Modified

Generally considered a **weak validator** since it has a one-second resolution

```
“Wed, 06 Jun 2012 14:19:53 UTC”
```

ETag

Entity Tag is a **strong validator** since its value can be changed every time the server modifies the representation

```
7a9f477cde424cf93a7db20b69e05f7b680b7f08
```

On ETags

- Clients should be able to use ETag to compare representations of a resouce
- An ETag is supposed to be like an object's hash code.
- Actually, some web frameworks and a lot of implementations do just that
- ETag computed on an entire representation of the resource may become a performance bottleneck

Last-Modified or ETag?

You can use either or both. Consider the types of client consuming your service. Hint: use both.

Validating cached representations

Clients use `If-Modified-Since` and `If-None-Match` in request headers for validating cached representations

If-Mod-Since & ETag

```
def get_document(collection, object_id=None, lookup=None):
    response = {}
    document = find_document(collection, object_id, lookup)
    if document:
        etag = get_etag(document)
        header_etag = request.headers.get('If-None-Match')
        if header_etag and header_etag == etag:
            return prep_response(dict(), status=304)

        if_modified_since = request.headers.get('If-Modified-Since')
        if if_modified_since:
            last_modified = document[LAST_UPDATED]
            if last_modified <= if_modified_since:
                return prep_response(dict(), status=304)

        response[collection.rstrip('s')] = document
        return prep_response(response, last_modified, etag)
    abort(404)
```

retrieve the document from the database

If-Mod-Since & ETag

```
def get_document(collection, object_id=None, lookup=None):
    response = {}
    document = find_document(collection, object_id, lookup)
    if document:
        etag = get_etag(document)
        header_etag = request.headers.get('If-None-Match')
        if header_etag and header_etag == etag:
            return prep_response(dict(), status=304)

        if_modified_since = request.headers.get('If-Modified-Since')
        if if_modified_since:
            last_modified = document[LAST_UPDATED]
            if last_modified <= if_modified_since:
                return prep_response(dict(), status=304)

        response[collection.rstrip('s')]
        return prep_response(response, ]
    abort(404)
```

compute ETag for the **current** representation. We test ETag first, as it is a **stronger validator**

If-Mod-Since & ETag

```
def get_document(collection, object_id=None, lookup=None):
    response = {}
    document = find_document(collection, object_id, lookup)
    if document:
        etag = get_etag(document)
        header_etag = request.headers.get('If-None-Match')
        if header_etag and header_etag == etag:
            return prep_response(dict(), status=304)

        if_modified_since = request.headers.get('If-Modified-Since')
        if if_modified_since:
            last_modified = document[LAST_UPDATED]
            if last_modified <= if_modified_since:
                return prep_response(dict(), status=304)

        response[collection.rstrip('s')] = document
        return prep_response(response, last_modified, etag)
    abort(404)
```

retrieve **If-None-Match** ETag
from request header

If-Mod-Since & ETag

```
def get_document(collection, object_id=None, lookup=None):
    response = {}
    document = find_document(collection, object_id, lookup)
    if document:
        etag = get_etag(document)
        header_etag = request.headers.get('If-None-Match')
        if header_etag and header_etag == etag:
            return prep_response(dict(), status=304)

        if_modified_since = request.headers.get('If-Modified-Since')
        if if_modified_since:
            last_modified = document[LAST_UPDATED]
            if last_modified <= if_modified_since:
                return prep_response(dict(), status=304)

        response[collection.rstrip('s')] = document
        return prep_response(response, ]
    abort(404)
```

if client and server representations **match**, return a 304 Not Modified

If-Mod-Since & ETag

```
def get_document(collection, object_id=None, lookup=None):
    response = {}
    document = find_document(collection, object_id, lookup)
    if document:
        etag = get_etag(document)
        header_etag = request.headers.get('If-None-Match')
        if header_etag and header_etag == etag:
            return prep_response(dict(), status=304)

        if_modified_since = request.headers.get('If-Modified-Since')
        if if_modified_since:
            last_modified = document[LAST_UPDATED]
            if last_modified <= if_modified_since:
                return prep_response(dict(), status=304)

        response[collection.rstrip('s')]
        return prep_response(response, 
    abort(404)
```

likewise, if the resource has not been modified since **If-Modifed-Since**, return 304 Not Modified

