ARCHITECTING DYNAMIC SITES USING REST & CRUD

Dr Simon Wells

s.wells@napier.ac.uk

http://www.simonwells.org

TL/DR

There is a lot more to designing effective dynamic web sites than just generating HTML on request

AIMS

- By the end of this topic you will be able to:
  - Use the Express.js & Pug packages to implement your own dynamic web apps
  - Design your dynamic web apps
  - Use REST
  - Develop APIs
  - Take into account RESTful principles & CRUD architectures

TOPIC OVERVIEW

- Now we are getting into the thick of it.
- As we've hinted at earlier, web technologies are about much more than just serving up static web pages
- Dynamic sites aren't just "*web-pages on demand*"
- They are information architectures that have theoretical underpinnings which fully exploit the facilities offered by HTTP

PART 1:
DYNAMIC SITES WITH EXPRESS THEN A QUICK DIVERSION TO JSON TOWN

OVERVIEW

- We don't want to build every project from first principles
- Sometimes we want something helps manage the process
- So we use libraries & packages that already implement some of the process for us
- Different packages approach this *support* issue from different perspectives - i.e. there can be great variation in how different packages get you to the same destination
- We'll look at **express.js** - a reasonably popular package for implementing server-side web functionality

Edinburgh Napier UNIVERSITY

BASIC HTTP TO EXPRESS

- In the last topic we got used to the idea of dynamic sites including pages that are generated on demand
- Whilst we can do everything using the basic HTTP package this is doing things the hard way
- We want feature rich & maintainable dynamic sites & we don't want to reinvent the wheel for every site
- So let's use a package that helps us get further, faster.
- We are going to use Express.js to build our pages & they are going to be glorious (eventually…)

EXPRESS.JS

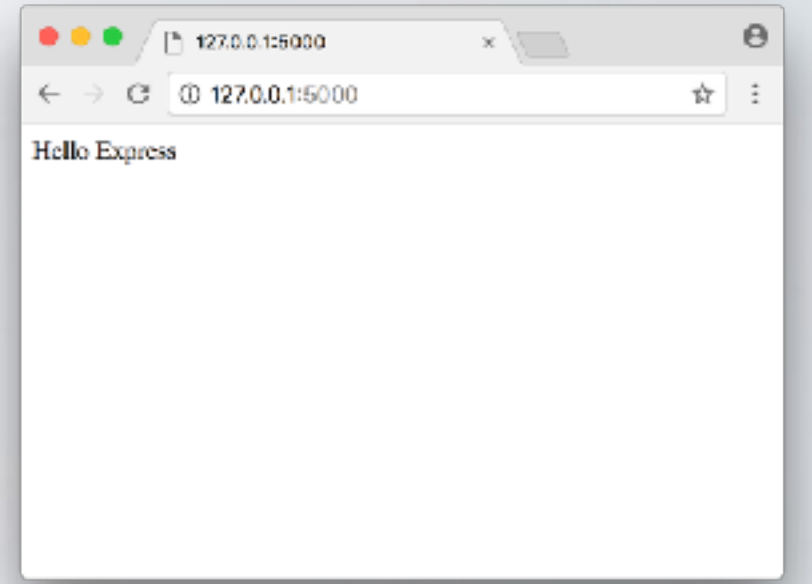
- A web application framework that has been around forever (at least in terms of Node history)
- Can be used to create web-apps (HTML for the browser) & APIs (JSON for everything)
- Open-source (MIT), reliable, stable, and regularly updated. Installed using NPM.
- Has some nice features:
  - It's fast
  - It let's you solve your web-app development problems your way - rather than telling you how to do things
  - Doesn't include lots of stuff we don't need - so need to install extra packages if you want specific features

GENERAL PROCESS

- Build a set of routes that correspond to web addresses (URLs) - write responses to requests to specific URLs
- Use a templating engine to guide the generation of HTML
  - Don’t want to write build our HTML pages from complete scratch for every page - it’s good to have some structure in place

"HELLO EXPRESS"

- What does an express.js web-app look like?
- Note: *route, request & response, port & host*




```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
   res.send('Hello Express');
})

var server = app.listen(5000, "127.0.0.1", function () {
   var host = server.address().address
   var port = server.address().port

   console.log("Listening on http://%s:%s", host, port)
})
```

COMPARE EXPRESS & RAW NODE HTTP

Node+http package

```
const { createServer } = require("http");
const PORT = process.env.PORT || 5000;
const server = createServer();

server.on("request", (request , response) => {
  response.end("Hello, world!");
});

server.listen(PORT, () => {
  console.log('starting server at port ${PORT}');
});
```

express.js

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
   res.send('Hello Express');
})

var server = app.listen(5000, "127.0.0.1", function () {
   var host = server.address().address
   var port = server.address().port

   console.log("Listening on http://%s:%s", host, port)
})
```

ROUTING & HTTP METHODS

- Again, neater than before, now use HTTP methods as functions on app object
- e.g. app.get

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
   res.send('A GET request to the root resource');
})

app.post('/', function (req, res) {
   res.send('A POST request to the root resource');
})

app.head('/', function (req, res) {
   res.send('A HEAD request to the root resource');
})

var server = app.listen(5000, "127.0.0.1", function () {
   var host = server.address().address
   var port = server.address().port

   console.log("Listening on http://%s:%s", host, port)
})
```

