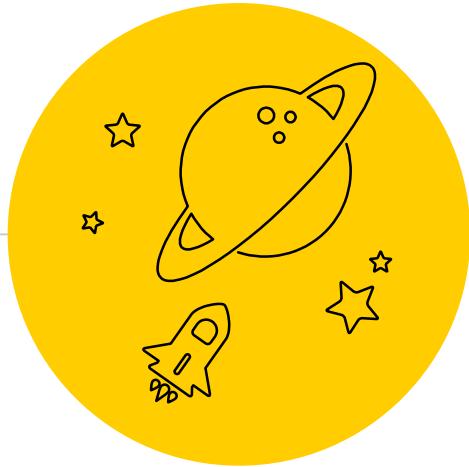


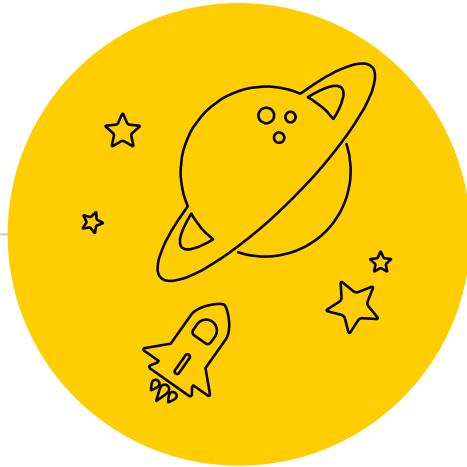
SENG 365 Week 1

Intro to HTTP and JavaScript





What is a web application?



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...

- TradeMe
- Gmail running in Chrome or equivalent
- Gmail app running on Android or iOS
- Facebook running in a browser
- Office 365
- DropBox, Google Drive, ...

