Edinburgh Napier
UNIVERSITY

ADVANCED WEB TECHNOLOGIES
SET09103
TOPIC #07

APIS, WEB SERVICES, & REST

Dr Simon Wells
s.wells@napier.ac.uk
http://www.simonwells.org

Edinburgh Napier UNIVERSITY

TL/DR

- Not just about throwing pages on a server.
- An API helps structure communications & interaction between your site & your user
- REST is one approach to designing & expressing the API that builds on the foundations of HTML

Edinburgh Napier UNIVERSITY

OVERVIEW

- As we start to build more feature rich sites - we generally have more data to structure, manipulate, store, & move around:
    - Data often sent to web-apps, and retrieved from web-apps.
    - Data often stored either in filesystem or in datastores.
    - Data often manipulated within a programming language.
- Last topic we saw how important it is to consider how data is **stored.**
- Now we'll consider how it is **presented** (but for machine rather than human consumption):
    - We'll take a data-oriented approach to providing information from a web-app — essentially: building data APIs using Flask.

DEFN: APIS

- *Contrast to a **user interface** (mediating communication between a person & a machine)*

- **A**pplication **P**rogramming **I**nterfaces:

- Term is used to denote a way to access information

- Ideally in a cohesive and structured form.

- Opens a computing system (program) to (automated) interaction from outside.