SUPPORT FOR MANY HTTP METHODS

- checkout
- copy
- delete
- get
- head
- lock
- merge
- mkactivity
- mkcol
- move
- m-search
- notify
- options
- patch
- post
- purge
- put
- report
- search
- subscribe
- trace
- unlock
- unsubscribe

TEMPLATING ENGINES

- So far, only returned either raw HTML or plain text from express
- Express supports templating engines (Most web-app frameworks do)
- Templating engines enable you to write the structure for a web-age with parts *to be filled in later*
    - Some use HTML but special instructions to indicate which parts of the HTML are part of a *template* i.e. are bits that can be competed by supplying extra data
    - Some use other markup languages to describe the structure of the HTML to be generated (but don't actually use HTML themselves)
- The template is not a complete HTML page - it's only completed when data is supplied, for example, data generated from code, from a file, or from a datastore

TEMPLATE ADVANTAGES

- Can write a page *archetype* - a template that captures design across multiple pages
- Many templating systems also support inheritance
  - So you can define the HTML for, e.g. headers, footers, asides, main content, etc (however your design breaks down logically)
  - Then incorporate multiple templates into the final page
  - Usually completed using data
- Multiple pages can use same template - just complete with different data

PUG

- Pug templates are compiled to HTML (& other targets)
- Is a programming language - supports conditions, loops, includes & mixins
- HTML is rendered based on PUG language + user input or data
- HTML is concerned mostly with *tags*, *attributes*, and *text* so Pug focusses heavily on this

```
doctype html
html(lang='en')
 head
   title Pug
 body
   h1 Pug Examples
   div.container
     p Cool Pug example!
```

```
<!DOCTYPE html>
<html lang="en">
 <head>
   <title>Pug</title>
 </head>
 <body>
   <h1>Pug Examples</h1>
   <div class="container">
     <p>Cool Pug example!</p>
   </div>
 </body>
</html>
```

EXPRESS + PUG

- HTTP web-app framework with templating
- Installed using NPM
- Use *express-generator* tool to create a skeleton web-app
- This should run as is (but won't do much)
- Contains multiple files including pug files which are enough to get started building our own site

```
.
├── app.js
├── bin
|   └── www
├── package.json
├── public
|   ├── images
|   ├── javascripts
|   └── stylesheets
|       └── style.css
├── routes
|   ├── index.js
|   └── users.js
└── views
    ├── error.pug
    ├── index.pug
    └── layout.pug
```

GENERATED PUG FILES

- index.pug "*inherits*" from layout.pug
- layout:
  - sets type of document - can support others
  - Similarities to HTML but:
    - removes angle brackets & paired tags
    - uses indentation to indicate encapsulation

layout.pug

```
doctype html
html
  head
    title= title
    link(rel='stylesheet',
      href='/stylesheets/style.css')
  body
    block content
```

index.pug

```
extends layout

block content
  h1= title
  p Welcome to #{title}
```

Edinburgh Napier UNIVERSITY

USING PUG TEMPLATES FROM EXPRESS

- From our express app we can send data to a pug template
  - e.g. in both cases we are using the index.pug template
  - But passing a different title to each

```
var express = require('express');
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

router.get('/hello',
  function(req, res, next{
    res.render('index',
      { title: 'The Hello Route' });
});

module.exports = router;
```

JSON

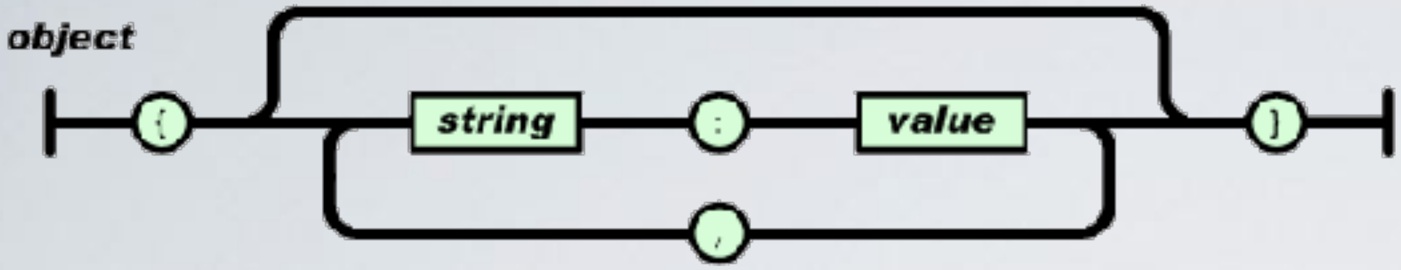
- Now need to take a slight diversion to fill in a gap
- We've mentioned JSON very briefly in topic 3 when we introduced JavaScript
- But as we start to build more feature rich sites - we have more data to structure, manipulate, and store
- Although originally for serialising JS objects, JSON is now *de facto* standard for describing data
- Data often sent to web-apps, and retrieved from web-apps in JSON format
- Data often stored as JSON either in filesystem or in datastores
- Data often manipulated within JS as JSON
- So we should probably take a closer look at it…