Concurrency control

Clients use **`If-Unmodified-Since`** and **`If-Match`** in request headers as preconditions for concurrency control

Concurrency control

Create/Update/Delete are controlled by ETag

```
def edit_document(collection, object_id, method):
    document = find_document(collection, object_id)
    if document:
        header_etag = request.headers.get('If-Match')
        if header_etag is None:
            return prep_response('If-Match missing from request header',
                                 status=403)
        if header_etag != get_etag(document[LAST_UPDATED]):
            # Precondition failed
            abort(412)
        else:
            if method in ('PATCH', 'POST'):
                return patch_document(collection, document)
            elif method == 'DELETE':
                return delete_document(collection, object_id)
    else:
        abort(404)
```

retrieve client's **If-Match** ETag from the request header

Concurrency control

Create/Update/Delete are controlled by ETag

```
def edit_document(collection, object_id, method):
    document = find_document(collection, object_id)
    if document:
        header_etag = request.headers.get('If-Match')
        if header_etag is None:
            return prep_response('If-Match missing from request header',
                                 status=403)
        if header_etag != get_etag(document[LAST_UPDATED]):
            # Precondition failed
            abort(412)
        else:
            if method in ('PATCH', 'POST'):
                return patch_document(collection, document)
            elif method == 'DELETE':
                return delete_document(collection, object_id)
    else:
        abort(404)
```

editing is **forbidden** if ETag is not provided

Concurrency control

Create/Update/Delete are controlled by ETag

```
def edit_document(collection, object_id, method):
    document = find_document(collection, object_id)
    if document:
        header_etag = request.headers.get('If-Match')
        if header_etag is None:
            return prep_response('If-Match missing from request header',
                                 status=403)
        if header_etag != get_etag(document[LAST_UPDATED]):
            # Precondition failed
            abort(412)
        else:
            if method in ('PATCH', 'POST'):
                return patch_document(collection, document)
            elif method == 'DELETE':
                return delete_document(co
    else:
        abort(404)
```

client and server representations don’t match. **Precondition failed.**

Concurrency control

Create/Update/Delete are controlled by ETag

```
def edit_document(collection, object_id, method):
    document = find_document(collection, object_id)
    if document:
        header_etag = request.headers.ge
        if header_etag is None:
            return prep_response('If-Mat
                                 status=
    if header_etag != get_etag(docume
        # Precondition failed
        abort(412)
    else:
        if method in ('PATCH', 'POST'):
            return patch_document(collection, document)
        elif method == 'DELETE':
            return delete_document(collection, object_id)
else:
    abort(404)
```

client and server representation **match**, go ahead with the edit

Sending cache & concurrency directives back to clients

Cache & Concurrency

```
def prep_response(dct, last_modified=None, etag=None, status=200):
    (...)
    resp.headers.add('Cache-Control',
                     'max-age=%s,must-revalidate' & 30)
    resp.expires = time.time() + 30
    if etag:
        resp.headers.add('ETag', etag)
    if last_modified:
        resp.headers.add('Last-Modified', date_to_str(last_modified))
    return resp
```

encodes ‘dct’ according to client’s accepted MIME Data-Type

(click here see that slide)

Cache & Concurrency

```
def prep_response(dct, last_modified=None, etag=None, status=200):
    (...)
    resp.headers.add('Cache-Control',
                     'max-age=%s,must-revalidate' & 30)
    resp.expires = time.time() + 30
    if etag:
        resp.headers.add('ETag', etag)
    if last_modified:
        resp.headers.add('Last-Modified', date_to_str(last_modified))
    return resp
```

