UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

SENG365 Web Computing Architecture: #1 Introduction to HTTP and JavaScript

Ben Adams
Course Coordinator & Lecturer
benjamin.adams@canterbury.ac.nz
310, Erskine Building

What we'll cover today

- What is a web application? And why a course on it?
- Reference model for web applications, for SENG365
- HTTP servers
- Other bits

- Course administration
  - Teaching team
  - Labs
  - Assessments
    - Assignments

UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

What is a web application?

What makes an application a *web* application rather than *desktop* application or a *mobile* application, or an *embedded* application, or …?

What is a web application? Consider …

- Gmail or equivalent running in Chrome or equivalent
- Facebook running in a browser
- Office 365
- DropBox, Google Drive, Microsoft OneDrive, …

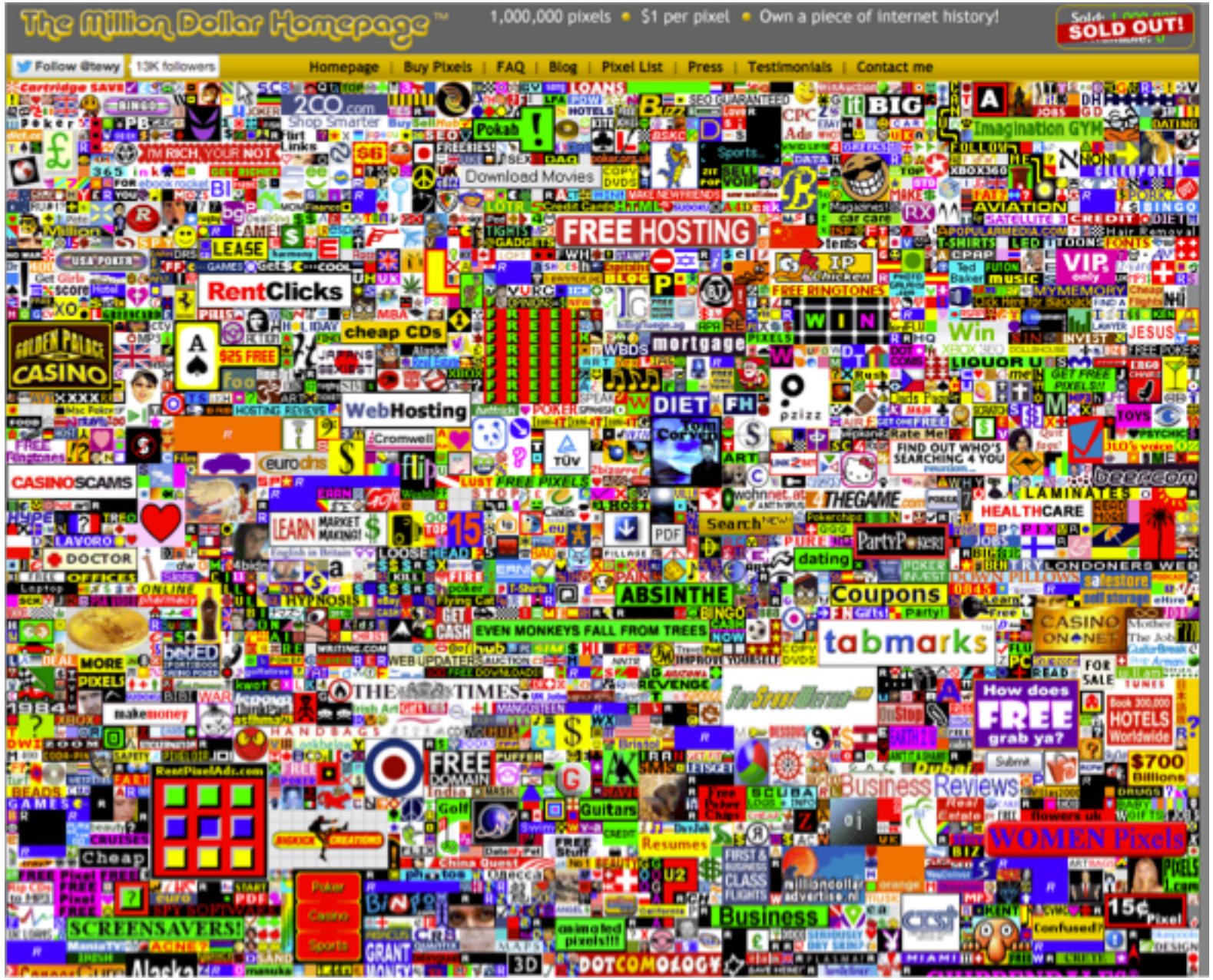
- Gmail application on your Android device (or iOS etc)
  - Native application
- Facebook application running in a browser on your tablet
  - Browser application

It’s 2005…

The iPhone wasn’t launched until 2007.