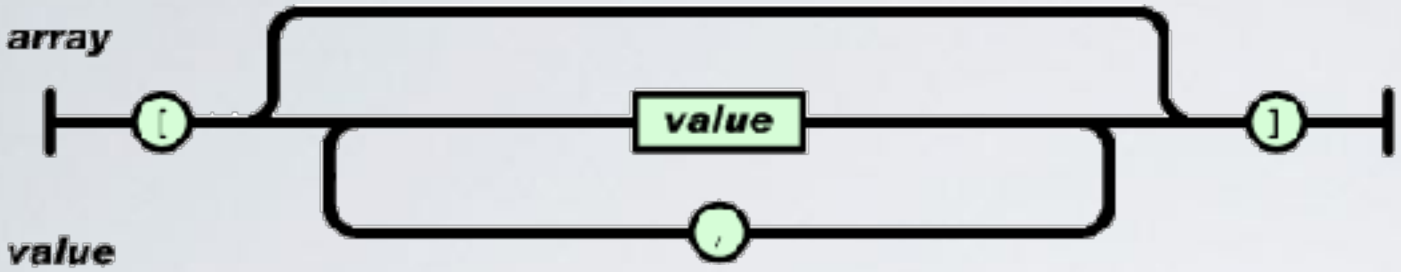
JSON LANGUAGE

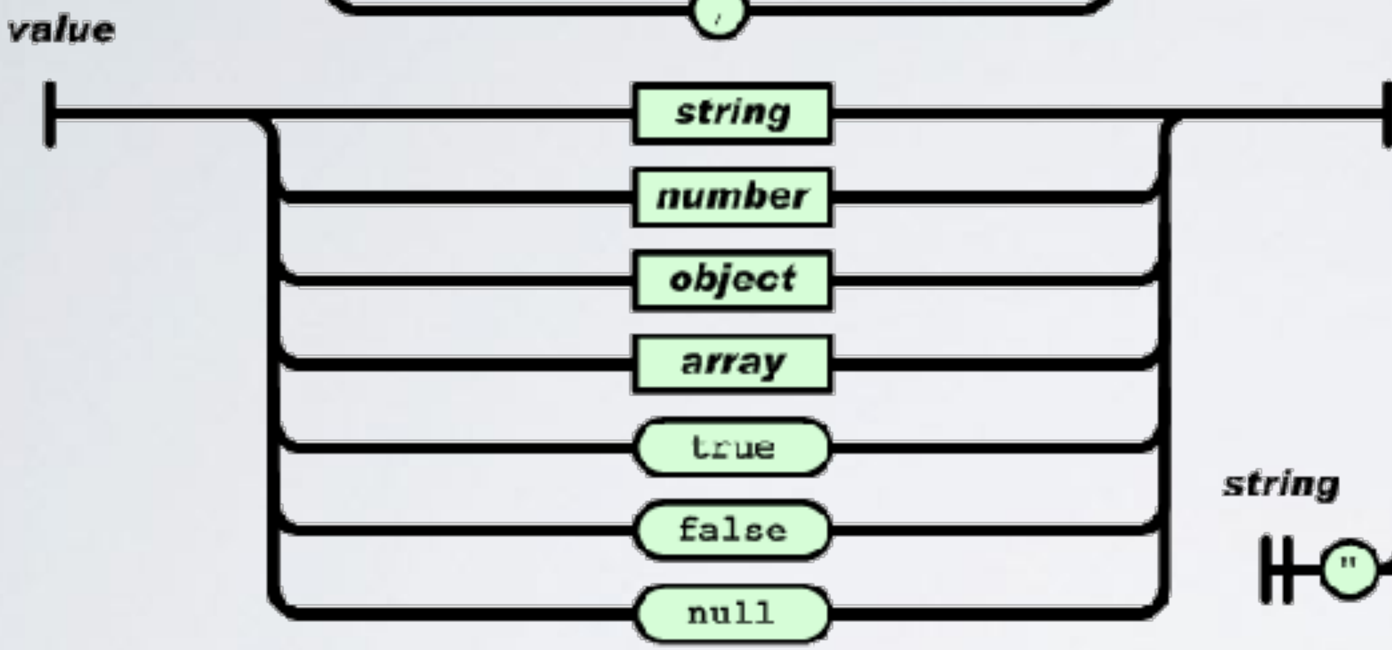
- Plain text format - can create & edit using your text editor
- Is iteratively constructed starting with either an object {} or an array []
- Array can store a comma separated list of values
- Object can store a comma separated list of key:value pairs
- Values are objects, arrays, strings, numbers, booleans, null
- An easier way to visualise this is with railroad diagrams