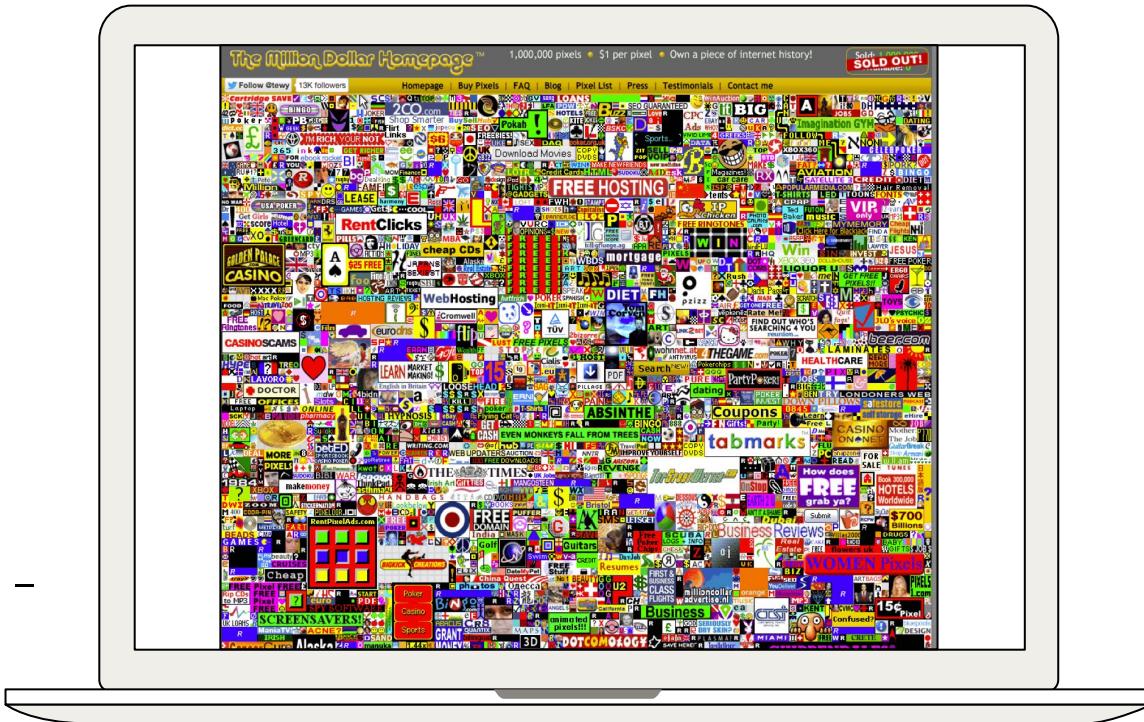


It's 2005...

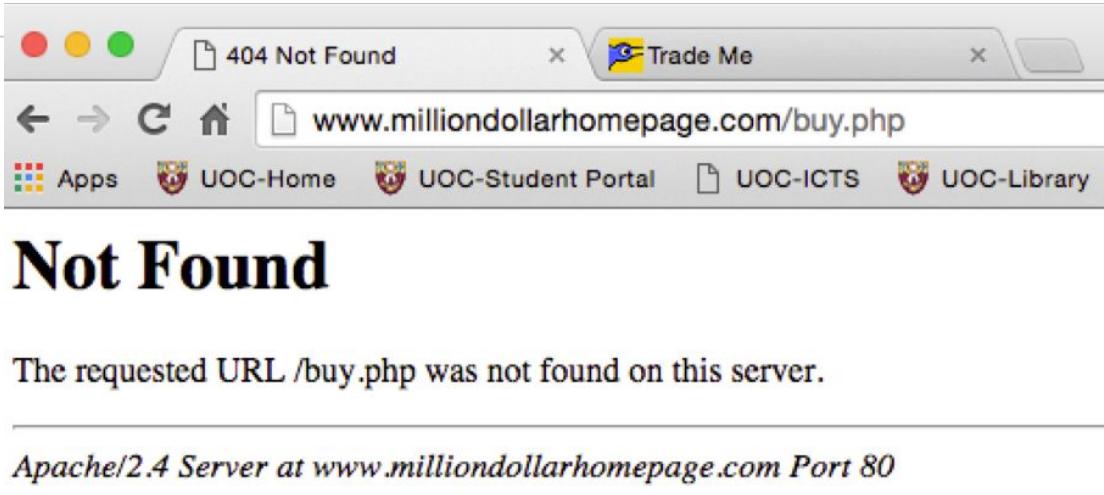
The iPhone wasn't launched until 2007.

V for Vendetta was showing at cinemas ...

... Star Wars: Episode III – Revenge of the Sith



php!



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

... and now

The internet fad of 2005 now stands as a stark demonstration of '**link rot**' & '**system decay**'.

22% of the links were dead by 2014

<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



1

Why take a course on web application architecture?