V for Vendetta was showing at cinemas (and you were too young to watch it)…

… *Star Wars: Episode III - Revenge of the Sith (*you were still too young*)*

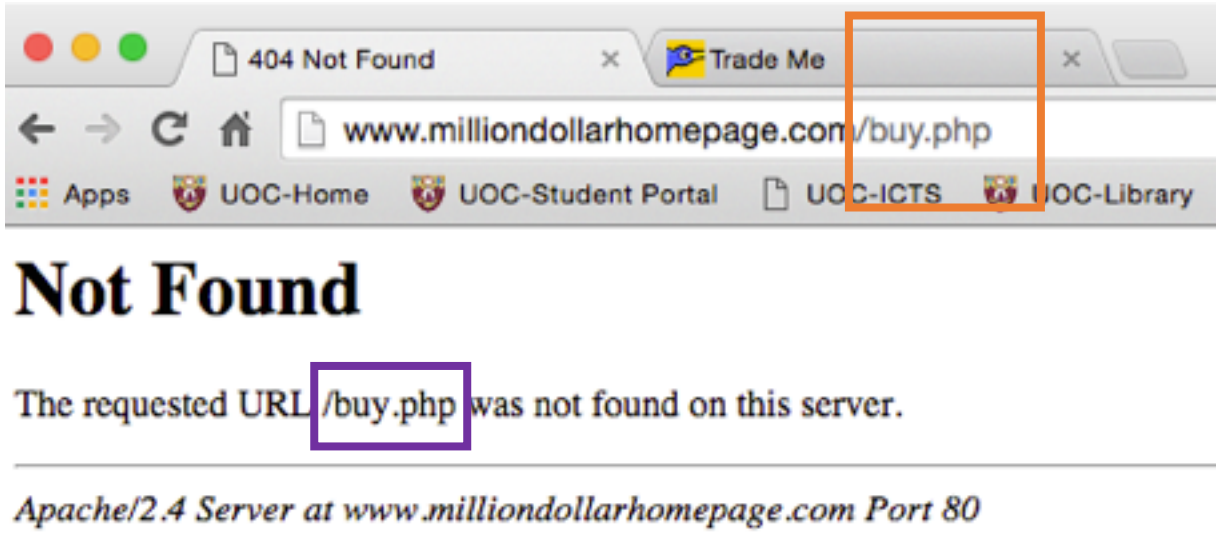


… and now

The internet fad of 2005 now stands as a stark demonstration of 'link rot' & 'system decay'.
22% of the links are dead
http://www.theguardian.com/technology/2014/mar/27/after-nine-years-the-million-dollar-homepage-dead (March 2014)

For a history of the site, see Wikipedia:
https://en.wikipedia.org/wiki/The_Million_Dollar_Homepage

php!



but what's this?
/buy.php
An API endpoint…?

UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

Why take a course on web application architecture?

How Prezzy® card works | :::Kiwi:::

https://www.prepaidaccount.co.nz/Prezzycustomer/html/FirstTimeLoginFrame.jsp

Prezzy card

First time login

By logging into this website for the first time, your card will automatically become active.

Enter Login Details All fields are mandatory

Enter Card Number [redacted]

CVV2 •••

Sign In | Sign In | Exit

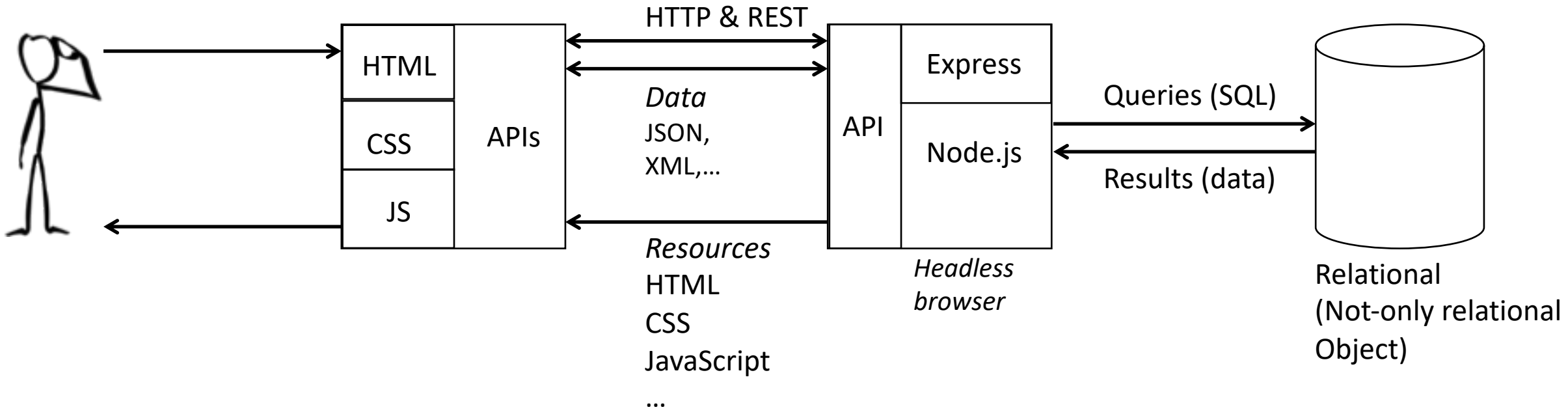
Date Of Birth Field cannot be left Blank