Edinburgh Napier UNIVERSITY

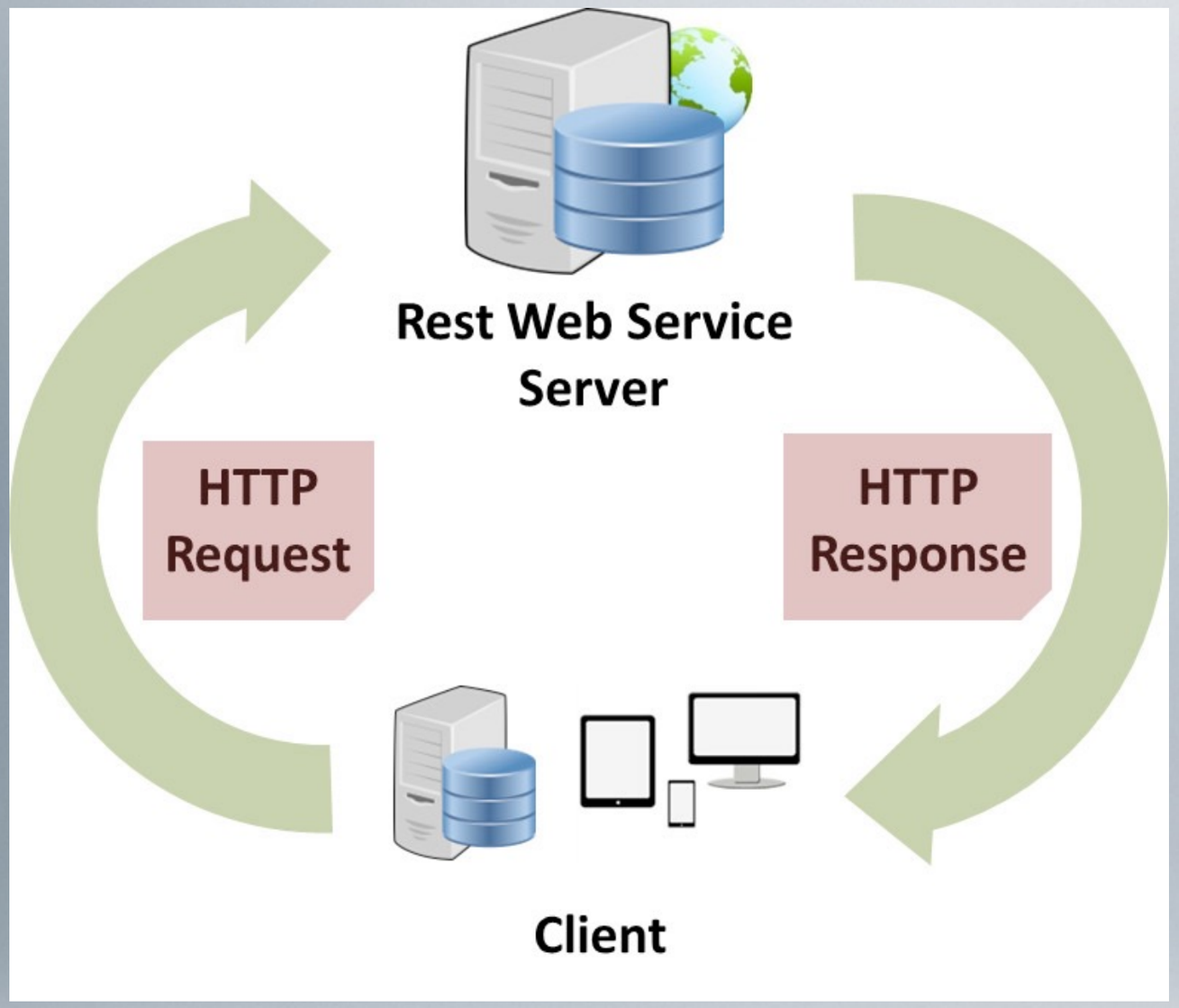
DEFN: WEB SERVICES

- A term used primarily to denote APIs made available using Web technologies.
- Can also indicate data APIs produced by one website and consumed by another.
- Note that there is a W3C standard for Web Services.
- But the term has been genericised over the years by pragmatic web developers (in contrast to standards setters).
  - Lesson (?): What people actually do is more important than the ideal (at least in terms of Web development, other arenas might vary).

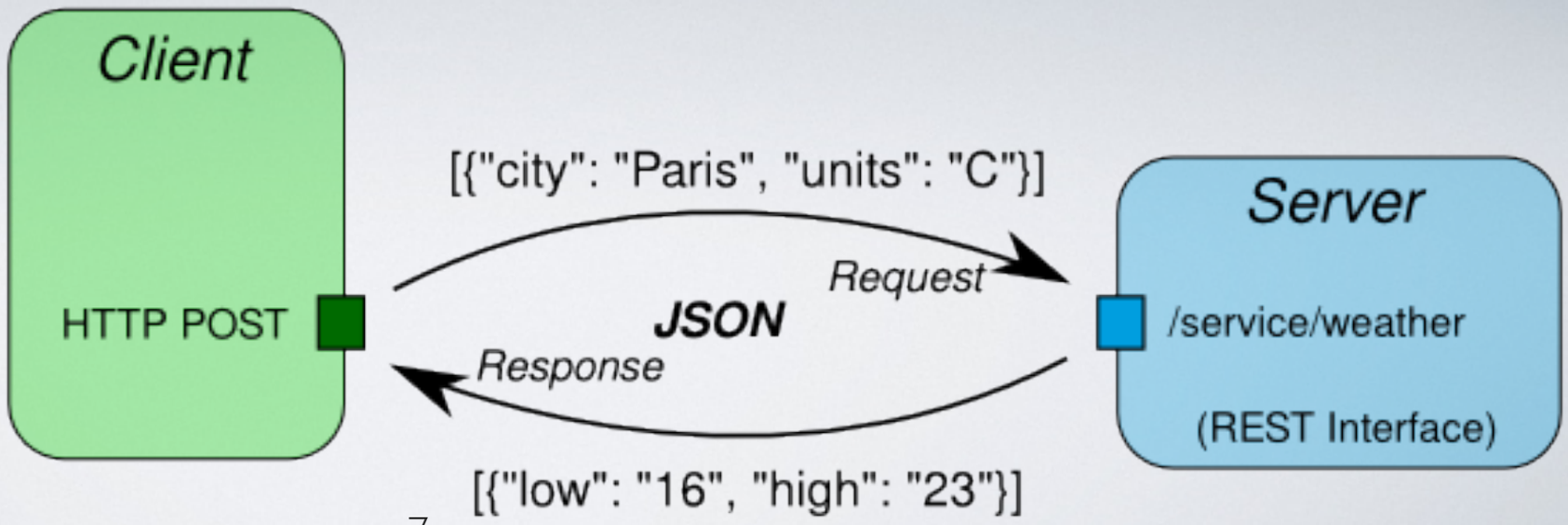
Edinburgh Napier UNIVERSITY

OVERVIEW OF WEB SERVICES

- We are concentrating on HTTP based services
- Where data is transported using the HTTP ...
  - ... & data is accessed using an API ...
  - ... & the data is represented & serialised using an interchange format:
    - **JSON** (or XML [, oy YAML, RDF, ATOM, …])
    - So **HTTP/JSON** (or JSON-RPC)
- There are alternative & related approaches, e.g. SOAP, WSDL, RPC, &c.