How Prezzy® card works

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

CVV2

Sign In **Sign In** **Exit**

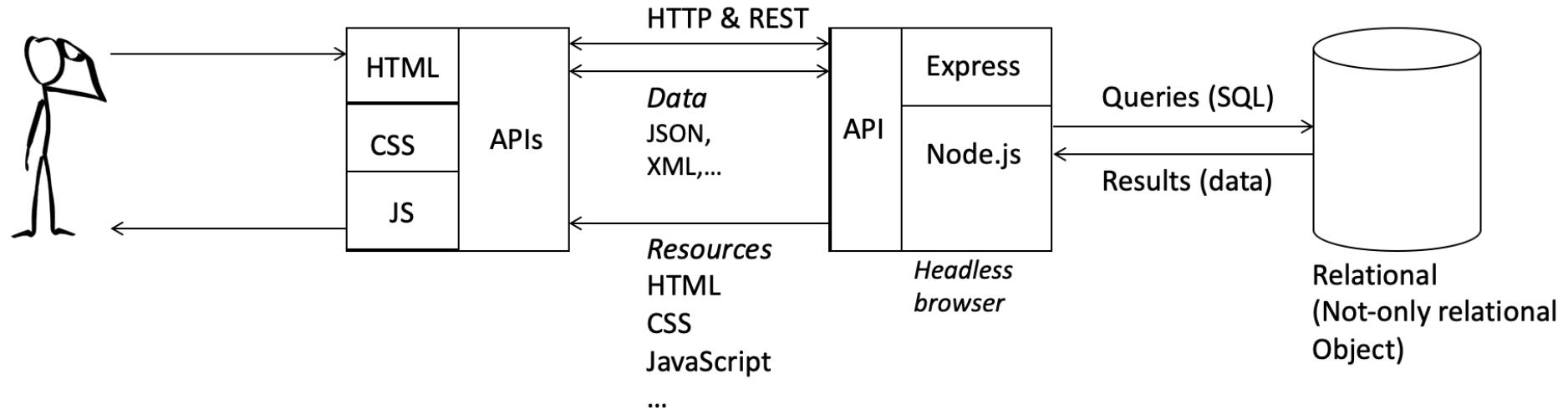
Date Of Birth Field cannot be left Blank





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**)



User	HTTP client	HTTP Server	Database
Human	Machine	Machine	Machine

Reference model



2

Course administration

Teaching team, Course requirements, Assignments



Teaching team

Ben Adams

Lecturer and Course
Coordinator

Erskine 310

benjamin.adams@canterbury.ac.nz

Moses Wescombe

Tutor

Frederik Markwell

Tutor

Morgan English

Senior Tutor

Erskine 324

morgan.english@canterbury.ac.nz



Overview of lecture topics

Term 1

- **Week 1 & 2:** HTTP, JS & asynchronous flow
- **Week 3:** TypeScript and Data persistence
- **Week 4:** HTTP Servers and APIs
- **Week 5:** GraphQL, API Testing
- **Week 6:** Security, Client-side basics

Term 2

- **Week 7:** Single Page Applications
- **Week 8 & 9:** React
- **Week 10:** Communication with server, performance
- **Week 11:** Web storage, Progressive web apps
- **Week 12:** Testing, Review



Assessment

The Assessments

- **Assignment 1** (30%)
 - No extension
- **Assignment 2** (30%)
 - No extension
- **Final Exam** (40%)
 - 2 hours

Additional information and requirements

- Assignment resources on Learn
- API specification with skeleton project
- Infrastructure
 - eng-git project
 - MySQL database
 - Postman tests



Assignment 1 - API Server

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 - Client front-end

HTTP client

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

And also

- Authentication and authorization
- Asynchronous requests
- RESTful API calls



Labs

Term 1

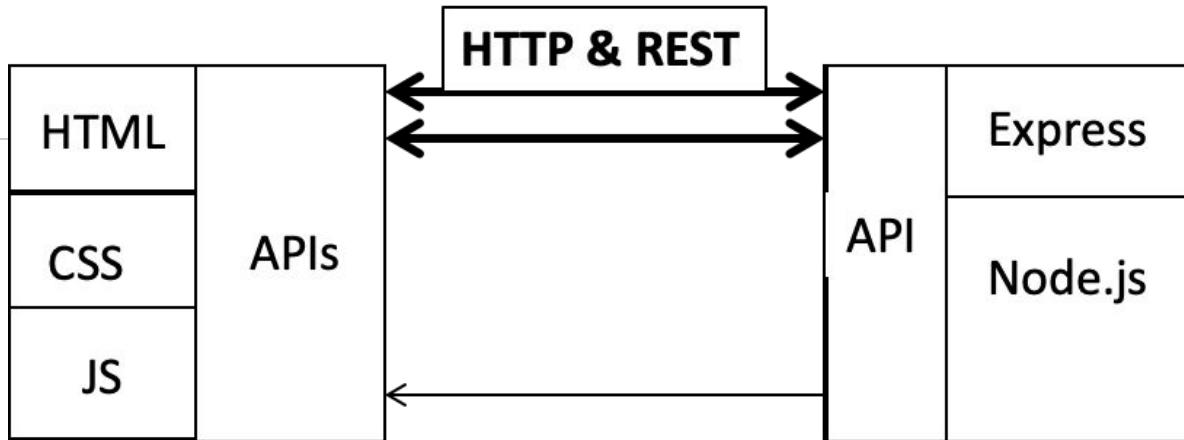
- **Week 1:** 3 x pre-labs (self-study)
- **Week 2:** Lab 1
- **Week 3 & 4:** Lab 2
- **Week 5 & 6:** assignment support