Challenges* for modern web applications

- Consume services from another system
- Provide services to another system
- Modularize my application (to manage complexity)
- Respond to multiple overlapping (asynchronous) requests
- Make changes persistent (for large, distributed systems)
- Allow and restrict user access (security and privacy)
- Display information from a source
- Synchronize information shown on different views
- Maximize responsiveness
- Adapt to different devices and screen sizes
- Protect user data from being harvested
- Protect my business from harm (prevent exploits)

* An illustrative list, not exhaustive

A reference model



User	HTTP client	HTTP Server	Database
Human	Machine	Machine	Machine

Assignment 1 server will need to handle

HTTP server + application

- HTTP request & response cycle
- URL e.g. protocol, path, endpoints, query parameters
- HTTP headers and body
  - Headers: e.g. Cookies
  - Headers: e.g. CORS
  - Body e.g. JSON data
- HTTP methods e.g. `GET`, `PUT`, `DELETE`
- HTTP status codes e.g. 201, 404

And also

- Authentication and authorization
- Asynchronous requests
- Database connectivity
- Conform to API specification
  - You will be given an API specification to implement

Assignment 2

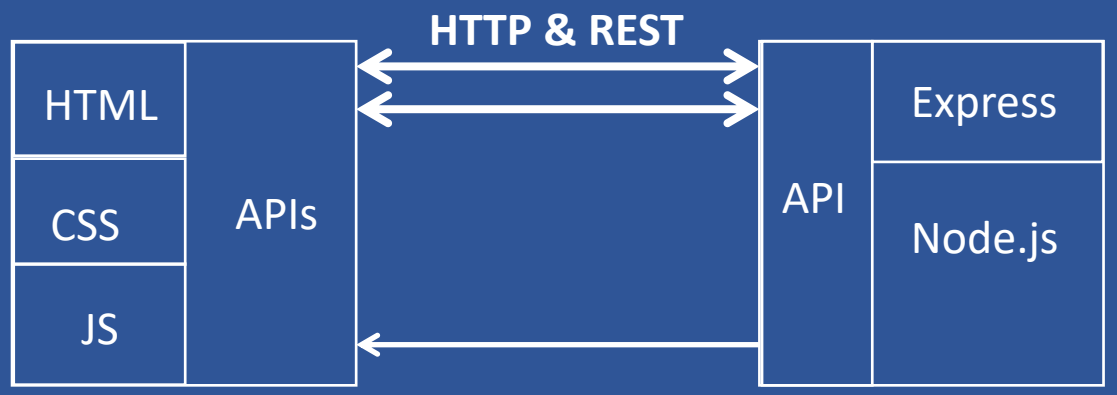
HTTP Client

- HTML + CSS + JS app
- Modern browser
- Implementing user story backlog

And also

- Authentication and authorization
- Asynchronous requests
- RESTful API calls

The HTTP protocol



Overview to HTTP

HTTP messages are how data is exchanged between a server and a client. There are two types of messages: *requests* sent by the client to trigger an action on the server, and *responses*, the answer from the server.