Cache-Control, a directive for HTTP/1.1 clients (and later) -RFC2616

Cache & Concurrency

```
def prep_response(dct, last_modified=None, etag=None, status=200):
    (...)
    resp.headers.add('Cache-Control',
                     'max-age=%s,must-revalidate' & 30)
    resp.expires = time.time() + 30
    if etag:
        resp.headers.add('ETag', etag)
    if last_modified:
        resp.headers.add('Last-Modified', date_to_str(last_modified))
    return resp
```

Expires, a directive for HTTP/1.0 clients

Cache & Concurrency

```
def prep_response(dct, last_modified=None, etag=None, status=200):
    (...)
    resp.headers.add('Cache-Control',
                     'max-age=%s,must-revalidate' & 30)
    resp.expires = time.time() + 30
    if etag:
        resp.headers.add('ETag', etag)
    if last_modified:
        resp.headers.add('Last-Modified', date_to_str(last_modified))
    return resp
```

ETag. Notice that we don't compute it on the rendered representation, this is by design.

Cache & Concurrency

```
def prep_response(dct, last_modified=None, etag=None, status=200):
    (...)
    resp.headers.add('Cache-Control',
                     'max-age=%s,must-revalidate' & 30)
    resp.expires = time.time() + 30
    if etag:
        resp.headers.add('ETag', etag)
    if last_modified:
        resp.headers.add('Last-Modified', date_to_str(last_modified))
    return resp
```

And finally, we add the **Last-Modified** header tag.

Cache & Concurrency

```
def prep_response(dct, last_modified=None, etag=None, status=200):
    (...)
    resp.headers.add('Cache-Control',
                     'max-age=%s,must-revalidate' & 30)
    resp.expires = time.time() + 30
    if etag:
        resp.headers.add('ETag', etag)
    if last_modified:
        resp.headers.add('Last-Modified', date_to_str(last_modified))
    return resp
```

the response object is now complete and ready to be returned to the client

that's one long ass acronym

#5

HATEOAS

"Hypertext As The Engine Of Application State"

HATEOAS in a Nutshell

- clients interact entirely through hypermedia provided dynamically by the server
- clients need no prior knowledge about how to interact with the server
- clients access an application through a single well known URL (the entry point)
- All future actions the clients may take are discovered within resource representations returned from the server

It’s all about Links

resource representation includes links to related resources

Collection Representation

```
{
 "links":[
    "<link rel='parent' title='home' href='http://api.example.com/' />",
    "<link rel='collection' title='contacts'
           href='http://api.example.com/Contacts' />",
    "<link rel='next' title='next page'
           href='http://api.example.com/Contacts?page=2' />"
    ],
    "contacts":[
       {
          "updated":"Wed, 06 Jun 2012 14:19:53 UTC",
          "name":"Jon Doe",
          "age": 27,
          "etag":"7a9f477cde424cf93a7db20b69e05f7b680b7f08",
          "link":"<link rel='self' title='C
              href='http://api.example.com/
              4f46445fc88e201858000000' />'
          "_id":"4f46445fc88e201858000000",
       },
     ]
}
```

every resource representation provides a **links** section with navigational info for clients

Collection Representation

```
{
 "links":[
    "<link rel='parent' title='home' href='http://api.example.com/' />",
    "<link rel='collection' title='contacts'
           href='http://api.example.com/Contacts' />",
    "<link rel='next' title='next page'
           href='http://api.example.com/Contacts?page=2' />"
    ],
    "contacts":[
       {
          "updated":"Wed, 06 Jun 2012 14:19:53 UTC",
          "name":"Jon Doe",
          "age": 27,
          "etag":"7a9f477cde424cf93a7db20b69e05f7b680b7f08",
          "link":"<link rel='self' title='C
              href='http://api.example.com/
              4f46445fc88e201858000000' />"
          "_id":"4f46445fc88e201858000000",
       },
     ]
}
```

the **rel** attribute provides the **relationship** between the linked resource and the one currently represented