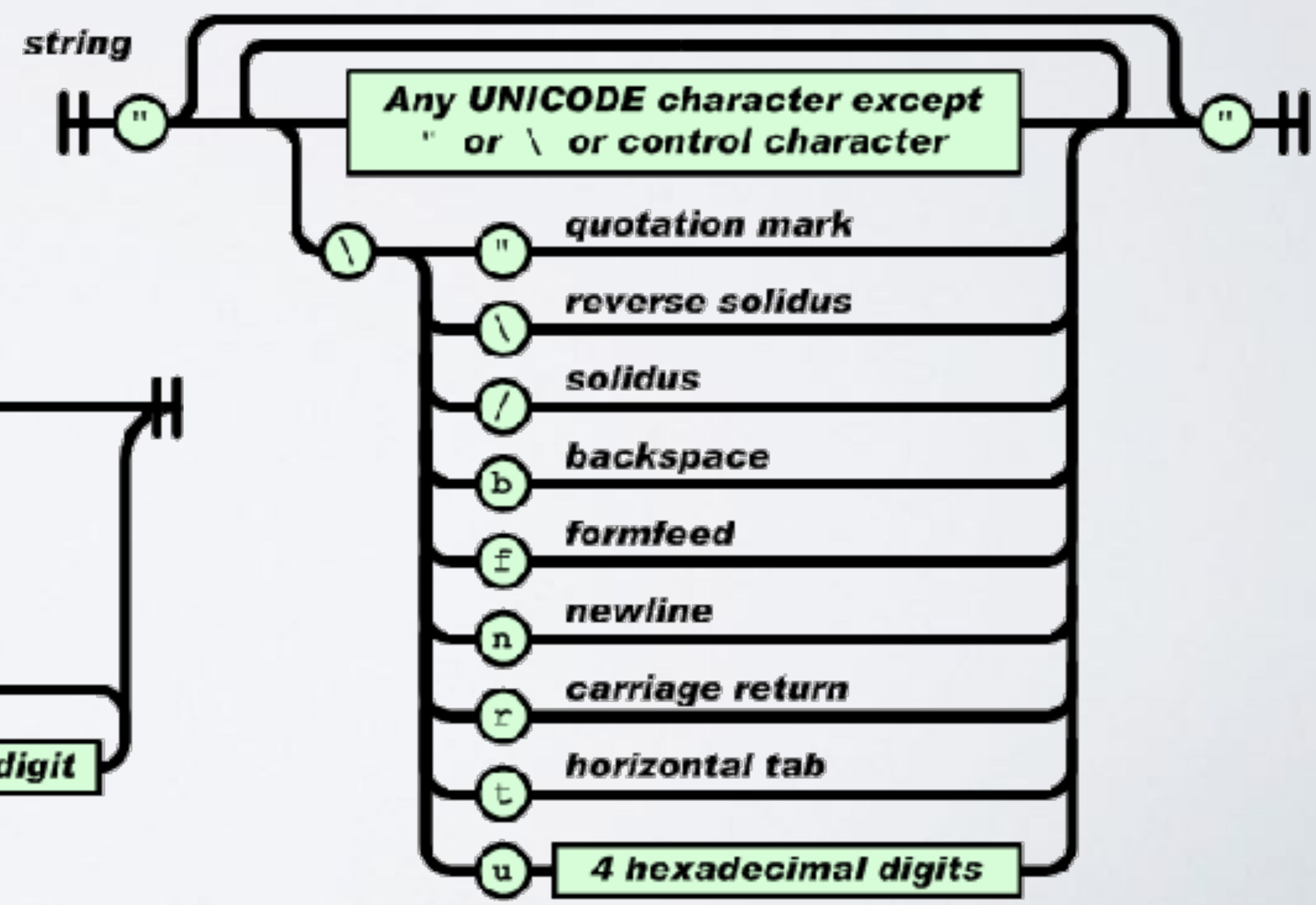
Edinburgh Napier
UNIVERSITY

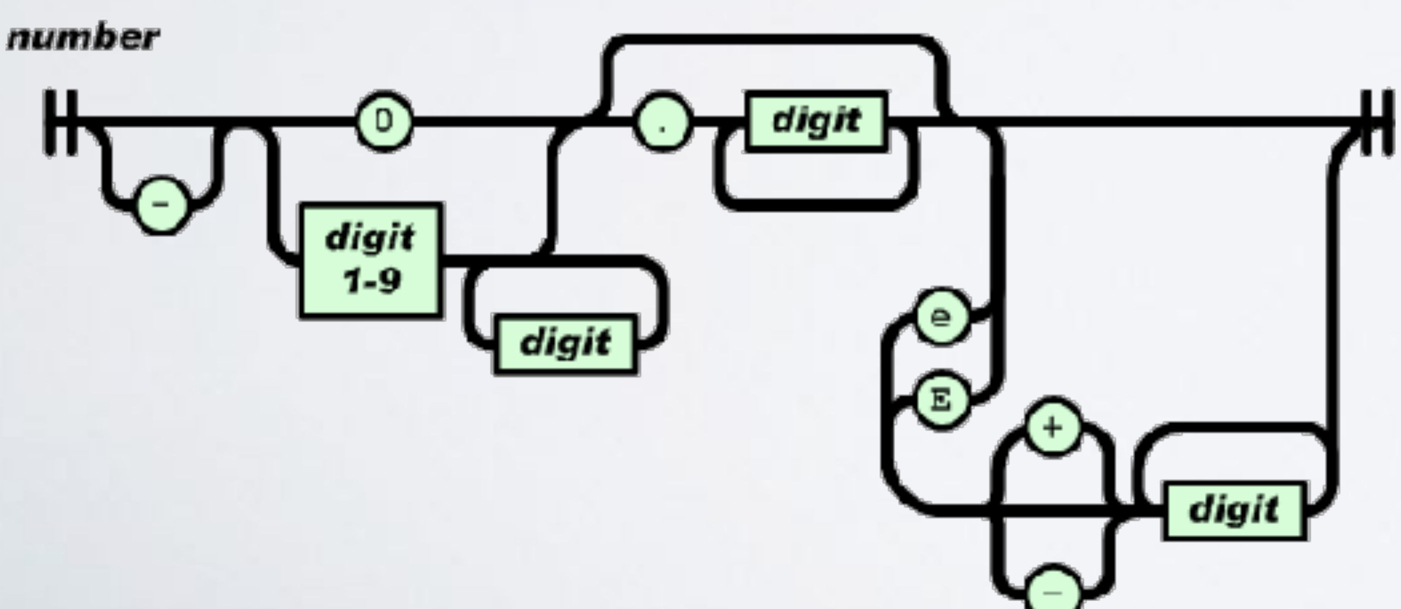
JSON RAILROAD DIAGRAMS





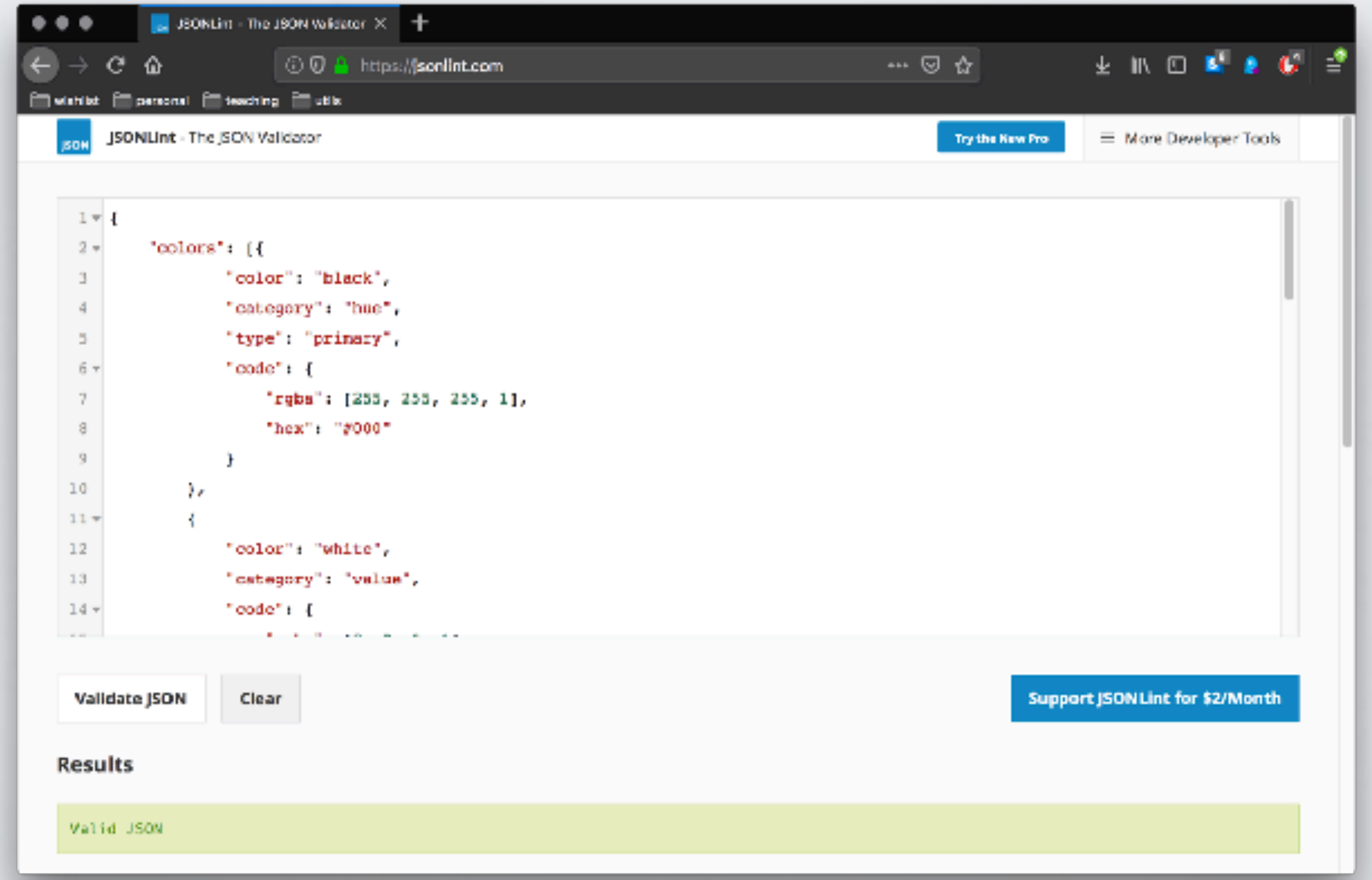






JSON LINT

- A useful online tool for quickly editing & *validating* JSON files
- Edit then copy/paste to text file and save



WHY?

- If we're going to build APIs, or really exploit JavaScript, or share data with other APIs (or retrieve data from them) then we need to use JSON
- There are other data transports, e.g. XML, but JSON occupies a *sweet spot*
  - not too complex,
  - tooling is lightweight
  - human readable
- Yes there are drawbacks to JSON, but the positives make it an easy tool to choose and use.

QUESTIONS ???