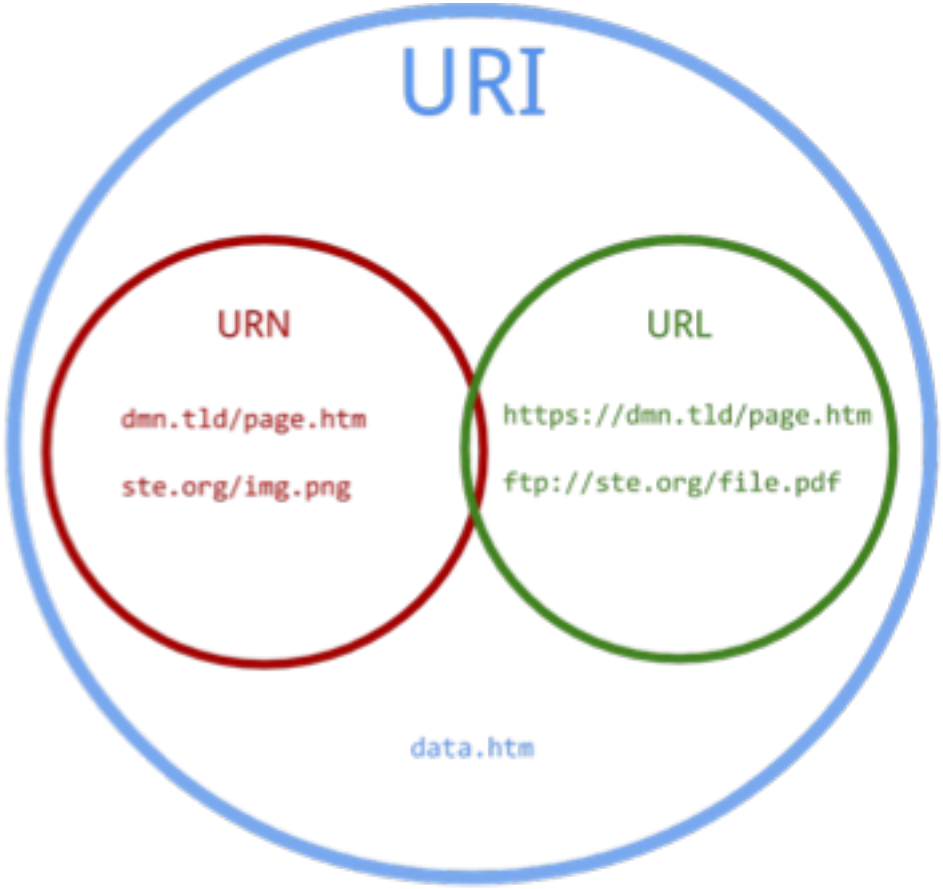
HTTP messages are composed of **textual** information encoded in ASCII*, and span over multiple lines. In HTTP/1.1, and earlier versions of the protocol, these messages were **openly** sent across the connection.

https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages

(* There's a bit more to it than just ASCII…)

Uniform Resource Identifiers (URIs)

https://danielmiessler.com/study/url-uri/



1. URI (Uniform Resource Identifier)
    - String of characters to identify (name, or name and location) resource
2. **URL (Uniform Resource Locator)**
    - A URI that also specifies the means of acting upon, or obtaining representation
    - That is, **a URI with access mechanism and location**
3. URN (Uniform Resource Name)
    - Deprecated: historical name for URI.

`scheme:[//[user[:password]@]host[:port]][/path][?query][#fragment]`

Your server needs to handle URLs like this:

```
http://www.example.com:80/path/to/myfile.html?key1=val1&key2=val2#Somewhere
```

The protocol

http://www.example.com:80/path/...

The domain name

`http://`**`www.example.com`**`:80/path/...`

The port

www.example.com:**80**/path/to/myfile...

- The default HTTP port is 80
- The default **HTTPS** port is **443**
- The port on which your HTTP server listens for requests is a different port to the port to which the server issues queries to the MySQL database (default 3306)

The path

...80/**path/to/myfile.html**?key1=val1...

- The path is increasingly an abstraction, not a ‘physical path’ to a file location.
- A path to an HTML file is not (quite) the same thing as the path to an API endpoint
- An API endpoint uses the standard URI path structure to achieve something different
  - In particular, parameter information
- The path may need to include information on the version of the API

An example *API endpoint* path

`...80/`**`api/v1/students/:id`**`?key1=val1...`

- The path contains versioning: **`api/v1/`**.
- There's an API endpoint: **`students`**
- The path contains a variable in the path itself: **`:id`**
  - How are these path variables handled by the server…?
- You may still pass query parameters: `?key1=val1`
  - Given the path variable, query parameters may be redundant for the endpoint
  - There is also the body of the HTTP request for passing information

Query parameters, and other parameters

`...path/to/myfile.html`**`?key1=val1&key2=val2`**`...`

The API could be designed to accept parameter information via

- The URI's `?` query parameters
- The URI's path (see previous slide/s) e.g. `:id`
- The body of the HTTP request e.g. JSON
- Or via some combination of the above
- What goes in query parameters, in the path, and in the body?

anchors

```
...key1=val1&key2=val2#Somewhere...
```

- Anchors used as ‘bookmarks’ within a classic HTML webpage
  - i.e. point to a ‘subsection’ of the page
- We don’t need to use anchors for our API requests
  - (Being creative, you might…?!?)