Term 2

- **Week 7:** Lab 3
- **Week 8:** Lab 4
- **Week 9:** Lab 5
- **Week 10 & 11:** assignment support
- **Week 12:** assignment 2 testing (**attendance mandatory!**)

The HTTP protocol

3





Overview to **HTTP**

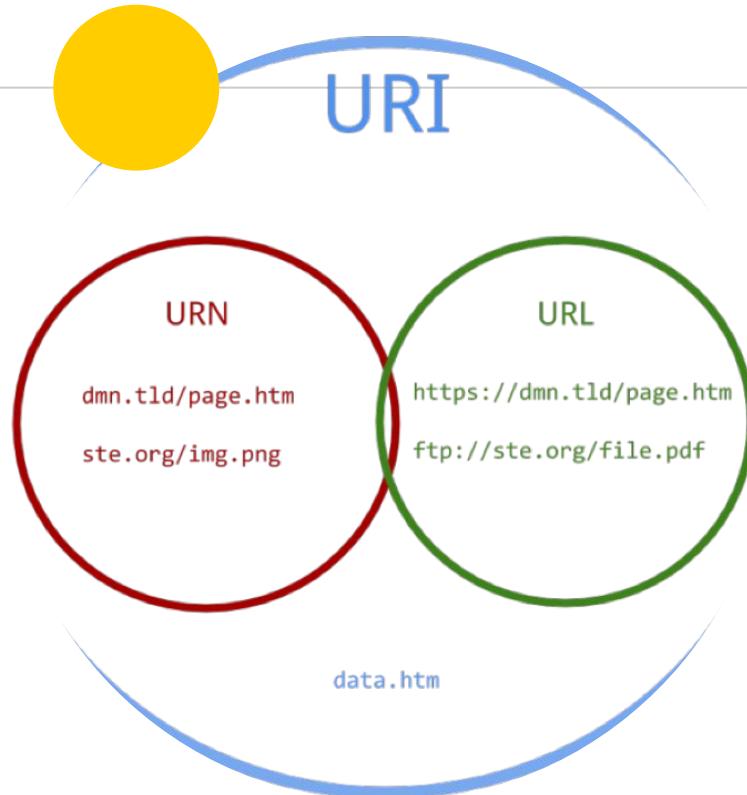
HTTP messages are how data is exchanged between a server and a client. There are two types of messages: requests sent by the client to trigger an action on the server, and responses, the answer from the server.

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



- **URI (Uniform Resource Identifier)**
String of characters to identify (name, or name and location) resource
- **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
- **URN (Uniform Resource Name)**
Deprecated: historical name for URI.

Your server needs to handle URLs like:

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

The protocol

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

The domain name

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

The port

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

- 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

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

- 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

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

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

http://www.example.com:80/path/to/myfile.html?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...?!?)