JSON / REST / HTTP



Edinburgh Napier UNIVERSITY

REST

- **RE**presentational **S**tate **T**ransfer (**REST**)
- A software **architectural style** for the WWW
- A set of coordinated constraints on designing components within a distributed hypermedia system
    - Aim for high-performance & maintainable architectures
- If a system conforms to the constraints of REST then can be termed *RESTful* - however many APIs only implement part of the constraints; hence my reference to HTTP/JSON RPC
- However, compared to alternatives like SOAP based web-services - there is no official standard for RESTful web APIs
    - SOAP is a protocol but REST is an architectural style (although it makes use of and is built upon standards (HTTP, URI, JSON, XML)

Edinburgh Napier UNIVERSITY

FORMAL REST CONSTRAINTS

Design according to the constraints:

1. Client Server
2. Stateless
3. Cacheable
4. Layered System
5. (optional) code on demand
6. Uniform Interface

To yield a site sharing the desired RESTful properties:

1. Performance
2. Scalability
3. Simple Interfaces
4. (Run-time) Modifiability of Interfaces
5. Visible Communication between components
6. Portability of components (move code with data)
7. Reliability (resistant to failure in the presence of failure)

(1OF6) CLIENT SERVER

- Clients & servers are separated by a uniform interface.
- Leads to **separation of concerns.**
- For example:
  - Clients not concerned with data storage:
    - Can you tell whether a site is storing data in a database, in memory, or on file?
    - internal responsibility of the server not the API.
    - portability is improved
  - Servers not concerned with UI or user state.
    - So servers can be simpler & more scalable.
    - NB. Often user state is usually the first casualty…

Edinburgh Napier UNIVERSITY

(2OF6) STATELESS

- No client context is stored on the server between requests
  - e.g. A good test is asking "can this interaction survive a server restart?"
- Each request must contain all the information necessary to service the request - session state is maintained by the client
  - Session state can be transferred by the server to another service, i.e. a database, that maintains a persistent state for a limited period & allows authentication
- Client sends requests when ready to make transition to a new state - If requests are outstanding then client is *in transition*
- Representation of each application state contains links that the client may use the next time it is ready to initiate a state transition

(3OF6) CACHEABLE

- Clients & intermediaries can **cache** responses - If the data hasn't changed since last time you requested it then why do more work?

- Responses must therefore (explicitly or implicitly) define themselves as cacheable or not to prevent reuse of stale data or excessive\inappropriate data transmission

- If implemented and managed well then caching can eliminate some client-server interactions - improving scalability and performance

(4OF6) LAYERED

- Client cannot tell whether connected directly to the end server or an intermediary.
- Intermediaries can improve system scalability, e.g.
  - load balancing,
  - shared caches,
  - enforcing security policies.

Edinburgh Napier UNIVERSITY

(5OF6) CODE ON DEMAND

- This is the only OPTIONAL constraint in a REST architecture (but is one of the most interesting).
- Servers can temporarily extend the functionality of the client by transferring executable code to it.
- By sending JS alongside the data, we can give the client the functionality it needs to handle or otherwise manipulate that data.
- Enables the client to do much more.

(6OF6) UNIFORM INTERFACE

- A fundamental of the design of a RESTful service:
- Aim is to simplify and decouple the architecture so that every part can evolve independently:
    - Identification of resources - URIs are conceptually separate from the representation returned to the client, e.g. data may be returned in HTML, XML, JSON
    - Manipulation of resource through representations of resources - If client hold representation (+necessary metadata) then has enough information to modify or delete resource
    - Self descriptive messages - Message must carry enough information to describe how it should be processed, i.e. parser may be invoked depending on specified internet media type
    - HATEOAS - Except for simple fixed entry points, clients make state transitions only through actions that are identified within hypermedia by the server, e.g. hyperlinks. Don't assume a particular action is available unless specified in the hypermedia related to that resource. Reduce out-of-band information. Decouple client & server.