To recap: your server needs to handle this

```
http://www.example.com:80/path/to/myfile.html?key1=val1&key2=val2#Somewhere
```

As well as the URL, there are HTTP headers

Distinguish between

- General headers: required and `additional'
- Entity headers (that apply to the body of the request)

And between:

- Request headers
- Response headers

- Cookies are implemented in/with the header
  - `Set-Cookie: <...>` (in the header of the server's HTTP response)
  - `Cookie: <...>` (in the header of subsequent client HTTP requests)
- Use headers to:
  - Maintain session
  - Personalise
  - Track (e.g. advertising)

Structure and example of HTTP requests

HTTP requests are of form:

```
HTTP-method SP Request-URL SP HTTP-Version CRLF
*(Header CRLF)
CRLF
Request Body
```

Example GET (no body):

```
GET /pub/blah.html HTTP/1.1
Host: www.w3.org
```

Example POST (with indication of body):

```
POST /pub/blah2.php HTTP/1.1
Host: www.w3.org

Body of post (e.g. form fields)
```

KEY:

```
SP  = space
CRLF = carriage return,
   line feed (\r\n)
```

Example HTTP responses

Headers become relevant e.g. CORS

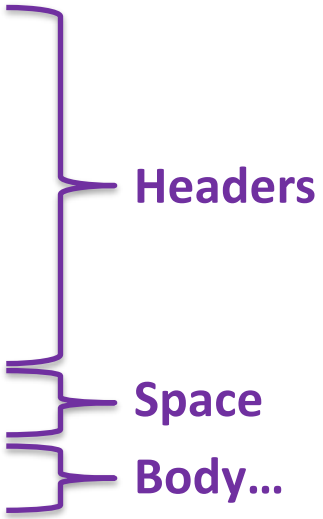
HTTP responses are of form

```
HTTP-Version SP Status-Code SP Reason-Phrase CRLF
*(Header CRLF)
CRLF
Response Body
```

Typical successful response (GET or POST):

```
HTTP/1.1 200 OK
Date: Mon, 04 Jul 2011 06:00:01 GMT
Server: Apache
Accept-Ranges: bytes
Content-Length: 1240
Connection: close
Content-Type: text/html; charset=UTF-8

<Actual HTML>
```



Response codes

1xx, informational (rare)

- e.g. 100 continue

2xx, success

- e.g. 200 OK, 201, Created, 204 No Content

3xx, redirections

- e.g. 303 See Other, 304 Not Modified

4xx, client error (lots of these)

- e.g. 400 Bad Request, 404 Not Found

5xx, server error

- e.g. 500 Internal Server Error, 501 Not Implemented

HTTP/1.1 and /2: https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html
http://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml

Brief examples in Node.js

With **`http`** package and with **`express`** package

Creating the HTTP server: simple example

- In your assignment you will use the `express` package

express: ‘Listening’ for a request to an endpoint

Note:
Nothing stated about
- ports
- domain names
- query parameters

The root of the *path* in the URL

A simple API endpoint:
An ’extension’ to the *path* in the URL

```
app.route(app.rootUrl + '/users')
    .post(users.create);
```

dot (.)
Method chaining

The HTTP
method: POST

express: Create a user and respond

The body of the HTTP request

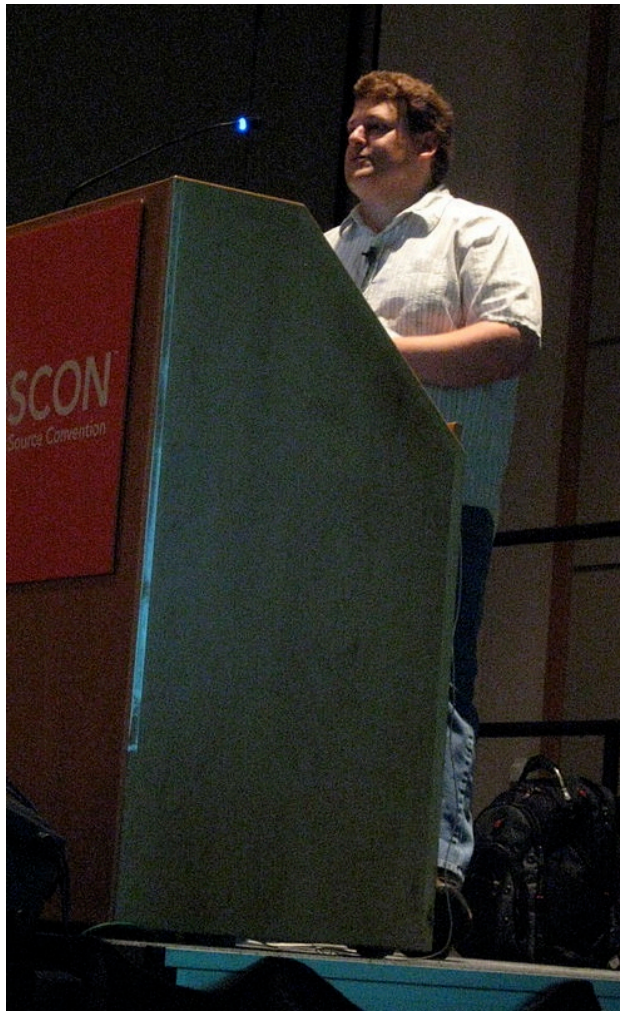
- Some HTTP requests (typically) do not need a body:
  - e.g. a `GET` request, a `DELETE` request
- Broadly, there are three categories of body:
  - Single-resource bodies, consisting of a single file of known length, defined by the two headers: `Content-Type` and `Content-Length`.
  - Single-resource bodies, consisting of a single file of unknown length, encoded by chunks with `Transfer-Encoding` set to chunked.
  - Multiple-resource bodies, consisting of a multipart body, each containing a different section of information. These are relatively rare.
- HTTP bodies can contain different kinds of content
  - We're going to be using JSON (because JSON is better than the others ;))

