THE UNIVERSITY OF WESTERN AUSTRALIA

Topic 13 RESTful APIs

CITS3403 Agile Web Development

Reading:
The Flask Mega Tutorial, Chapter 23
Miguel Grinberg

Semester 1, 2022

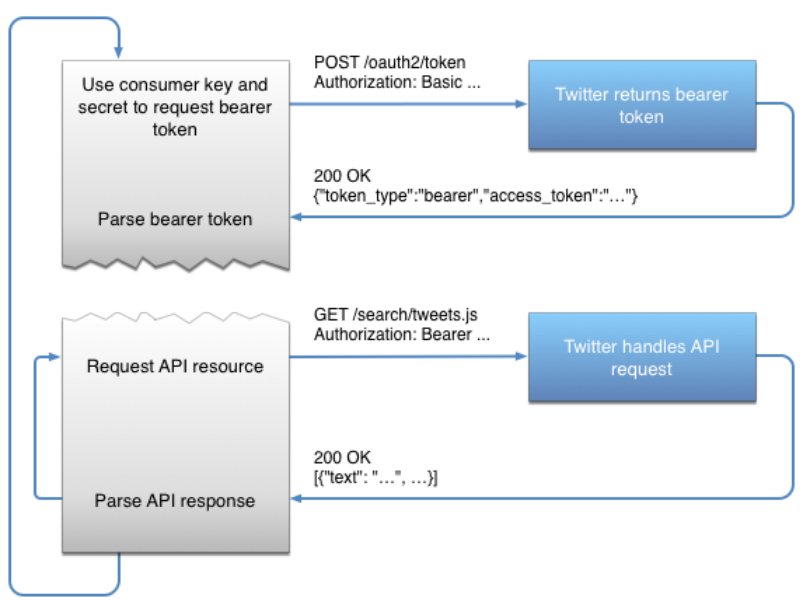
Application Programming Interfaces

THE UNIVERSITY OF WESTERN AUSTRALIA

- The web applications we have looked at so far have been complete applications. The backend provides the logic and persistent data storage and then serves a graphical user interface to a browser for a user to access the logic.
- This has the logic and the presentation coupled together. If we wanted to have a mobile version of the application, (iOS or Android or…) or some other way of interacting with the web we would have to rebuild it.
- An application programming interface is a means to provide the logic and data structures of your app as a service to other developers so they can embed the functionality into different applications and customise the user interface.
- Common examples are the Google Maps API, Dropbox API, Facebook API, ...
-

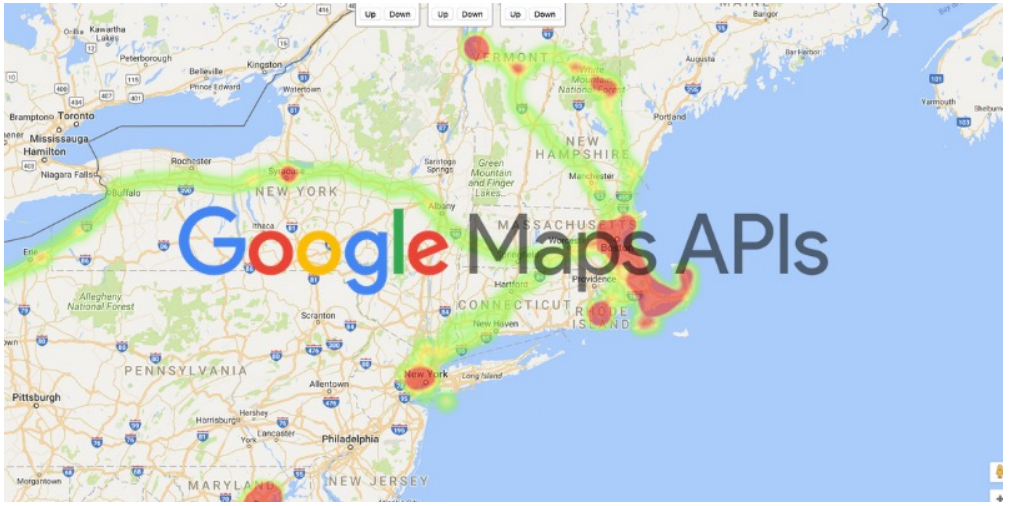
Common APIs

- APIs allow developers to release software as a service, and is a key building block for modern web applications.
- Web APIs work with http requests, in standardized formats with docuemented response types.



REST API Developer Endpoint Reference #

Resource	Base Route
Posts	/wp/v2/posts
Post Revisions	/wp/v2/revisions
Categories	/wp/v2/categories
Tags	/wp/v2/tags
Pages	/wp/v2/pages
Comments	/wp/v2/comments
Taxonomies	/wp/v2/taxonomies



