RESTful APIs

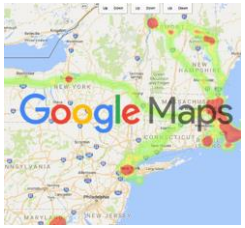
CITS3403 and CITS5505 - Agile Web Development

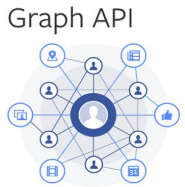
Matthew Daggitt

Semester 1, 2024

Motivation

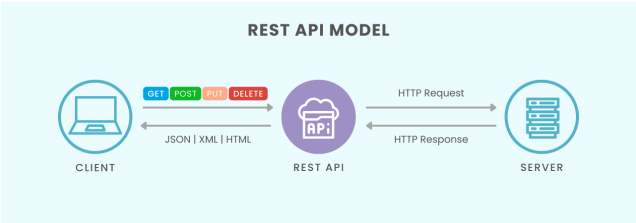
- The web applications we have looked at so far have been complete applications. The backend provides the logic and persistent data storage and then serves a graphical user interface to a browser for a user to access the logic.
- This has the logic and the presentation coupled together. If we wanted 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 (API) 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, ...





Motivation

- APIs allow developers to release software as a service and is a key building block for modern web applications.
- Web APIs use HTTP requests, in standardized formats with documented response types.
- The most common API design is that based on REST APIs.



- Client-side page rendering, covered a few weeks ago, uses what are essentially REST APIs designed for *private* rather than *public* use.

Principles of REST

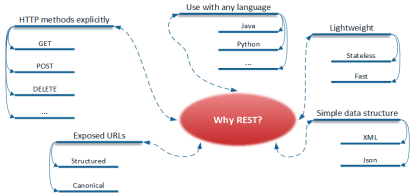
Representational State Transfer

- 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 centre of an entire industry. …That process honed my model down to a core set of principles, properties, and constraints that are now called REST.

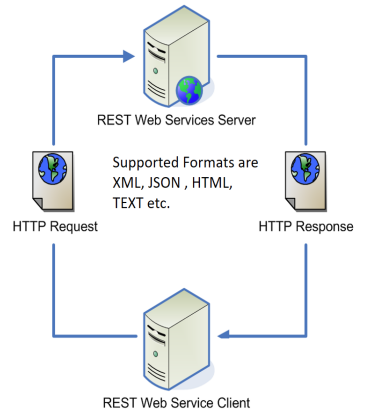
The six Characteristics of REST

- 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.
- He 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 and Facebook implement a pragmatic version of REST.



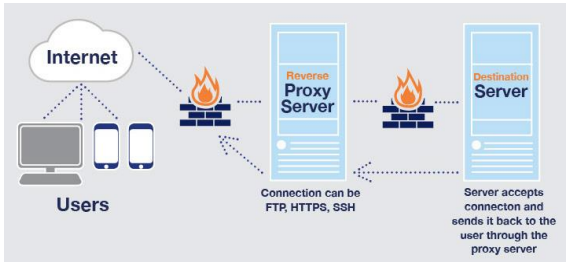
1. Client-server model

- 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.



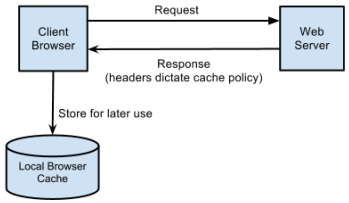
2. Layered System

- 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 whether 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/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.

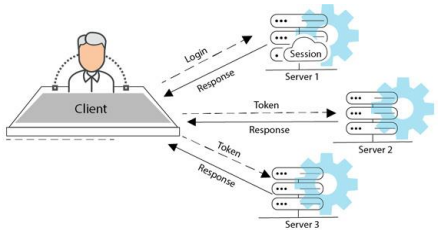


4. Code on Demand (optional)

- 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 a key property of the HTTP protocol, and most associated with REST APIs.
- Servers should not maintain any memory of prior transactions, and every request from clients 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 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 identifiers. 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.

Designing a REST API

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 HTTP methods: POST, GET, PUT/PATCH and DELETE.

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 APIs are usually organised so that each model in your database has an associated URL.
- This naturally satisfies the principle that when considered together, the URL and HTTP method should be self-descriptive of the action being performed.

REST URLs and Operations

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.

REST in Flask

- We can augment a Flask web application so that it provides a REST API but shares the database with the web application.
- Remember that the REST API provides a web interface to the backend 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 often implements a full MVC architecture itself, via the model's interface with the API.

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...

                api\
                        __init__.py
                        auth.py
                        models_api.py
                        token_api.pi
```

- We are going to add an `api` submodule 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.

Tokens

- Access to many APIs is controlled by tokens.
- The token API allows a logged-in user to generate a token that allows access to the API.
- It also provides a method to revoke the token, for the case that the API consumer believes their token has been compromised.