HTTP verbs

- GET
- PUT
- POST
- DELETE
- HEAD
- Others

UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

REST: Representational State Transfer



https://commons.wikimedia.org/wiki/File:Roy_Fielding.jpg

CHAPTER 5
Representational State Transfer (REST)

This chapter introduces and elaborates the Representational State Transfer (REST) architectural style for distributed hypermedia systems, describing the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, while contrasting them to the constraints of other architectural styles. REST is a hybrid style derived from several of the network-based architectural styles described in Chapter 3 and combined with additional constraints that define a uniform connector interface. The software architecture framework of Chapter 1 is used to define the architectural elements of REST and examine sample process, connector, and data views of prototypical architectures.

5.1 Deriving REST

The design rationale behind the Web architecture can be described by an architectural style consisting of the set of constraints applied to elements within the architecture. By examining the impact of each constraint as it is added to the evolving style, we can

https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf (2000)

REST and HTTP methods

REST asks developers to use HTTP methods **explicitly** and **consistently** with the HTTP protocol definition.

(cf. safe and idempotent methods)

CRUD and REST

- To **C**reate a resource on the server, you should use `POST`
- To **R**etrieve a resource, you should use `GET`
- To change the state of a resource or to **U**pdate it, … use `PUT`
- To remove or **D**elete a resource, you should use `DELETE`

Bad practice

- You don’t have to map CRUD operations to HTTP methods.
- Here’s a BAAAAD practice:

```
GET /path/user?userid=1&action=delete
```

- What’s this HTTP request doing?

The Joy (or not) of JavaScript

The JavaScript way of programming

- Objects, methods & functions
- Expressions, statements and declarations
- Functions
  - Immediately invoked function expressions (IIFE)
- Scoping
- Variables
  - Variable hoisting
- Closures
- `this`
- Method chaining (cascading)
- `'use strict';` mode

Next week(ish)

- Modularisation: `export` & `require()`
- Node.js
- Asynchronous (event) handling
  - Callbacks, Promises, Async/Await

Objects, methods and functions

Objects:

- JavaScript is an **object-oriented** programming language
  - Not as strict as Java, in its definition of objects e.g. not compulsory to have classes
- An **object** is a collection of properties, and a **property** is an association between a name (or *key*) and a value.
- A property can itself be an object.
- A ***method*** is a function associated with an object; or, alternatively, a method is a property (of an object) that is a function.

Functions

- Functions are first-class objects
- They can have **properties** and **methods**, just like any other object.
- Unlike other objects, functions can be called.
- Functions are, technically, function objects.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions

Expressions, statements, declarations …

- An expression produces a value.
- A statement does something.
- Declarations are something a little different again: creating things.
- BUT, JavaScript has:
  - Expression statements: wherever JavaScript expects a statement, you can also write an expression!
  - The reverse does not hold: you cannot write a statement where JavaScript expects an expression.
- http://2ality.com/2012/09/expressions-vs-statements.html

Functions and their execution (1/n)

Example 1

```
var result = function aFunction (){
  return -1;
};
```

What is the value of `result`?

Functions and their execution (2/n)

Example 2

```
var result1 = function aFunction1 (){
  return -1;
};
var foo = result1();
```

What is the value of `foo`? Why?

Functions and their execution (3/n)

Example 3

```
var result1 = function aFunction1 (){
  return -1
}();
```

What is the value of `result1`?

Functions and their execution (4/n)

Example 4

```
(function aFunction3 () {
    return 2;
}());
```

What is happening here? Why?

Digression: A pair of brackets, `()`

The pair of brackets, `()`, is:

- Used to execute a function e.g. `function();`
- The grouping operator e.g. to force precedence
  `(a + b) * c;`

Immediately invoked function expressions (IIFE)

```
(function () {
    statements
})();
```

Also known as: Self-Executing Anonymous Function

A JavaScript function that runs as soon as it is defined

Immediately invoked function expressions (IIFE)

```
(function () {
    statements
})();
```

The **outer** brackets, `(function...)()`, enclose an anonymous function.

The subsequent empty brackets, `();` execute the function.

The anonymous function establishes a lexical scope. Variables defined in `statements` cannot be accessed outside the anonymous function

Uh-oh: Immediately invoked function expression not invoking

At the console I type this:

```
> function () {
    statements
}();
```

But this doesn’t work. Why?