Collection Representation

```
{
 "links":[
    "<link rel='parent' title='home' href='http://api.example.com/' />",
    "<link rel='collection' title='contacts'
            href='http://api.example.com/Contacts' />",
    "<link rel='next' title='next page'
            href='http://api.example.com/Contacts?page=2' />"
    ],
    "contacts":[
       {
          "updated":"Wed, 06 Jun 2012 14:19:53 UTC",
          "name":"Jon Doe",
          "age": 27,
          "etag":"7a9f477cde424cf93a7db20b69e05f7b680b7f08",
          "link":"<link rel='self' title='C
              href='http://api.example.com/
              4f46445fc88e201858000000' />"
          "_id":"4f46445fc88e201858000000",
       },
     ]
}
```

the **title** attribute provides a tag (or description) for the linked resource. Could be used as a caption for a client button.

Collection Representation

```
{
 "links":[
    "<link rel='parent' title='home' href='http://api.example.com/' />",
    "<link rel='collection' title='contacts'
            href='http://api.example.com/Contacts' />",
    "<link rel='next' title='next page'
            href='http://api.example.com/Contacts?page=2' />"
    ],
    "contacts":[
       {
          "updated":"Wed, 06 Jun 2012 14:19:53 UTC",
          "name":"Jon Doe",
          "age": 27,
          "etag":"7a9f477cde424cf93a7db20b69e05f7b680b7f08",
          "link":"<link rel='self' title='C
              href='http://api.example.com/
              4f46445fc88e201858000000' />"
          "_id":"4f46445fc88e201858000000",
       },
     ]
}
```

the **href** attribute provides and **absolute path** to the resource (the “permanent identifier” per REST def.)

Collection Representation

```
{
 "links":[
    "<link rel='parent' title='home' href='http://api.example.com/' />"
    "<link rel='collection' title='contact
              href='http://api.example.com/
    "<link rel='next' title='next page'
              href='http://api.example.com/
   ],
   "contacts":[
      {
         "updated":"Wed, 06 Jun 2012 14:19:53 UTC",
         "name":"Jon Doe",
         "age": 27,
         "etag":"7a9f477cde424cf93a7db20b69e05f7b680b7f08",
         "link":"<link rel='self' title='Contact'
             href='http://api.example.com/Contacts/
             4f46445fc88e201858000000' />",
         "_id":"4f46445fc88e201858000000",
      },
    ]
}
```

every resource listed exposes its own link, which will allow the client to perform PATCH, DELETE etc. on the resource

Collection Representation

```
{
 "links":[
    "<link rel='parent' title='home' href=
    "<link rel='collection' title='contact
              href='http://api.example.com/
    "<link rel='next' title='next page'
              href='http://api.example.com/
    ],
    "contacts":[
       {
          "updated":"Wed, 06 Jun 2012 14:19:53 UTC",
          "name":"Jon Doe",
          "age": 27,
          "etag":"7a9f477cde424cf93a7db20b69e05f7b680b7f08",
          "link":"<link rel='self' title='Contact'
              href='http://api.example.com/Contacts/
              4f46445fc88e201858000000' />",
          "_id":"4f46445fc88e201858000000",
       },
     ]
}
```

while we are here,
notice how every resource
also exposes its own **etag**,
last-modified date.

HATEOAS

The API entry point (the homepage)

```
@app.route('/', methods=['GET'])
def home():
    response = {}
    links = []
    for collection in DOMAIN.keys():
        links.append("<link rel='child' title='%(name)s'"
                     "href='%(collectionURI)s' />" %
                     {'name': collection,
                      'collectionURI': collection_URI(collection)})
    response['links'] = links
    return response
```

the **API homepage** responds to **GET** requests and provides links to its top level resources to the clients

HATEOAS

The API entry point (the homepage)

```
@app.route('/', methods=['GET'])
def home():
    response = {}
    links = []
    for collection in DOMAIN.keys():
        links.append("<link rel='child' title='%(name)s'"
                     "href='%(collectionURI)s' />" %
                     {'name': collection,
                      'collectionURI': collection_URI(collection)})
    response['links'] = links
    return response
```

for every collection of resources...