PART 2:

APIS, REST, & CRUD

OVERVIEW

- Developing a dynamic site isn't **just** about executing code on the server when a request comes in
- There are principled approaches to designing dynamic sites
- Intimately connected with design of HTTP APIs, covering at least:
    - The structure of the hierarchy of URL that make up the addresses of the site
    - The range of HTTP methods that can be applied to each URL
    - The data associated with each URL & the effect on the data of applying each HTTP method.
- Dynamic sites need not just return HTML (JSON, XML or any other **mediatype**)

APIS & THE WEB

- **A**pplication **P**rogramming **I**nterfaces
  - A user interface (but for different groups of users)
  - *Generally:* A way for communications to occur between software, i.e.
    - Web Server <——> Browser (HTML/CSS/JS)
    - Web Server <——> Mobile App (JSON\XML)
- An API can return data formatted in different ways depending upon the request

WHY?

- Build platforms rather than sites
  - People do innovative things with what you provide
  - Contributes back more data
  - Interesting applications attract more users
    1. You don't have time to develop everything for everyone
    2. Restricting what users can do can alienate them

Which organisations have developed a platform rather than a site?

GOOGLE, NETFLIX, FLICKR, GITHUB, TWITTER, FACEBOOK, OTHERS….?

HOW TO BUILD GREAT SITES

- Even if you're not sharing a public API & building a platform

API design is important

- A good, well-structured, easy to use API can help your site to be:

scalable, extendable, easy to develop for, maintainable, robust - **all the things that we should aim for**

- But a nuanced, ongoing debate about how to achieve this - no official guidelines

API DESIGN APPROACHES

- Two main approaches:
  - **CRUD** - **C**reate, **R**ead, **U**pdate, **D**elete
  - **REST** - **RE**presentation **S**tate **T**ransfer
- Both deal mainly with data
  - How we interact with it & how it should be structured
  - CRUD is more of a general pattern or interaction cycle for how software interacts with data
  - REST is more focussed on building hypertext systems (but not specifically web systems)

CRUD

- Built around the four basic functions used for interacting with persistent storage

Consider the ways that you interact with a file - what can you do with it?

- Turns out that interaction patterns occur all over the place - the way that we interact with files is mirrored in the way that we interact with databases, and in the way that we interact with websites
- NB. CRUD is not the whole story but covers the core interactions:
  - We create data, retrieve it, update it, and destroy it
    - Notice that the terminology can vary a little read/retrieve, delete/destroy - but the semantics are the same
    - Obviously we also do a lot more with data than just this

CRUD AS USER INTERACTION

- Those four CRUD functions can also be used to characterise the basic user interaction conventions:
  - Creating, reading, updating, and deleting facilitate the viewing, searching, and alteration of information

Aren't these the kind of interactions that we expect a dynamic web-app to provide?

CRUDS RELATIONSHIP TO DATABASE INTERACTIONS

- The Structured Query Language (SQL) has been pervasive in influencing the design of dynamic web-applications:
  - We INSERT, SELECT, UPDATE, & DELETE our data to/from a db
  - Many web-apps are (or were) backed by SQL databases
  - So ideas from SQL adapted & applied to web design

CRUD / SQL / HTTP

- Notice the mapping between CRUD operations, SQL commands, and HTTP methods
- Note that there is no clear mapping between operations and HTTP methods
  - One reason why CRUD is more of a *de facto* pattern than a carefully designed architectural style
  - Can't necessarily reason about the effect of any given HTTP method in terms of the underlying CRUD operation, e.g. will an HTTP PUT create or update the resource it's applied to?

Operation	SQL	HTTP
Create	INSERT	PUT/POST
Read	SELECT	GET
Update	UPDATE	PUT/POST/PATCH
Delete	DELETE	DELETE

URLS, HTTP, & CRUD

- For many dynamic sites we can design our URL structure to reflect the needs of the underlying CRUD pattern
  - Think of each URL as a document or even as a data-item
  - The CRUD operations manipulate that data via applying HTTP methods to it's URL

EXERCISE

- In pairs, consider a social media site of your choice,
  - Determine the kinds of data that the site needs to store
  - Design a simple URL hierarchy for organising the site
  - For each URL, list the HTTP methods that need to be supported
  - For each URL & method pair, identify the kinds of CRUD operations that will be applied as a result of calling that URL/Method pair

REST

- **RE**presentational **S**tate **T**ransfer (**REST**)
- A software **architectural style** for the WWW
- Developed by Roy Fielding (architect of HTTP1.1 [96-99] with Berners-Lee) in his Ph.D Thesis (2000 "Architectural Styles and the Design of Network-based Software Architectures" - chapter 5 specifically (*don't read the entire thing unless you're really interested)*
- A set of coordinated constraints on the design of components within a distributed hypermedia system

With an aim towards 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 - ***buzzwordy*** (everything is REST right now, even if it's **not**)

ROY FIELDING'S DESCRIPTION OF REST

HINT: *THE FINAL PARAGRAPH IS THE IMPORTANT PART*

REST's client–server separation of concerns simplifies component implementation, reduces the complexity of connector semantics, improves the effectiveness of performance tuning, and increases the scalability of pure server components. Layered system constraints allow intermediaries—proxies, gateways, and firewalls—to be introduced at various points in the communication without changing the interfaces between components, thus allowing them to assist in communication translation or improve performance via large-scale, shared caching. REST enables intermediate processing by constraining messages to be self-descriptive: interaction is stateless between requests, standard methods and media types are used to indicate semantics and exchange information, and responses explicitly indicate cacheability.