HTTP Headers

Distinguish between

- **General headers** required
- **Entity headers** (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

KEY:

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

Example POST (with indication of body):

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

Body of post (e.g. form fields)



HTTP verbs

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



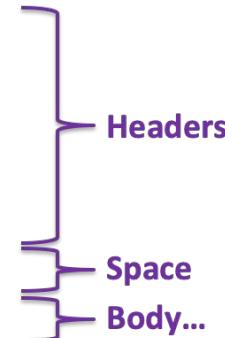
Structure and example of HTTP responses

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



Brief examples in Node.JS

With **http** package and with **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

Object containing
content of request body

```
try {
  const userId = await Users.create(req.body);
  res.statusMessage = 'Created';
  res.status(201)
  .json({ userId });
} catch (err) {
  ...
}
```

HTTP status message

HTTP status code (201)

Set HTTP headers and
return JSON in body



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 😊)

4

JavaScript



The JavaScript way of programming

Highlights

- ◉ Objects, methods & functions
- ◉ Expressions, statements and declarations
- ◉ Functions
- ◉ Immediately invoked function expressions (IIFE)
- ◉ Scoping
- ◉ Variables
- ◉ Variable hoisting
- ◉ Closures
- ◉ **this**

Next week

- ◉ Method chaining (cascading)
- ◉ 'use strict'; mode
- ◉ Modularisation: `export` & `require()`
- ◉ Node.js
- ◉ Asynchronous (event) handling
- ◉ Callbacks, Promises, Async/Await



Objects and methods

- 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
- 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

Example 1

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

What is the value of result?



Functions and their execution

Example 2

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

What is the value of foo? Why?



Functions and their execution

Example 3

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

What is the value of result1?



Functions and their execution

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  
})();
```

- 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: IIFE not executing

At the console I type this:

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

But this doesn't work. Why?



But: These IIFEs are working, Why?

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

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



Scoping

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

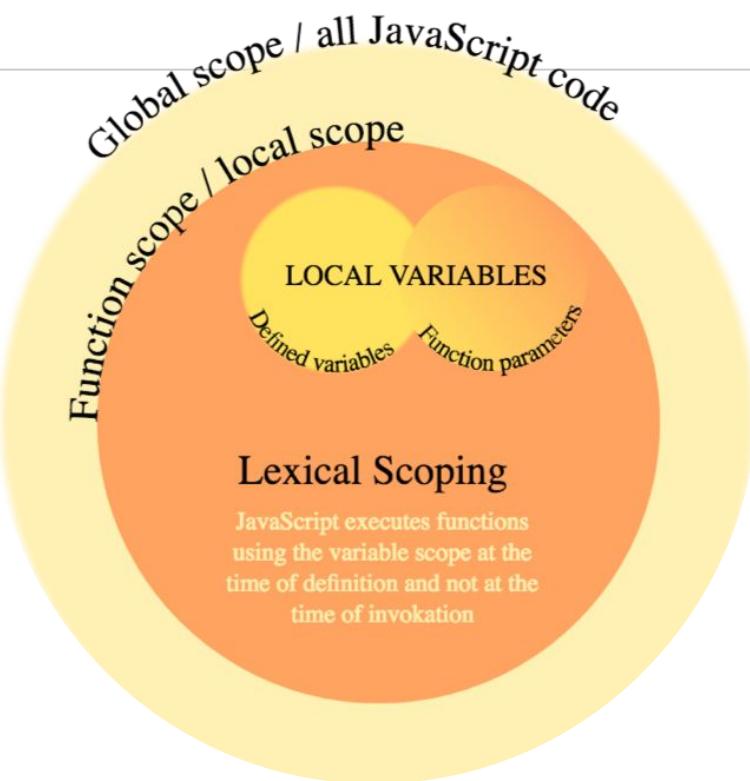
```
public void foo() {  
    if (something){  
        int x = 1;  
    }  
}
```

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**.



Things have changed with ES6

Examples of JS variables