Oh! These IIFEs are working! Why?

```
+function afunction () {
    console.log('Here I am!');
}();

!function afunction () {
    console.log('Here I am!');
}();
```

http://2ality.com/2012/09/expressions-vs-statements.html

Scoping: block vs lexical

Block scope (Java, C#, C/C++)

```
public void foo() {
  if (something){
    int x = 1;
  }
}
```

`x` is available only in the `if () {}` block

Lexical scope (JavaScript, R)

```
function foo () {
  if (something) {
    var x = 1;
  }
}
```

`x` is available to the `foo` function (and any of `foo`'s inner functions)

JavaScript functions

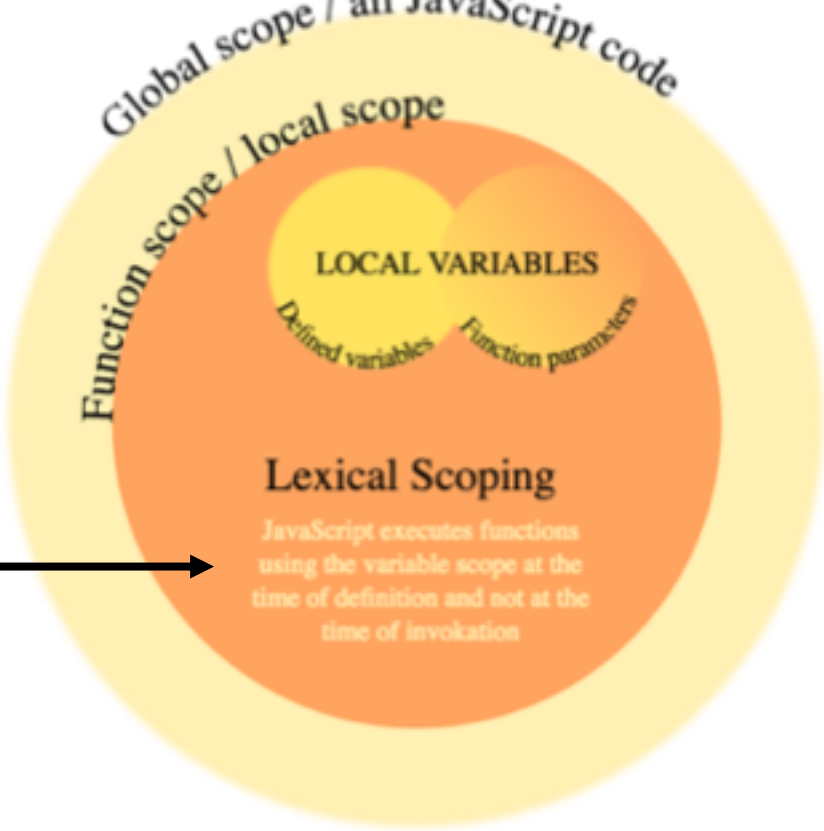
JavaScript executes any function:

- using the variable scope at the time of definition of the function
- **not** the variable scope at the time of invocation of the function

In other words:

- Did the variable exist at the time of definition e.g. in an outer function?

This approach to function execution supports **closures**.



https://commons.wikimedia.org/wiki/File:SCOPE_in_JavaScript.svg

Variables (things have changed with ES6)

Examples of JS variables

```
a = 1; //undeclared
var b = 1;
let c = 1;
const d = 1;
```

- Undeclared variables shouldn't be used in code
  - But they can be
    - Unless you use `'use strict';`
  - Doesn't exist until the assignment
- Always declare a variable
- `var` is **lexically** scoped
- `let` is **block** scoped
- `const` is **block** scoped, and can't be changed

To `var`, or not even to `var`

```
function x(){
      y = 1;       // Throws a ReferenceError in strict mode
                   // Otherwise creates a global variable!
      var z = 2;   // Constrained to the execution
                   // environment in which it's declared
                   // i.e. lexical scope.
}
```

BUT… ES6 changes things

`var`:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var

`let`:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let

`const`:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const

Note: with `const`, the reference to an object can't change, but the properties of the object being referenced can be changed

Variable *hoisting*

```
function foo () {
  // x hoisted here
  if (something) {
    let x = 1;
  }
}
```

- Variable declarations in a function are *hoisted* (pulled) to the top of the function.
  - Not variable assignment
- *Invoking* functions before they're declared works using hoisting
  - Note: doesn't work when assigning functions

Closures

When JavaScript executes a function (any function), it:

- uses the **variables in-scope at the time of definition** of the function
- **not** the variable scope at the time of invocation of the function