Fielding (2000)

Chapter 5 of "Architectural Styles and the Design of Network-based Software Architectures"

REST VS CRUD

"REST enables intermediate processing by constraining messages to be self-descriptive: interaction is stateless between requests, standard methods and media types are used to indicate semantics and exchange information, and responses explicitly indicate cacheability."

- These bits are similar to CRUD approaches to HTTP APIs:
  - Stateless between requests - HTTP
  - standard methods - HTTP
  - media types - HTML, JSON, & others
- Innovations are many but include: self-descriptive, indication of semantics
- It is the innovative parts that are frequently not implemented in real world REST systems so systems described as *RESTful* are often a variation on CRUD - not terrible, just not perfect.

ARCHITECTURAL PROPERTIES

- A REST*ful* approach **affects** (ideally positively) a range of identifiable properties of distributed hypermedia systems:
  - (Perceived) **Performance** - component interactions can be a dominant factor in user perception of system performance and network efficiency
  - **Scalability** - Support large numbers of components & interactions between them
- Simple interfaces
- (Run-time) **modifiability** of interfaces
- Visibility of communication between components
- **Portability** of components (move code with data)
- **Reliability** - system should be resistant to failure even if components, connectors, or data fail in some way

ARCHITECTURAL CONSTRAINTS

- A RESTful approach applies constraints to the interface:
  - **Client/Server** - separation of concerns
  - **Stateless** - can it survive a server restart?
  - **Cacheable** - if data hasn't changed since last request, why recalculate?
  - **Layered System** - client can't tell if connected to end server or intermediary
  - (optional) **code on demand** - temporarily extend client functionality (e.g. JS)
  - **Uniform Interface** - URI identifies resource that can be manipulated (verbs) & represented (mediatype) in different ways.

Let's look at each constraint over the next few slides…

CLIENT/SERVER

- Underpins the idea that REST is a distributed approach
- A client/server architecture forces this separation:
  - Server provides services & client is a consumer of such
- Each service can do multiple things and listens for requests
- Requests made by a consumer are accepted or rejected by the service
- NB. Basically describing what an HTTP server does

STATELESSNESS

- A guiding principle that affects the kinds of services that a server can offer
- For a service to be stateless it must be
  - Initiated by the consumer (using a request)
  - Request must contain all of the information necessary to know how to respond to it
- Again, this is essentially what HTTP requires (unless you're getting around things with URL parameters and cookies & clever server hacks)

CACHEING

- Response to requests must be labelled by the server as cacheable or otherwise

- If cacheable then can be more efficient:

  - i.e. don't need to actually send same request all the way to the server a second time to get the same response if we have the first response cached at some intermediate point

LAYERED

- Principle that underpins the scalability of REST interfaces
- Multiple layers used to grow and expand the interface
  - i.e. New functionality can be added, e.g. middleware & servers without impacting any pre-existing functionality between client and server

CODE ON DEMAND

- The only optional constraint - even though many sites that describe themselves as REST often don't actually respect all the constraints
- Enable logic (functionality) on the client to be separate from functionality on the server
- Client functionality can be updated independently of server
- Consider code-on-demand to be equivalent to downloading JS (or any other language/plugin) that can lead to client side functionality
  - This was rare when REST was defined but is now commonplace
- Code moves from server to the client on demand & the server maintains control of where it's code is run

UNIFORM INTERFACE

- REST APIs are meant to be *discoverable*
- To be discoverable they are *self-descriptive* - provide information about themselves, how they are used and where you can get to from the current URL (e.g. Hyperlinks to other URLs)
- Uniform Interface defines a contract for interacting with the system:
    - i.e. Fixed entry point URLs followed by transitions through states identified by hyperlinks to other URLs
    - Leads to Hypermedia As The Engine of Application State (HATEOAS)
    - Given the entry point you should be able to discover the rest of the app even if things are changing on the fly behind the URL - The API (URLS+Methods+mediatypes) define the interface, not how the server is implemented behind the scenes
- In OOP terms this is decoupling of interface/API from implementation

PRACTICAL RESTFUL APPROACHES

- Note that the CRUD cycle can be mapped onto part of REST
- But REST is much than CRUD
  - CRUD gives permanence - it's about persisting data
  - REST cares about the entire hypermedia system and how hypermedia resources are interacted with via HTTP
- Many useful sites can be built on top of a CRUD based data cycle. Similarly for REST. Choice is dependent upon the problem domain in which the system is being developed for,
  - i.e. do you want a simple API to access persistent data or a full hypermedia application for interacting with that data?