Edinburgh Napier UNIVERSITY

REST & THE WEB

- RESTful systems typically use HTTP
- Use the same verbs (GET, POST, PUT, DELETE, &c.) - that we have seen in labs & used (perhaps unknowingly until now) in our browsers - to retrieve web pages & send data to servers
- Web pages are (collections of) resources - identified by a URL, e.g.
  - napier.ac.uk/students/ - would indicate a collection of student resources
  - napier.ac.uk/students/09321234 - would indicate a single student resource
- Word & naming are important in RESTful API design
- However, it is easy to tie yourself up in knots trying to create a RESTful design.
- Difficult to adhere to all of the constraints and satisfy desires of current Web users
- Better to take a pragmatic approach that is inspired by RESTful ideals: **HTTP/JSON RPC**

Edinburgh Napier UNIVERSITY

LAYERED APPROACH TOWARDS REST

- Four layers:
  - **Level 0** - HTTP as transport system
  - **Level 1** - Resources
  - **Level 2** - HTTP Verbs
  - **Level 3** - Hypermedia controls
- Start with level 0. Stop at any point. Elaborate as required.

LEVEL 0

- Use HTTP as the transport system for remote interactions.
- "tunnel" your data & functionality through HTTP.
- Content can be expressed in any language (we've already looked at data transport languages in an earlier topic):
  - **JSON**, XML, Key:Value Pairs, YAML, Custom formats,…

Edinburgh Napier UNIVERSITY

LEVEL 0

- Service exposes an endpoint (some URL), e.g.
  - /make-an-appointment
- Client calls endpoint and passes a document describing requirements
- Service responds appropriately (e.g. open appointments)
- Client then requests a specific appointment
- Services responds with status (success/failure/?)
- **Level 0: Just using HTTP as transport to moves messages between client and server**

LEVEL 1

- In level 0 we just used a single end-point for all communications
- Now we consider each end-point as a **resource**
  - Make calls to specific resources
  - Multiple resources so multiple end-points
- e.g. each lecturer is a resource **/lecturer/simon**
- each resource has availability **/lecturer/simon/calendar**
- can request a specific slots **/lecturer/simon/calendar/1234**

Edinburgh Napier UNIVERSITY

HTTP + RESOURCES

- System has a base URL, e.g. napier.ac.uk/
- An internet media type for data - usually JSON (but can be any other valid internet media type, e.g. XML, atom, microformats, images, &c.)
    - Aren't restricted to a single data transport - can support multiple
- Standard HTTP methods - just POST and GET for now (could all just be GETs)
- Instead of single end-point we **design** a URL hierarchy that is appropriate to the problem (remind anyone of other kinds of software development?)
    - Use singular noun for individuals and plural for collections, e.g
        - GET /books/
        - GET /books/<?ISBN>
        - GET /books/<?year>
- **Level 1: Using HTTP to move messages & also to organise the places those messages go to**

LEVEL 2

- Until now still just sending messages through HTTP. Meaning of message entirely dependent on server deciding what to do with it
    - Get some differentiation by using different end-points
- Why not differentiate messages? i.e. same message can be interpreted in different ways using HTTP verbs
    - i.e. If we GET an end-point, then treat this differently to if we POST, or PUT, or DELETE, or PATCH, &c.
- When responding, why not indicate the status of the interaction?
    - i.e. that there was success (200) or failure (304/404/501)?
- Now we are no longer just tunnelling messages through HTTP - still using HTTP to transport messages but also using HTTP to convey how those messages should be used.

Resource	GET	PUT	POST	DELETE
Collection URI, such as `http://api.example.com/v1/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.[10]	**Delete** the entire collection.
Element URI, such as `http://api.example.com/v1/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 in it.[10]	**Delete** the addressed member of the collection.

IDEMPOTENCY & NULLIPOTENCY

Edinburgh Napier UNIVERSITY

- PUT & POST differ because PUT is considered to be **idempotent**
  - Repeating the same operation will produce the same result no matter how many times it is repeated
    - e.g. PUT a resource in a location - the state should be identical afterwards but POST a resource to a location & you should get a different resource each time
- DELETE is also considered to be idempotent - deleting a resource a second time won't make it more deleted - same result
- GET is considered to be **nullipotent** - calling GET has no side-effects - retrieving a resource should not change it (GET is frequently considered safe)