Representational State Transfer

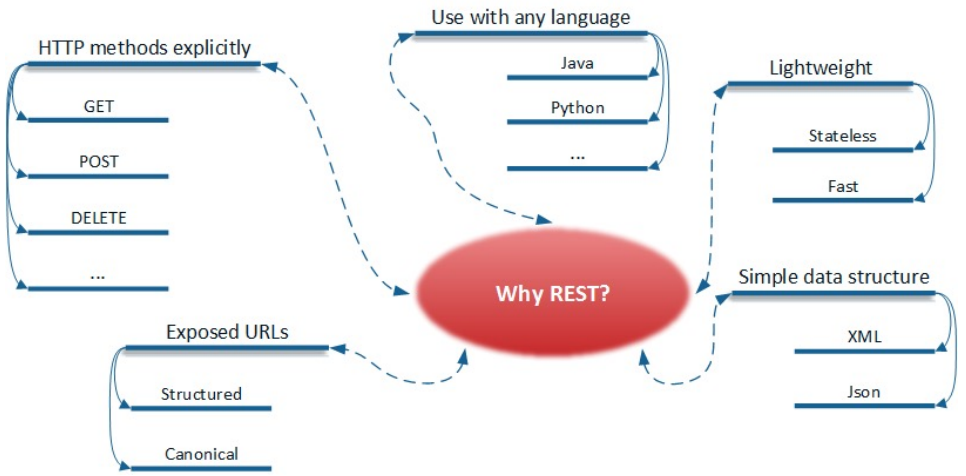
THE UNIVERSITY OF WESTERN AUSTRALIA

- REpresentational State Transfer (REST) is an architecture for the web that describe interactions with web based resources.
- HTTP is stateless, so there is no memory between transactions. REST uses the current page as a proxy for state, and operations to move from one to the other.
- REST was defined in 2000 by Roy Thomas Fielding:

Throughout the HTTP standardization process, I was called on to defend the design choices of the Web. That is an extremely difficult thing to do within a process that accepts proposals from anyone on a topic that was rapidly becoming the center of an entire industry. …That process honed my model down to a core set of principles, properties, and constraints that are now called REST.

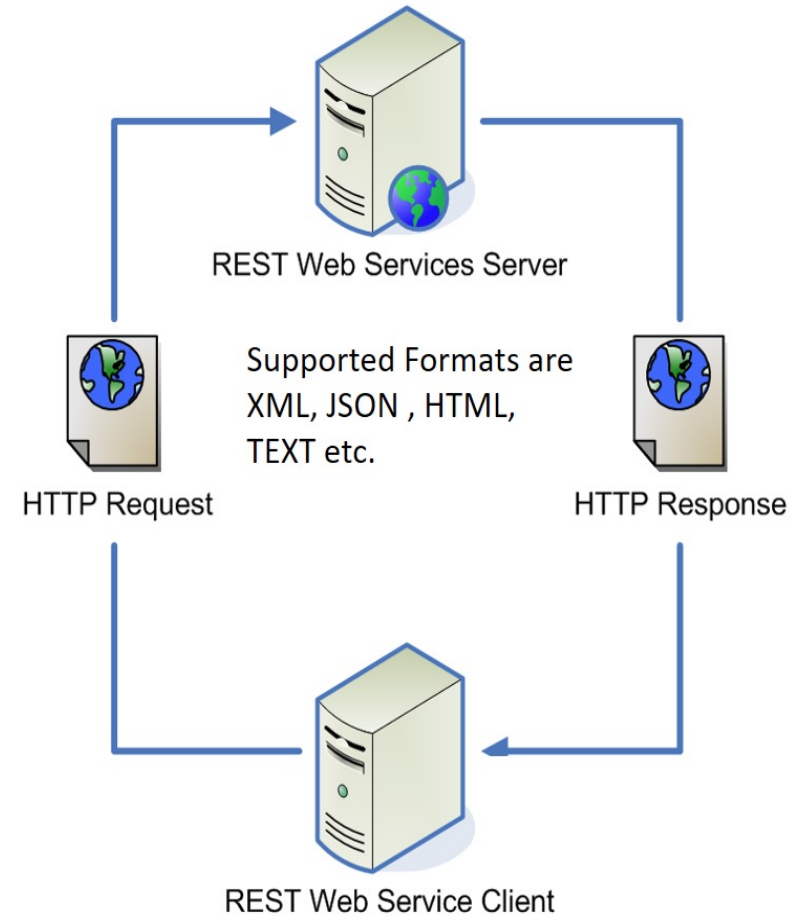
The six Characteristics of REST

- Dr Fielding was one of the principal authors of the HTTP protocol, and his thesis sought to make the design choices of the web explicit.
- In his thesis, Dr Fielding set out six high level characteristics of REST: *client-server, layered system, cache, code on demand, stateless, uniform interface.*
- These are not enforced, so are interpreted differently by developers, and there is one optional characteristic.
- Most big companies, like Google, Facebook and Twitter implement a pragmatic version of REST.