HATEOAS

The API entry point (the homepage)

```
@app.route('/', methods=['GET'])
def home():
    response = {}
    links = []
    for collection in DOMAIN.keys():
        links.append("<link rel='child' title='%(name)s'"
                     "href='%(collectionURI)s' />" %
                     {'name': collection,
                      'collectionURI': collection_URI(collection)})
    response['links'] = links
    return response
```

... provide relation, title and link, or the **persistent identifier**

Wanna see it running?

Hopefully it won't explode right into my face

Only complaint I have with Flask so far...

Most recent HTTP methods not supported

508 NOT MY FAULT

Not supported yet

208 WORKS FOR ME

Not supported yet

it isn't even my joke!

Just kidding!

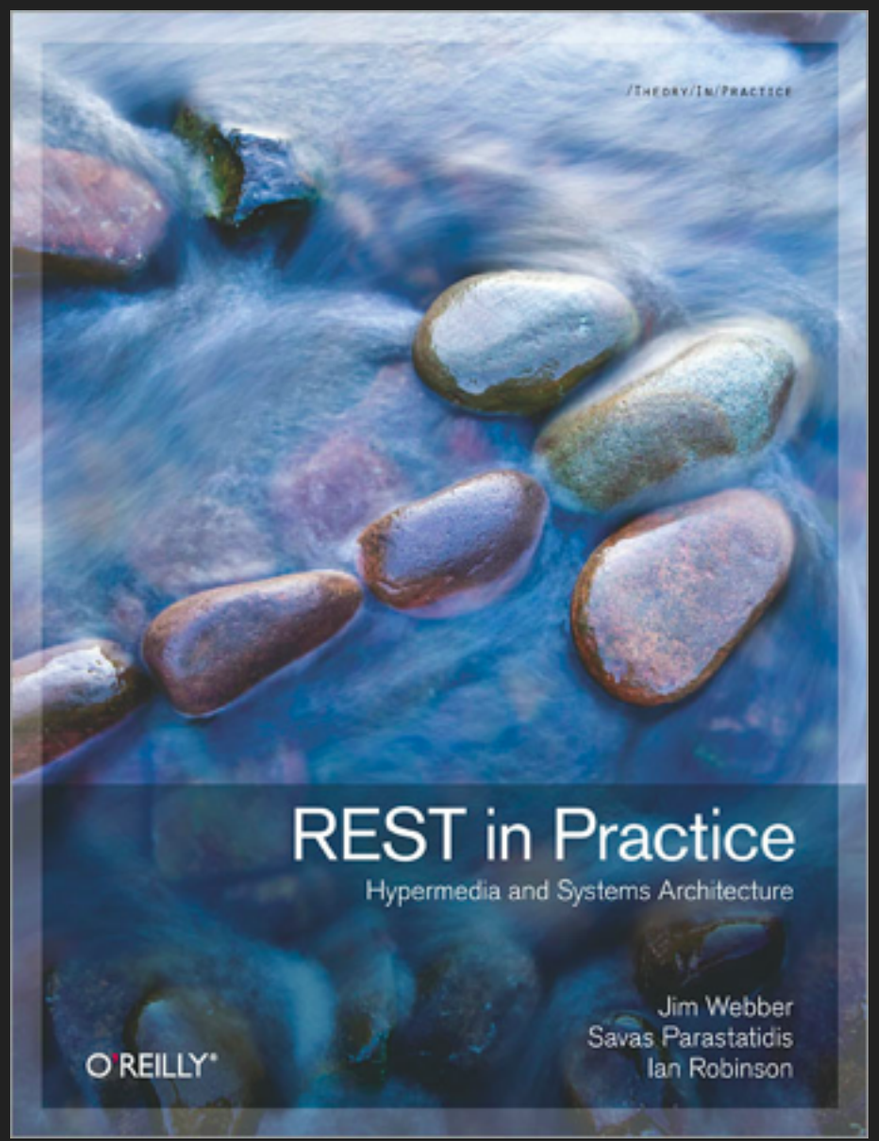
Open Source it?

as a Flask extension maybe?