CLEARER SEMANTICS

- REST provides more clear semantics over how to interpret HTTP methods applied to a URL:
  - Note that GET & PUT are considered to be *idempotent* - Applying them multiple times to the same URL should not lead to different behaviour

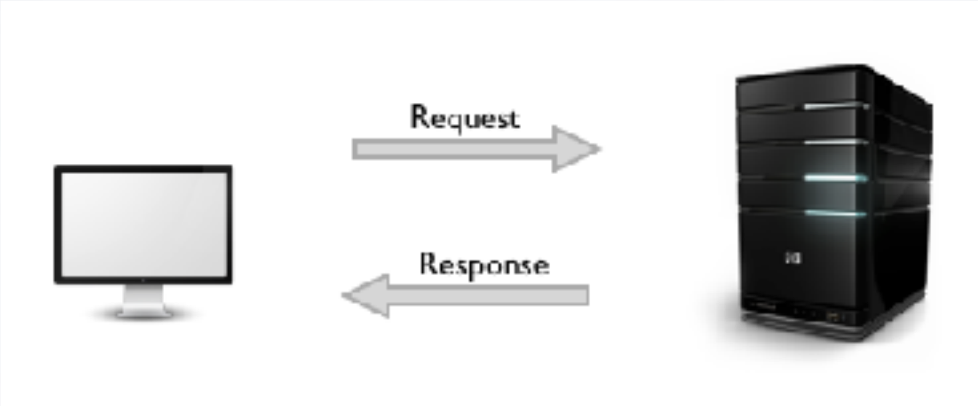
Operation	SQL	HTTP	REST
Create	INSERT	PUT/POST	POST
Read	SELECT	GET	GET
Update	UPDATE	PUT/POST/	PUT
Delete	DELETE	DELETE	DELETE

Edinburgh Napier UNIVERSITY

REST & THE WEB

- A RESTful system is a slightly relaxed interpretation of REST properties & constraints

- RESTful systems typically use HTTP & standard requests-responses:

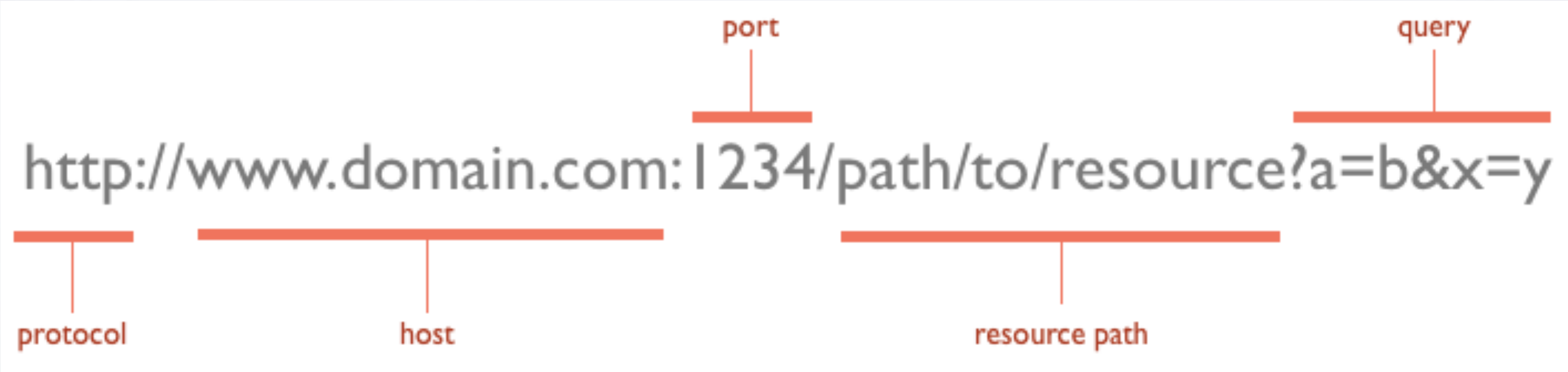


- Use the same verbs (GET, POST, PUT, DELETE, &c.) - that we have seen in labs & used (perhaps unknowingly until now) in our browsers - to retrieve web pages & send data to servers

- Often RESTful systems implement some form of Create-Retrieve-Update- Delete (CRUD) system

RESOURCES, COLLECTIONS, & URLS

- 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 - Nouns rather than verbs for resources



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.

REAL WORLD APIS

- Programmable Web API Directory

  http://www.programmableweb.com/apis/directory

- Programmable Web API Dashboard:

  http://www.programmableweb.com/apis

SUMMARY

- We should now…
    - Understand the relationship between APIs & the Web
    - Know about the CRUD & REST approaches to architecting APIs & sites
    - Be able to consider the design of a URL hierarchy in terms of the collections of data that it exposes

Edinburgh Napier UNIVERSITY

QUESTIONS ???

RESOURCES

- Express.js API Reference:

  https://expressjs.com/en/api.html

- Pug API Reference:

  https://pugjs.org/api/reference.html

- JSON Language Reference:

  https://json.org/

- JSON Lint Tool:

  https://jsonlint.com/

- Roy Fielding PhD Thesis "Architectural Styles and the Design of Network-based Software Architectures" specifically chapter 5 "Representational State Transfer (REST)":

  https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

SUMMARY

- You should now be able to:
  - Use the Express.js & Pug packages to implement your own dynamic web apps
  - Design your dynamic web apps
  - Use REST
  - Develop APIs
  - Take into account RESTful principles & CRUD architectures

QUESTIONS ???

COMING UP…

- We've seen how to build APIs & web-pages
- These are all about storing and manipulating information
- So far information has been either encoded into static web pages, or generated from code, or loaded from file
- Now lets round things out by considering data & datastores