1. Client Server Model

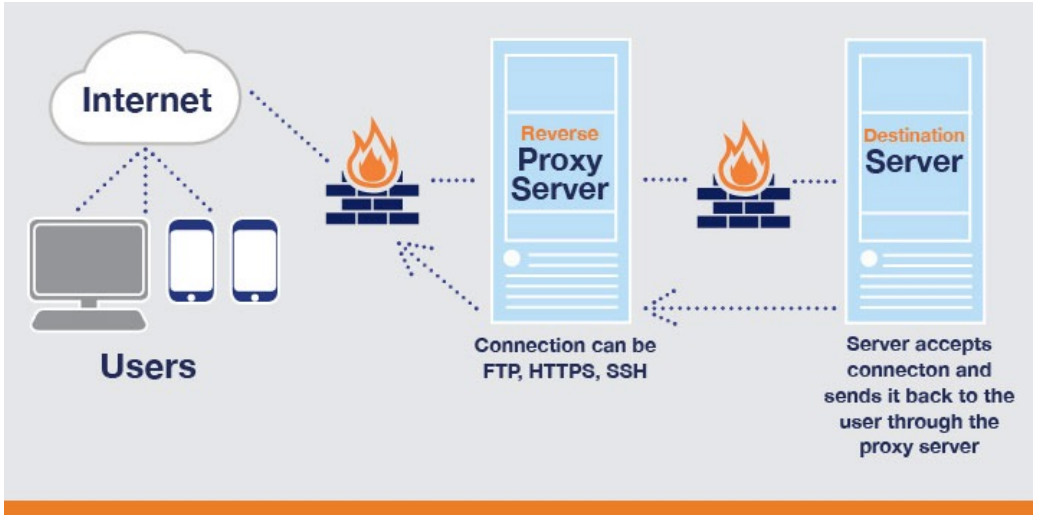
- The client server model sets out the different roles of the client and the server in the system.
- They should be clearly differentiated and running as separate processes, and communicate over a transport layer.
- In practice the interface between the client and the server is through HTTP, and the transport layer is TCP/IP.



RESTful Web Service Architecture

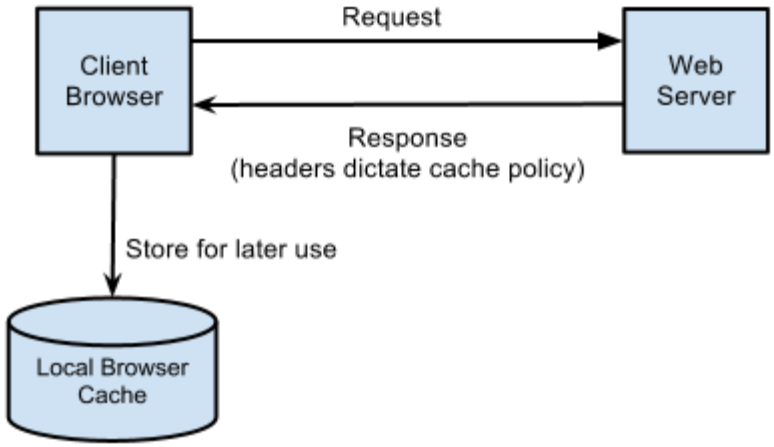
2. Layered System

- The layered system characteristic states that there does not need to be a direct link between the client and the server, and that they can communicate through intermediate nodes.
- The client does not need to distinguish between the actual server and an intermediary, and the server doesn't need to know whetehr it is communicating directly with the client.
- This encapsulates the abstract nature of the interface , and allows web services to scale, through proxy servers and load balancers.



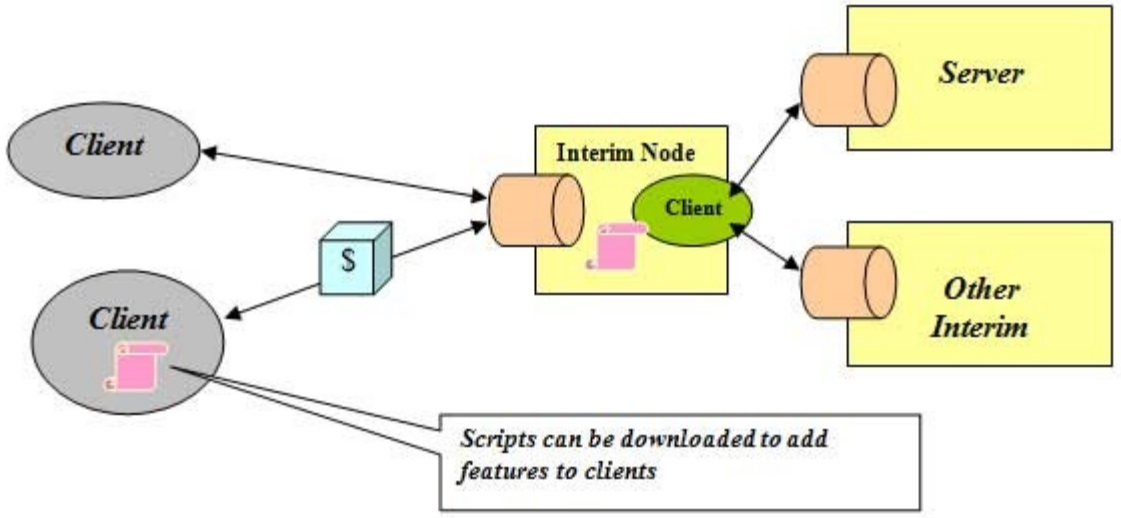
3. Cache

- The cache principle states that it is acceptable for the client or intermediaries to cache responses to requests, and serve these without going back to the server every time.
- This allows for efficient operation of the web.
- The server needs to specify what can and can't be cached, (i.e. what is static and dynamic data)
- Also, anything encrypted cannot be cached by an intermediary.
- All web browsers implement a cache to save reloading the same static files.