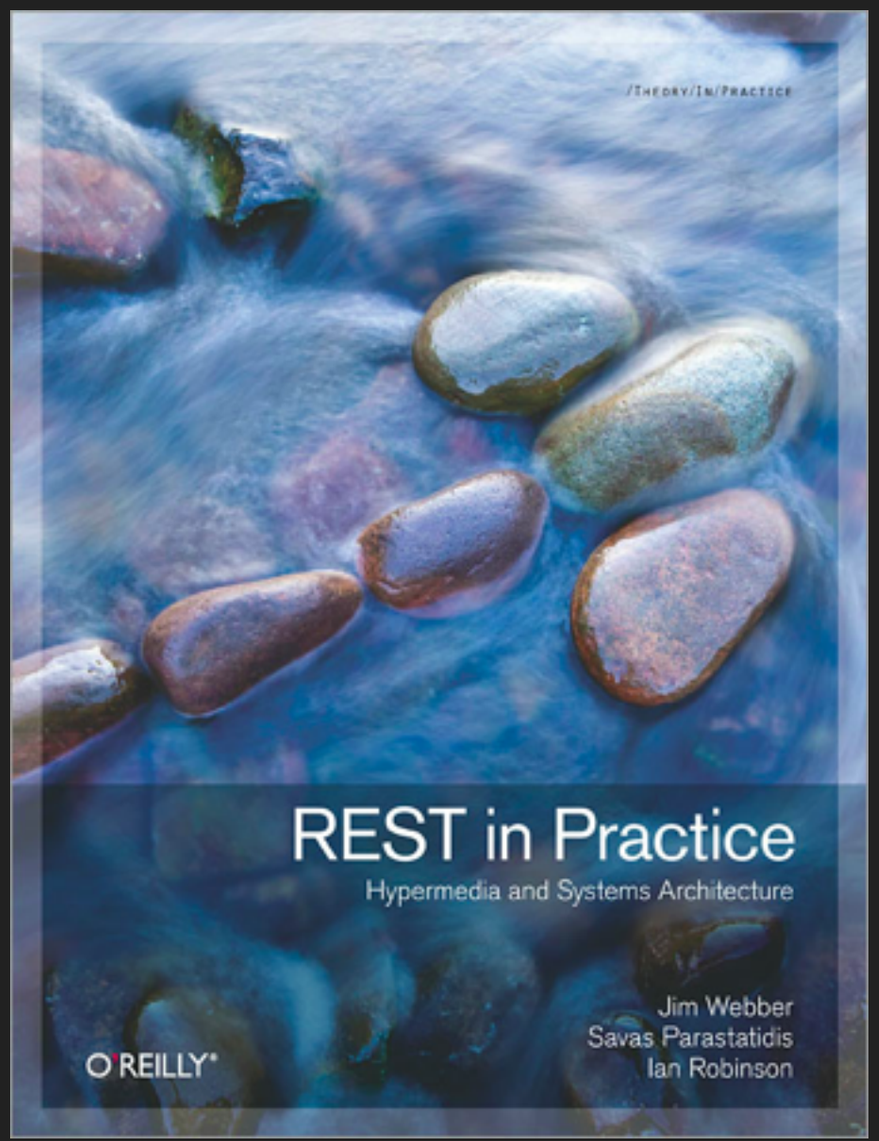
Web Resources

- Richardson Maturity Model: steps toward the glory of REST
  by Richard Flowers
- RESTful Service Best Practices
  by Todd Fredrich
- What Exactly is RESTful Programming?
  StackOverflow (lots of resources)
- API Anti-Patterns: How to Avoid Common REST Mistakes
  by Tomas Vitvar

Excellent Books



Excellent Books



I'm getting a cut.

I wish!

Thank you.

@nicolaiarocci