`this` needs careful attention

- With any given piece of JavaScript code, the context (for that code) is made up of:
  - The current function's (lexical) scope, and
  - Whatever is referenced by `this`
- By default **in a browser**, `this` references the global object (`window`)
- By default **in node**, `this` references the global object (`global`)
- `this` can be manipulated, for example
  - Invoke methods directly on an object e.g. with `foo.bar();` the object `foo` will be used as `this`.
- But `this` is fragile:
  - `let fee = foo.bar;` // this=foo
  - `fee();` // this=global/window

Method chaining (cascading)

```
let result = method1().method2(args).method3();
```

- Each method in the chain returns an object…
  - each method must execute a `return…;` statement
- The returned object must ‘contain’ within it the next method being invoked in the chain.
  - `method1()` returns an object that has `method2()` within it, so that you can call `method2()`
- The first method in the chain may need to create the object.
- Usually, each method contains a `return this;` as a pointer to a *common* object being worked with
  - That common object contains all of the methods e.g. `method1`, `method2` and `method3`
- You can’t just arbitrarily chain together any ol’ set of methods

Example

```
let anotherperson = {firstname: 'Ben',
                     surname: 'Adams',
                     printfullname: function () {
                         console.log(this.firstname + ' ' + this.surname);
                         return this;
                     },
                     printfirstname: function () {
                         console.log(this.firstname);
                         return this;
                     },
                     printsurname: function () {
                         console.log(this.surname);
                         return this;
                     }
}
```

Why chain

- Reduces temporary variables
  - No need to create temporary variable(s) to save each step of the process.
- The code is expressive
  - Each line of code expresses clearly and concisely what it is doing
  - (Using verbs as names for methods helps).
- The code is more maintainable
  - Because it’s easier to read e.g. it can read like a sentence.
  - Because it requires a coherent design to the chained methods
- Method chaining used in, for example, Promises and other ‘then-able’ functions

- Remember: you can’t chain any ol’ bunch of methods together.

`'use strict'` (Strict mode)

Strict mode:

- (a way of managing backward compatibility)
- modifies ***semantics**** of your code
  (modifies the interpretation of your code), e.g.:
  - `this` is defaulted to `undefined`
  - less lenient about variable declarations e.g. `var`
  - throws errors rather than tolerating some code
  - rejects `with` statements, octel notation
  - Prevents keywords such as `eval` being assigned
- like a linter (e.g. linters give warnings and strict mode throws errors)
  - Linters need to be configured to 'play nicely' with strict mode
- can be applied to entire script or at function level

* not **syntax**

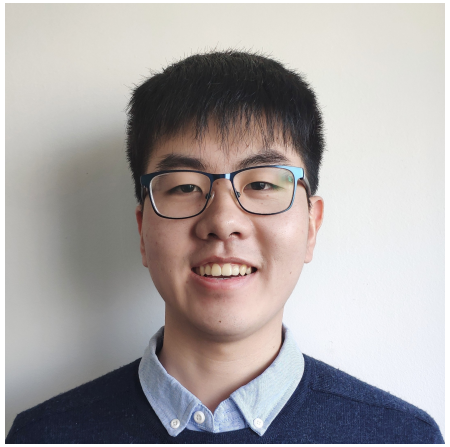
UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

Course administration

Teaching team

Ben Adams
Course Coordinator
& Lecturer

Morgan English
Tutor



Yu Duan
Tutor

Patricia Inez
Senior tutor

Contacting us

- Individual email addresses on Learn
- Prefer you to use seng365@canterbury.ac.nz. This will forward to entire teaching team
- Forums on Learn – often get answers quicker from your classmates
- Come to lab sessions!

Labs

Term 1 (6 weeks)

- Week 1: 3 x pre-labs (self-study)
- Week 2: Lab 1
- Week 3: Lab 2
- **Week 4: Lab 3**
- Week 5 & 6: assignment support

Term 2 (6 weeks)

- Week 7: Lab 4
- Week 8: Lab 5
- Week 9: Lab 6
- Week 10 & 11: assignment support
- Week 12: **compulsory** lab
  - Assignment 2 testing

Assessment

The assessment

- Assignment 1 (25%)
  - **No extension**
- Mid-semester test (20%)
  - **Wed 24 March, 7pm**
  - **E8 Lecture Theatre**
- Assignment 2 (25%)
  - Compulsory lab in final week
  - **No extension**
- Exam (30%; 2hrs)

Additional information and requirements

- Assignment resources on Learn
- API specification with skeleton project
- Infrastructure
  - eng-git project
  - MariaDB server (MySQL compatible)
  - Docker VM server
  - ‘Behavioural server’

Where next?

- Continue with labs
- Assignment 1 projects available ~end of week
  - API specification
- Next week’s lecture
  - Continue with JavaScript and asynchronous behaviour

UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

SENG365 Web Computing Architecture: #1 Introduction to HTTP and JavaScript

Ben Adams
Course Coordinator & Lecturer
benjamin.adams@canterbury.ac.nz
310, Erskine Building