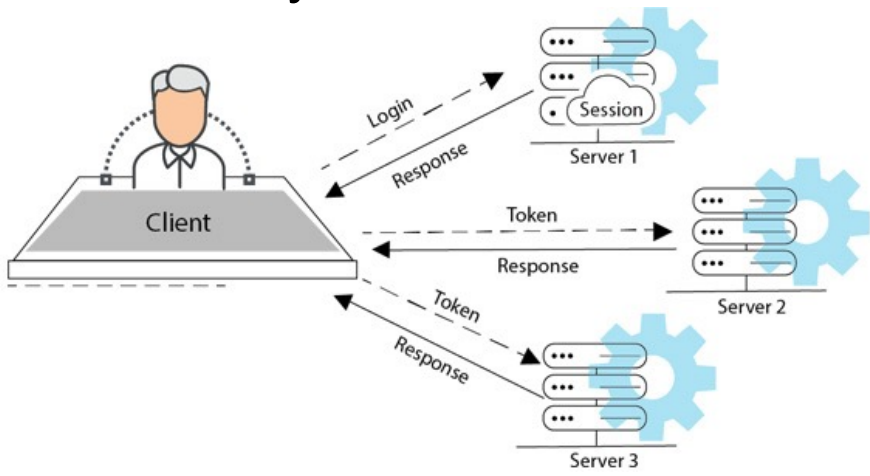
4. Code on Demand (optional)

- The code on demand principle states that the server can provide executable code in responses to a client.
- This is common practice with web browsers, where javascript is provided to be run by the client.
- However this isn't commonly included in REST APIs since there is no standard for executable code, so for example, iOS won't execute javascript.



5. Stateless

- Statelessness is one of the key properties of the HTTP protocol, and most associated with REST APIs.
- It states that the server should not maintain any memory of prior transactions, and every request from the client should include sufficient context for the server to satisfy the request.
- The *representative state* is in the url or route that is requested by the client, and is sent through with each request.
- This makes the service easy to scale, as a load balancer can deploy two servers to satisfy arbitrary requests, and they do not need to communicate.
- Pragmatically, many REST APIs do record state for session management.



6. Uniform Interface

THE UNIVERSITY OF WESTERN AUSTRALIA

- The most important, and most vague, requirement of REST is that there be a uniform interface, so clients in principle do not need to be specifically designed to consume a server.
- The four aspects of the uniform interface are:
  - *Unique resource identifers.* This is the url, and typically is of the form `api/users/<id>`
  - *Resource representations.* The data exchange between client and server should be through an agreed format, typically JSON, but possibly others. HTTP can do content negotiation.
  - *Self descriptive messages.* The communication between client and server should make the intended action clear.
  - *Hypermedia links.* A client should be able to discover new resources by following provided hyperlinks.

RESTful operations

- The standard CRUD operations are *create, read, update* and *delete*, and these are typical ways to interact with our data model.
- In web apps, these operations are mapped to the operations: POST, GET, PUT (PATCH) and DELETE.
- These operations can be applied to each route in our application to allow interaction with the server side data model.

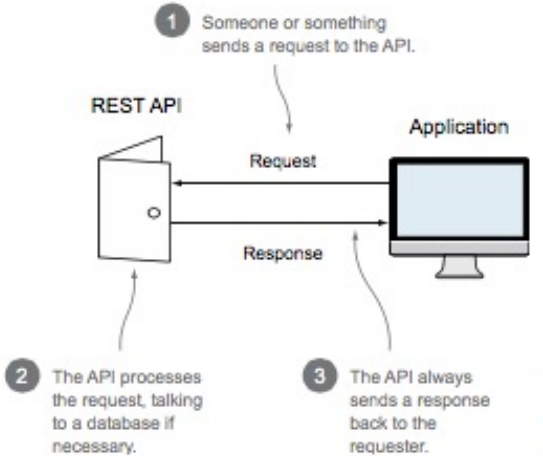


Figure 6.2 A REST API takes incoming HTTP requests, does some processing, and returns HTTP responses.

URL	HTTP Verb	POST Body	Result
/api/movies	GET	empty	Returns all movies
/api/movies	POST	JSON String	New movie Created
/api/movies/:id	GET	empty	Returns single movie
/api/movies/:id	PUT	JSON string	Updates an existing movie
/api/movies/:id	DELETE	empty	Deletes existing movie

REST URLs and Operations

REST APIs offer a standard approach to accessing web-based resources

- Request URLs for a REST API have a simple standard.
- Consider each collection in your data base as having an associated URL.

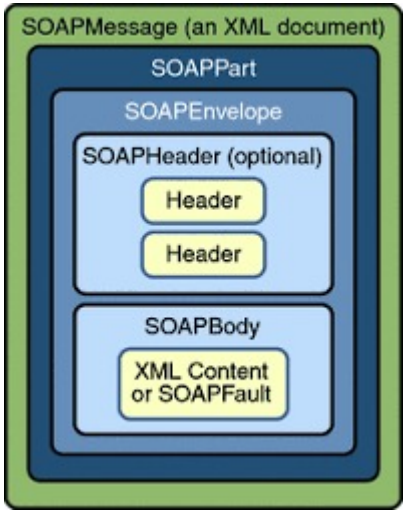
HTTP METHOD	CRUD	ENTIRE COLLECTION (E.G. /USERS)	SPECIFIC ITEM (E.G. /USERS/123)
POST	Create	201 (Created), 'Location' header with link to /users/{id} containing new ID.	Avoid using POST on single resource
GET	Read	200 (OK), list of users. Use pagination, sorting and filtering to navigate big lists.	200 (OK), single user. 404 (Not Found), if ID not found or invalid.
PUT	Update/Replace	404 (Not Found), unless you want to update every resource in the entire collection of resource.	200 (OK) or 204 (No Content). Use 404 (Not Found), if ID not found or invalid.
PATCH	Partial Update/Modify	404 (Not Found), unless you want to modify the collection itself.	200 (OK) or 204 (No Content). Use 404 (Not Found), if ID not found or invalid.
DELETE	Delete	404 (Not Found), unless you want to delete the whole collection — use with caution.	200 (OK). 404 (Not Found), if ID not found or invalid.