```
1  from flask import jsonify, g
2  from app import app, db
3  from app.api.auth import basic_auth, token_auth
4
5  @app.route('/api/tokens', methods=['POST'])
6  @basic_auth.login_required
7  def get_token():
8    token = g.current_user.get_token()
9    db.session.commit()
10   return jsonify({'token':token})
11
12 @app.route('/api/tokens', methods=['DELETE'])
13 @token_auth.login_required
14 def revoke_token():
15   g.current_user.revoke_token()
16   db.session.commit()
17   return '', 204  # no response body required
app/api/token_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.
- The `jsonify` method in Flask takes in a Python dictionary and produces a suitable response, so we add methods to our models to read from and write to dictionaries. Not all data should necessarily be writable. For example, in a `Student` model:

```
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'])
```

Adding a blueprint for the API

- We can separate out the API from the main webserver by using a blueprint.

```
from flask import Blueprint

main = Blueprint('main', __name__)
api = Blueprint('api', __name__)

from app import models, routes, token_api, student_api
```

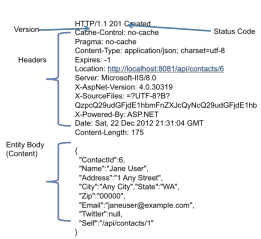
- This, in theory, allows us to spin up separate servers, one for the API and one for the web version of our application, while allowing both to share the same codebase.
- However, for the moment we can simply register the blueprint in the factory method:

```
from app.blueprints import main, api
flaskApp.register_blueprint(main)
flaskApp.register_blueprint(api, url_prefix="/api/v1")
```

Error messages in a REST API

- 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.

```
from flask import jsonify
from werkzeug.http import HTTP_STATUS_CODES

def error_response(status_code, message=None):
    payload={'error': HTTP_STATUS_CODES.get(status_code, 'Unknown Error')}
    if message:
        payload['message'] = message
    response = jsonify(payload)
    response.status_code = status_code
    return response

def bad_request(message):
    return error_response(400, message)
~
~
app/api/errors.py                                    1,1
```



Serving the routes: GET requests

- The route decorators allows us to specify parameters, which align with the parameter name in the method

```
@api.route('/students/<int:id>', methods=['GET'])
@token_auth.login_required
def get_student(id):
    if g.current_user != id:
        abort(403)

    data = Student.query.get_or_404(id).to_dict()
    return jsonify(data)
```

- When no parameter is specified for a GET request, the assumption is that the user wants the collection of all resources.
- In this case however, we don't want to enable to users to scrape each other's data and so we don't provide such a route for the `Student` model
- We would provide such a route for the `Project` model however.

Serving the routes: POST requests

- A POST request is used to create a new resource. For example, we could allow a new student to register as follows:

```
@api.route('/students', method=["POST"])
def register_student():
    data = request.get_json() or {}
    if 'id' not in data:
        return bad_request("Must include student number")

    student = Student.query.get(data['id'])
    if student is not None:
        return bad_request("Student already exists")

    student = Student()
    student.from_dict(data)
    db.session.commit()

    response = jsonify(student.to_dict())
    response.status_code = 201
    response.headers['Location'] = url_for('api.get_student', id=student.id)
    return response
```

- New resources should include their location in the response

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).

```
@api.route("/students/<int:id>", methods=["PUT"])
@token_auth.login_required
def update_student(id):
    if g.current_user(id) != id:
        abort(403)

    student = Student.query.get(id)
    if student is None:
        return bad_request("Student does not exist")

    data = request.get_json() or {}
    student.from_dict(data)
    return jsonify(student.to_dict())
```

Serving the routes: DELETE requests

- Finally delete methods allow the API users to delete objects from a collection.

```python
@api.route("/students/<int:id>", methods=["DELETE"])
@token_auth.login_required
def delete_student(id):
    if g.current_user(id) != id:
        abort(403)

    student = Student.query.get(id)
    if student is None:
        return bad_request("Student does not exist")

    db.session.delete(student)
    db.session.commit()
    return jsonify({})
```

Consuming a REST API with jQuery

- We have already seen how we can consume an arbitrary REST API in a webpage using AJAX and jQuery:

consume-rest.js

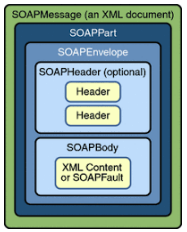
```javascript
$(document).ready(function () {
    $.ajax({
        url: "http://rest-service.guides.spring.io/greeting",
        success: function (data) {
            $("#id").append(data.id);
            $("#content").append(data.content);

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

});
```

- The advantage now of having an API is that we can also consume it from any programming environment that can make HTTP requests to the server, e.g. Android/IOs apps.

Other

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.



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