Edinburgh Napier UNIVERSITY

HTTP+RESOURCES+VERBS & STATUSES

- Return appropriate status alongside response - Otherwise our response is still tunnelling messages through HTTP
- Verbs & Status codes are complimentary ways to communicate meta-data about the message
- i.e. Don't need to read the message to know how to treat it
- **LEVEL 2: Using HTTP to transport messages, to organise resources, & to interact with resources**

Edinburgh Napier UNIVERSITY

LEVEL 3

- **HATEOAS -** Hypertext As The Engine Of Application State
- This is the bridge from knowing about resources to knowing what to do with them. How do we find out what we can do with a resource? - The API should tell us
- How does the API tell us? - through hyperlinks (this is still a hypertext system after all)
- In your response to the caller, provide links that are valid next responses, e.g. if booking an appointment response shouldn't just tell us which slots are available, but provide the links for booking those slots
    - Using hypertext to specify valid transitions from the current state to succeeding states
- Really powerful - can generate/change our API on the fly by just responding with new URLS during any given call
- At the extreme: site becomes self-documenting and dynamic. A call to the root URL will tell you what you can do with it (**Uniform Interface**) so the system can change on the fly.
- Users however need to explore the API & can't rely on it not changing in the future.

Edinburgh Napier UNIVERSITY

REST: ARE WE THERE YET?

- Officially levels 0-3 only get us to the starting point for a full RESTful application
  - Level 3 is a pre-condition for REST
  - But show the progression from simple remote procedure call to hypermedia application using existing web technologies:
    - Level 1 - handle complexity through divide & conquer. Break service in to multiple resources
    - Level 2 - Handle similar situations in similar way using verbs & status codes
    - Level 3 - Discoverability & self-documentation
- Web users & developers are pragmatic
  - Have tended towards simplicity & robustness

HTTP & WEB SERVICES

- A program, i.e. a Web Server
- Provides resources at locations (URLs)
    - Uniform Resource Locators
- Use HTTP Verbs to indicate what to do with the resource at a given location
    - GET, PUT, POST, HEAD, DELETE, PATCH, OPTIONS, …
- Use response codes to indicate status/ result of the call
    - 200, 201, 304, 401, 404, 500, …
- API: A set of end-points (i.e. web addresses) made available on a server using HTTP (i.e. a web server) that can be called by a client (i.e. your app) & which accept &/ or return data

/api/join
/api/user/account
/api/user/account/confirm/<key>
/api/user/profile
/api/notifications

Edinburgh Napier UNIVERSITY

REST APPLIED TO W/S

- Simpler than alternatives (but no 'official' standard)
- Base URI: e.g. http://napier.ac.uk/api/
- Media type: e.g. JSON (could also be XML, ATOM, microformates, &c.)
- Standard HTTP verbs (GET (nullipotent), PUT (idempotent), POST, DELETE (idempotent))
- Hypertext links
- to reference state & related resources

Resource	GET	PUT	POST	DELETE
Collection http://napier.ac.uk/api/students/	List the URIs (other details) of the collections members	Replace the entire collection with another collection	Create a new entry in the collection (automatically assign a URI	Delete the entire collection
Element http://napier.ac.uk/api/students/student9707112	Retrieve a representation of the element using an appropriate media type	Replace the element of the collection or create it (if doesn't exist)	Not generally used	Delete the element from the collection

```
from flask import Flask, request, jsonify
import json

app = Flask(__name__)

status_data = []

@app.route("/", methods=['GET'])
def get_status():
    print(status_data)
    return (jsonify(status_data))

@app.route("/", methods=['POST'])
def post_status():
    content = request.get_json()
    if(content):
        status_data.append(content)
    return (jsonify(status_data))
```

```
@app.route("/", methods=['PUT'])
def put_status():
    content = request.get_json()
    if(content):
        index = content['index']
        status = content['status']
        status_data[index] = status
    return(jsonify(status_data))

@app.route("/", methods=['DELETE'])
def delete_status():
    content = request.get_json(silent=True)
    status_data.clear()
    return (jsonify(status_data))
```

Edinburgh Napier UNIVERSITY

WRAPPING UP

- We've covered aspects of API design for the Web and the development of Web Services (Data APIs for machines to consume).
- We've taken a pragmatic HTTP/JSON RPC approach (inspired heavily by the ideals of RESTful design.
- Next topic:

  Privacy by design & related security engineering topics.