HTTP methods

Uniform Resource Locator (URL)	GET	PUT	POST	DELETE
Collection, such as `http://api.example.com/resources/`	**List** the URIs and perhaps other details of the collection's members.	**Replace** the entire collection with another collection.	**Create** a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation.[17]	**Delete** the entire collection.
Element, such as `http://api.example.com/resources/item17`	**Retrieve** a representation of the addressed member of the collection, expressed in an appropriate Internet media type.	**Replace** the addressed member of the collection, or if it does not exist, **create** it.	Not generally used. Treat the addressed member as a collection in its own right and **create** a new entry within it.[17]	**Delete** the addressed member of the collection.

REST vs SOAP

- The Simple Object Access Protocol is often seen as an alternative to REST and is used in many enterprise systems.
- It is a protocol, rather than an architectural style like REST, and is much stricter in its implementation.



SOAP vs. REST Comparison: Which is Right for You?

Difference	SOAP	REST
Style	Protocol	Architectural style
Function	Function-driven: transfer structured information	Data-driven: access a resoruce for data
Data format	Only uses XML	Permits many data formats, including plain text, HTML, XML, and JSON
Security	Supports WS-Security and SSL	Supports SSL and HTTPS
Bandwidth	Requires more resources and bandwidth	Requires fewer resources and is lightweight
Data cache	Can not be cached	Can be cached
Payload handling	Has a strict communication contract and needs knowledge of everything before any interaction	Needs no knowledge of the API
ACID compliance	Has built-in ACID compliance to reduce anomalies	Lacks ACID compliance

Visit upwork.com to learn more

Upwork

Providing a REST API to a web application.

THE UNIVERSITY OF WESTERN AUSTRALIA

- The simple project described here is quite basic: it does all the hard work on the server side.
- A more responsive web application does most of the work on the client side.
- A REST API provides a web interface to the back end data model.
- This serves JSON to the client application, but all rendering of the data is then done on the client side by JavaScript modules (e.g. Angular, or AJAX and jQuery)
- In fact, the client side can implement a full MVC architecture, where the models interface with the API.
- We can augment a flask web application so that it provides a REST API but shares the database with the web application.

New application structure

- Currently our Flask Application looks something like:

```
myapp\
        app\
                __init__.py
                FORMS.PY
                models.py
                controllers.py
                routes.py
                templates\
                        base.html...
                static\
                        bootstrap.css...
        app.db
        config.py
        myapp.py
        tests\
                unittest.py
        virtualenv\
                ....
```

```
app\
        __init__.py
        api\
                __init__.py
                auth.py
                models_api.py
                token_api.pi
        FORMS.PY
        models.py
```

- We are going to add an `api` module in the app folder containing:
  - `__init__.py` to initialise the api
  - `auth.py` to handle the token based authentication
  - `models_api.py` to handle the api routes for each model
  - `token_api.py` to handle the tokens.

Choosing a route structure.

- The route structure indicates the requests that the application should serve, or the resources a client can access.
- They are typically aligned with the model structure.
- The api can assign methods to routes.
- This application structure form the mega-tutorial uses blueprints.

HTTP Method	Resource URL	Notes
`GET`	*/api/users/<id>*	Return a user.
`GET`	*/api/users*	Return the collection of all users.
`GET`	*/api/users/<id>/followers*	Return the followers of this user.
`GET`	*/api/users/<id>/followed*	Return the users this user is following.
`POST`	*/api/users*	Register a new user account.
`PUT`	*/api/users/<id>*	Modify a user.

app/api/users.py: User API resource placeholders.

```
from app.api import bp

@bp.route('/users/<int:id>', methods=['GET'])
def get_user(id):
    pass

@bp.route('/users', methods=['GET'])
def get_users():
    pass

@bp.route('/users/<int:id>/followers', methods=['GET'])
def get_followers(id):
    pass

@bp.route('/users/<int:id>/followed', methods=['GET'])
def get_followed(id):
    pass

@bp.route('/users', methods=['POST'])
def create_user():
    pass

@bp.route('/users/<int:id>', methods=['PUT'])
def update_user(id):
    pass
```