```
a = 1; //undeclared  
var b = 1;
```

New in ES6

```
let c = 1;  
const d = 1;
```

Undeclared variables shouldn't be used in code

But they can be, unless you use 'use strict';

Always declare a variable

var is lexically-scoped

let is block-scoped

const is block-scoped, and can't 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
- a closure is a **record** storing a **function *together with an environment***
 - Variables **used locally** but **defined in enclosing scope**



this needs careful attention

The context of any given piece of JavaScript 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`



More JS next week

Any *questions* ?

SENG 365 Week 2

More JavaScript and Asynchronous Flow





The story so far

In the lectures

- Introduction to Web Computing
- HTTP
- JavaScript basic concepts
- Introduction to the assignments

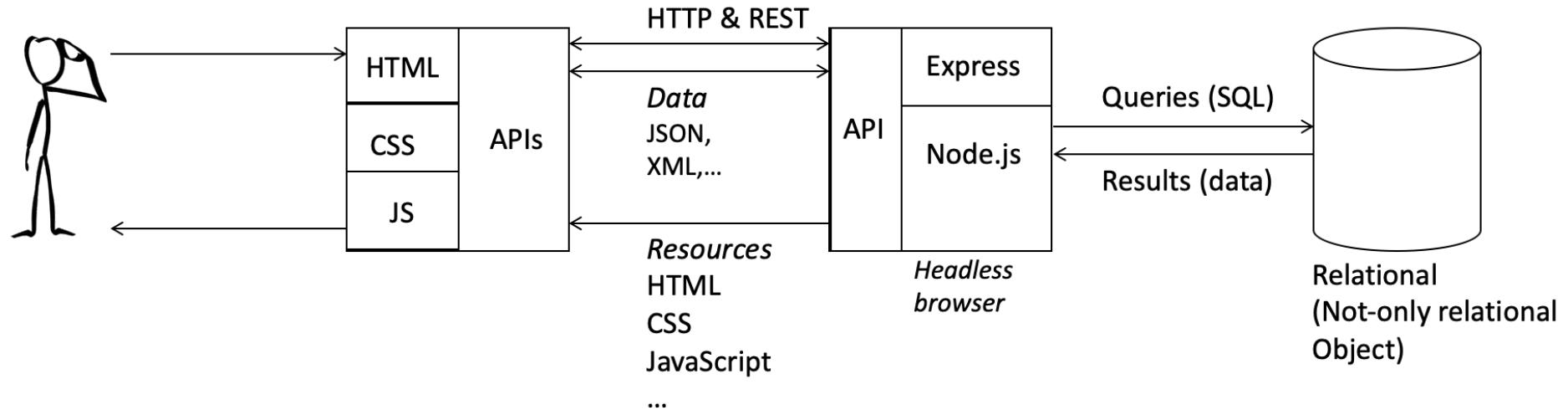
In the lab

- Pre-labs x 3
- Coming up:
 - Introduction to Node.js
 - Introduction to persistence
 - Structuring your server application



In lecture this week

- Method chaining
- Asynchronous programming
- Event loop
- Callback hell
- Promises
- async/await syntax
- Module dependency



User	HTTP client	HTTP Server	Database
Human	Machine	Machine	Machine

Reference model





Assignment 1

- Your eng-git repo has been created
 - skeleton project
 - Clone from eng-git into your own development environment
 - Install node modules: `npm install`
 - Create a `.env` file in the root directory of your project
 - Add `.env` to `.gitignore`
 - Add your specific environment variables to `.env`
 - API specification (see next slide)
 - `README.md`



The assignment in essence

- (Assignment Briefing on Learn)
- Implement the API specification provided in the repo
- We will assess the implementation using a suite of automated tests.
 - Assessing API coverage: how much of the API was implemented?
 - Assessing API correctness: was an endpoint correctly implemented?
- The automated tests are available for you
 - See the information in the README.md
 - You can see how well you are progressing
- For the actual assessment we will use different data, but intend to use the same (or similar) automated test suite.



JavaScript cont.



Arrow functions (ES6)

New anonymous function notation

```
function (a, b) { return a + b; }
```

Becomes

```
(a, b) => a + b;
```

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



Arrow functions and **this**

Unlike anonymous functions, arrow functions **do not bind their own this**

...