A simpler structure

- Blueprints use the factory pattern to make testing and deployment easier, but are out of scope for this course.
- A simpler structure is as follows:

```
1 from app.api import student_api, project_api, token_api
2 from app import app
~
~
app/api/__init__.py
```

```
1 from flask import Flask
2 from config import Config
3 from flask_sqlalchemy import SQLAlchemy
4 from flask_migrate import Migrate
5 from flask_login import LoginManager
6
7 app = Flask(__name__)
8 app.config.from_object('config.DevelopmentConfig')
9 db = SQLAlchemy(app)
10 migrate = Migrate(app, db)
11 login = LoginManager(app)
12 login.login_view = 'login'
13
14 #from app import routes,models
15 from app import routes,models,api
~
~
app/__init__.py                    1,1           All
```

```
1 from app import app, db
2 from app.models import Student,Project,Lab
3 from app.api.errors import bad_request, error_response
4 from flask import jsonify, url_for, request, g, abort
5 from app.api.auth import token_auth
6
7
8 @app.route('/api/students/<int:id>',methods=['GET'])
9 @token_auth.login_required
10 def get_student(id):
11   if g.current_user != id:
12     abort(403)
13   return jsonify(Student.query.get_or_404(id).to_dict())
14
15 @app.route('/api/students',methods=['POST'])
16 def register_student():
17   data = request.get_json() or {}
18   if 'id' not in data or 'pin' not in data:
19     return bad_request('Must include student number and pin')
20   student = Student.query.get(data['id'])
21   if student is None:
22     return bad_request('Unknown student')
23   if student.password_hash is not None:
24     return bad_request('Student already registered')
25   student.from_dict(data)
26   db.session.commit()
27   response =jsonify(user.to_dict())
28   response.status_code = 201 #creating a new resource should chare the
   n....
29   response.headers['Location'] = url_for('get_student',id=student.id)
30   return response
31
32 @app.route('/api/students/<int:id>',methods=['PUT'])
33 @token_auth.login_required
34 def update_student(id):
35   if g.current_user != id:
36     abort(403)
37   data = request.get_json() or {}
38   student = Student.query.get(id)
39   if student is None:
40     return bad_request('Unknown student')
41   if student.password_hash is None:
42     return bad_request('Student not registered')
43   student.from_dict(data)
44   db.session.commit()
45   return jsonify(student.to_dict())
46
app/api/student_api.py                    1,1
```

Choosing a JSON structure

- The requests and responses to the API needs to be in some standard format. For each route we can assign a JSON structure for data transfer.
- We add methods to our models to read from and write to the JSON.

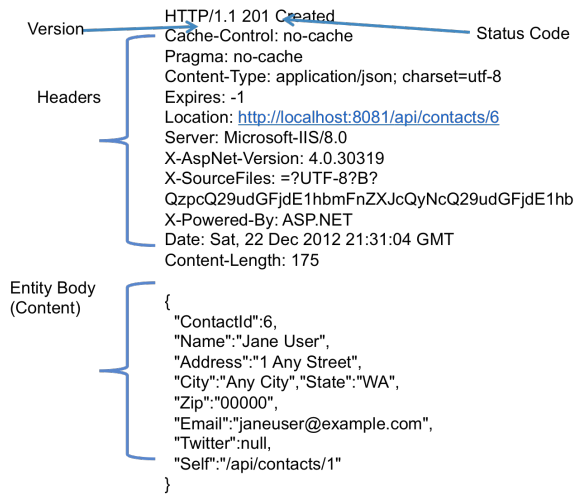
```
{
    "id": 123,
    "username": "susan",
    "password": "my-password",
    "email": "susan@example.com",
    "last_seen": "2017-10-20T15:04:27Z",
    "about_me": "Hello, my name is Susan!",
    "post_count": 7,
    "follower_count": 35,
    "followed_count": 21,
    "_links": {
        "self": "/api/users/123",
        "followers": "/api/users/123/followers",
        "followed": "/api/users/123/followed",
        "avatar": "https://www.gravatar.com/avatar/..."
    }
}
```

```
76   '''Adding in dictionary methods to convert to JSON
77      Format
78      {
79      'id':'19617810',
80      'first_name':'Timothy',
81      'surname': 'French',
82      'prefered_name':'Tim',
83      'cits3403':False,
84      'pin':'0000',
85      '_links':{
86         'project': 'api/student/19617810/project'
87       }
88     }'''
89
90   def to_dict(self):
91     data = {
92         'id': self.id,
93         'first_name':self.first_name,
94         'surname': self.surname,
95         'prefered_name': self.prefered_name,
96         'cits3403':self.cits3403,
97         '_links': {'project':url_for('get_student_project',id = self.id)}
98     }
99     return data
100
101  def from_dict(self, data):
102    if 'prefered_name' in data:
103      self.prefered_name=data['prefered_name']
104    if 'pin' in data :
105      self.set_password(data['pin'])
106
app/models.py                                    103,1
```

Error messages

- As we no longer have a web page to display errors, we need to send them as responses.
- The `jsonify` module in flask will automatically build a JSON response with the JSON payload and the response code.
- `bad_request` is just a wrapper for any error caught when trying to serve a request.

```
1  from flask import jsonify
2  from werkzeug.http import HTTP_STATUS_CODES
3
4  def error_response(status_code, message=None):
5    payload={'error': HTTP_STATUS_CODES.get(status_code, 'Unknown Error')}
6    if message:
7      payload['message'] = message
8    response = jsonify(payload)
9    response.status_code = status_code
10   return response
11
12 def bad_request(message):
13   return error_response(400, message)
~
~
app/api/errors.py                                          1,1
```



HTML response codes

THE UNIVERSITY OF WESTERN AUSTRALIA

1xx Informational

100 Continue
101 Switching Protocols
102 Processing (WebDAV)

2xx Success

★ 200 OK
★ 201 Created
202 Accepted
203 Non-Authoritative Information
★ 204 No Content
205 Reset Content
206 Partial Content
207 Multi-Status (WebDAV)
208 Already Reported (WebDAV)
226 IM Used

3xx Redirection

300 Multiple Choices
301 Moved Permanently
302 Found
303 See Other
★ 304 Not Modified
305 Use Proxy
306 (Unused)
307 Temporary Redirect
308 Permanent Redirect (experiemental)

4xx Client Error

★ 400 Bad Request
★ 401 Unauthorized
402 Payment Required
★ 403 Forbidden
★ 404 Not Found
405 Method Not Allowed
406 Not Acceptable
407 Proxy Authentication Required
408 Request Timeout
★ 409 Conflict
410 Gone
411 Length Required
412 Precondition Failed
413 Request Entity Too Large
414 Request-URI Too Long
415 Unsupported Media Type
416 Requested Range Not Satisfiable
417 Expectation Failed
418 I'm a teapot (RFC 2324)
420 Enhance Your Calm (Twitter)
422 Unprocessable Entity (WebDAV)
423 Locked (WebDAV)
424 Failed Dependency (WebDAV)
425 Reserved for WebDAV
426 Upgrade Required
428 Precondition Required
429 Too Many Requests
431 Request Header Fields Too Large
444 No Response (Nginx)
449 Retry With (Microsoft)
450 Blocked by Windows Parental Controls (Microsoft)
451 Unavailable For Legal Reasons
499 Client Closed Request (Nginx)

5xx Server Error

★ 500 Internal Server Error
501 Not Implemented
502 Bad Gateway
503 Service Unavailable
504 Gateway Timeout
505 HTTP Version Not Supported
506 Variant Also Negotiates (Experimental)
507 Insufficient Storage (WebDAV)
508 Loop Detected (WebDAV)
509 Bandwidth Limit Exceeded (Apache)
510 Not Extended
511 Network Authentication Required
598 Network read timeout error
599 Network connect timeout error

Serving the routes: GET requests

- The `@app.route` decorator allows us to specify parameters, which align with the parameter name in the method

```
 8 @app.route('/api/students/<int:id>',methods=['GET'])
 9 @token_auth.login_required
10 def get_student(id):
11   if g.current_user != id:
12     abort(403)
13   return jsonify(Student.query.get_or_404(id).to_dict())
14
app/api/student_api.py
```

- When no parameter is specified for a GET request, the assumption is that the user wants the collection of all resources.

```
 1 from app import app, db
 2 from app.models import Student,Project,Lab
 3 from app.api.errors import bad_request, error_response
 4 from flask import jsonify, url_for, request
 5
 6 @app.route('/api/projects',methods=['GET'])
 7 def list_projects():
 8   projectList = Project.query.all()
 9   projects = []
10   for p in projectList:
11     t = p.get_team()
12     if len(t)==2:
13       team = t[0].prefered_name +' & '+t[1].prefered_name
14     else:
15       team = t[0].prefered_name
16     l = Lab.query.filter_by(lab_id = p.lab_id).first()
17     time = str(l.time)
18     lab = l.lab
19     projects.append({'project_id':p.project_id,'description':
   p.description,'team':team,'lab':lab,'time':time})
20   projects.sort(key = lambda p: p['lab']+p['time'])
21   return jsonify(projects)
22
23 @app.route('/api/available_labs/',methods=['GET'])
24 def get_available_labs():
25   lab_id = request.args.get('lab_id')
26   labs = Lab.get_available_labs()
27   if lab_id!=None:
28     lab = Lab.query.get(lab_id)
29     choices = [{'lab_id': lab.lab_id, 'lab_name': lab.lab+' '
   +str(lab.time)}]
app/api/project_api.py                    1,1           Top
```

Serving the routes: POST requests

- A POST request is used to create a new resource.
- Creating a project is done by a student, so is included in `api/student/<id>/project`
- New resources should include their location in the response

```
15 @app.route('/api/students',methods=['POST'])
16 def register_student():
17   data = request.get_json() or {}
18   if 'id' not in data or 'pin' not in data:
19     return bad_request('Must include student number and pin')
20   student = Student.query.get(data['id'])
21   if student is None:
22     return bad_request('Unknown student')
23   if student.password_hash is not None:
24     return bad_request('Student already registered')
25   student.from_dict(data)
26   db.session.commit()
27   response =jsonify(user.to_dict())
28   response.status_code = 201 #creating a new resource should chare the location....
29   response.headers['Location'] = url_for('get_student',id=student.id)
30   return response
31
app/api/student_api.py                                    1,1           Top
```

```
59 @app.route('/api/students/<int:id>/project/',methods=['POST'])
60 @token_auth.login_required
61 def new_student_project(id):
62   if g.current_user != id:
63     abort(403)
64   data = request.get_json() or {}
65   if 'description' not in data or 'lab_id' not in data:
66     return bad_request('Must include description and lab_id')
67   student = Student.query.get(id)
68   if student is None:
69     return bad_request('Unknown student, or wrong id')
70   if student.project_id is not None:
71     return bad_request('Student already committed')
72   partner=None
73   if 'partner' in data:
74     partner = Student.query.get(data['partner'])
75     if partner is None:
76       return bad_request("Unknown partner")
77     if partner.project_id is not None:
78       return bad_request('Partner already committed')
79   if partner is None and student.cits3403:
80     return bad_request('CITS3403 students require a partner')
81   lab = Lab.query.get(data['lab_id'])
82   if lab is None or not lab.is_available():
83     return bad_request('Lab not available')
84   #all good, create project
85   project=Project();
86   project.description = description
87   project.lab_id=lab.lab_id
88   db.session.add(project)
89   db.session.flush() #generates pk for new project
90   student.project_id = project.project_id
91   if partner is not None:
92     partner.project_id=project.project_id
93   db.session.commit()
94   response =jsonify(project.to_dict())
95   response.status_code = 201 #creating a new resource should chare the
   location....
96   response.headers['Location'] = url_for('new_student_project',id=stude
   nt.id)
97   return response
app/api/student_api.py                                    95,1          51%
```

Serving the routes: PUT requests

- Updating resources is typically done through PUT requests, although some people distinguish between PUT (overwrite resource) and PATCH (update some resource fields).

```
32 @app.route('/api/students/<int:id>',methods=['PUT'])
33 @token_auth.login_required
34 def update_student(id):
35   if g.current_user != id:
36     abort(403)
37   data = request.get_json() or {}
38   student = Student.query.get(id)
39   if student is None:
40     return bad_request('Unknown student')
41   if student.password_hash is None:
42     return bad_request('Student not registered')
43   student.from_dict(data)
44   db.session.commit()
45   return jsonify(student.to_dict())
46
app/api/student_api.py
```

```
100 @app.route('/api/students/<int:id>/project/',methods=['PUT'])
101 @token_auth.login_required
102 def update_student_project(id):
103   if g.current_user != id:
104     abort(403)
105   data = request.get_json() or {}
106   if 'description' not in data or 'lab_id' not in data:
107     return bad_request('Must include description and lab_id')
108   student = Student.query.get(id)
109   if student is None:
110     return bad_request('Unknown student')
111   if student.project_id is None:
112     return bad_request('Student has no project')
113   project = Project.query.get(student.project_id)
114   team = project.get_team()
115   if not team[0].id==current_user.id:
116     partner = team[0]
117   elif len(team)>1:
118     partner = team[1]
119   else:
120     partner=None
121   lab = Lab.query.get(data['lab_id'])
122   if lab is None or (not lab.is_available() and lab.lab_id != project.lab_id):
123     return bad_request('Lab not available')
124   #all good, create project
125   project.description = description
126   project.lab_id=lab.lab_id
127   student.project_id = project.project_id
128   if partner is not None:
129     partner.project_id=project.project_id
130   db.session.commit()
131   return jsonify(project.to_dict())
app/api/student_api.py                                    96,1
```

Serving the routes: DELETE requests

- Finally for our delete operation, we will return the deleted project.
- We also have a delete operation for our authentication token, that has an empty response body

```python
from flask import jsonify, g
from app import app, db
from app.api.auth import basic_auth, token_auth

@app.route('/api/tokens', methods=['POST'])
@basic_auth.login_required
def get_token():
  token = g.current_user.get_token()
  db.session.commit()
  return jsonify({'token':token})

@app.route('/api/tokens', methods=['DELETE'])
@token_auth.login_required
def revoke_token():
  g.current_user.revoke_token()
  db.session.commit()
  return '', 204  # no response body required
```

app/api/token_api.py 1,1

```python
@app.route('/api/students/<int:id>/project/',methods=['DELETE'])
@token_auth.login_required
def delete_student_project(id):
  if g.current_user != id:
    abort(403)
  student = Student.query.get(id)
  if student is None:
    return bad_request('Unknown student, or wrong number')
  if student.project_id is None:
    return bad_request('Student does not have a project')
  project = Project.query.get(student.project_id)
  if project is None:
    return bad_request('Project not found')
  for s in project.get_team():
    s.project_id = None
  db.session.delete(project)
  db.session.commit()
  return jsonify(project.to_dict())
```

app/api/student_api.py 151,1

Consuming a REST API with AJAX and jQuery

THE UNIVERSITY OF WESTERN AUSTRALIA

- We can consume the REST API in a webpage using AJAX and jQuery

```
{
    "id": 166,
    "content": "Hello, World!"
}
```

consume-rest.js

```
$(document).ready(function () {
    $.ajax({
        url: "http://rest-service.guides.spring.io/greeting",
        success: function (data) {
            $("#id").append(data.id);
            $("#content").append(data.content);

        },
        error: function (xhr, ajaxOptions, thrownError) {
            alert(xhr.responseText + "\n" + xhr.status + "\n" + thrownError);
        }
    });

});
```

consume-restful-webservice-jquery.html

```
<!DOCTYPE html>
<html>
    <head>
        <title>How to consume RESTful Web Service using jQuery</title>
        <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/
        <script src="js/consume-rest.js"></script>
    </head>
    <body>
        <div>
            <h3>How to consume RESTful Web Service using jQuery</h3>

            <p id="id">ID: </p>
            <p id="content">Content: </p>

        </div>
    </body>
</html>
```