```
this.color = 'red';  
setTimeout(() => {this.color = 'green'}, 1000);
```



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 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



'use strict'

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, octal 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



Asynchronous JavaScript



The Event Loop (JavaScript Concurrency model)

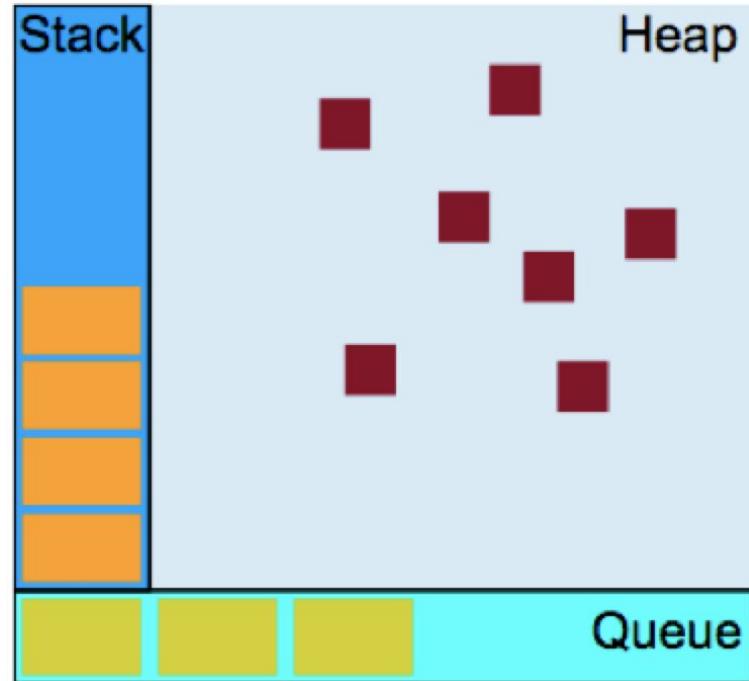
Call Stack: a data structure to maintain record of function calls.

- Call a function to execute: push something on to the stack
- Return from a function: pop off the top of the stack.

(The single thread.)

Heap: Memory allocation to variables and objects.

Queue: a list of messages to be processed and the associated callback functions to execute.



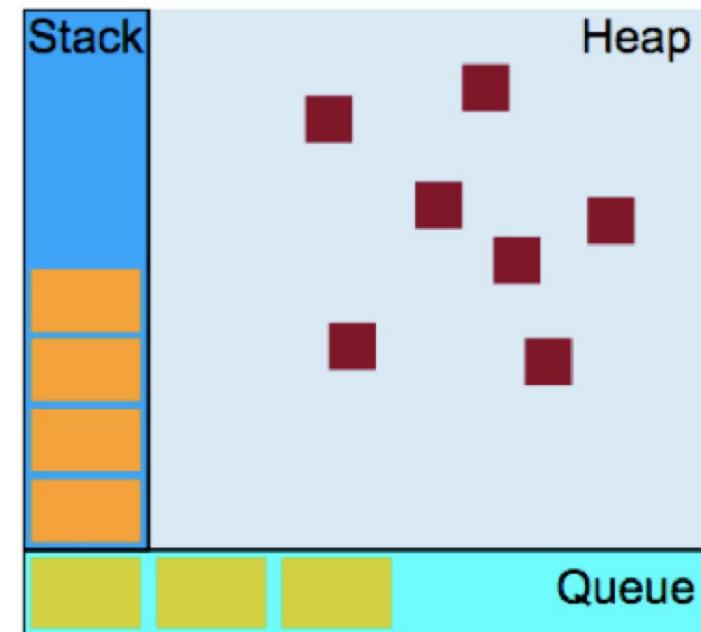


An initial example

What will complete first? ... and why?

Line Code

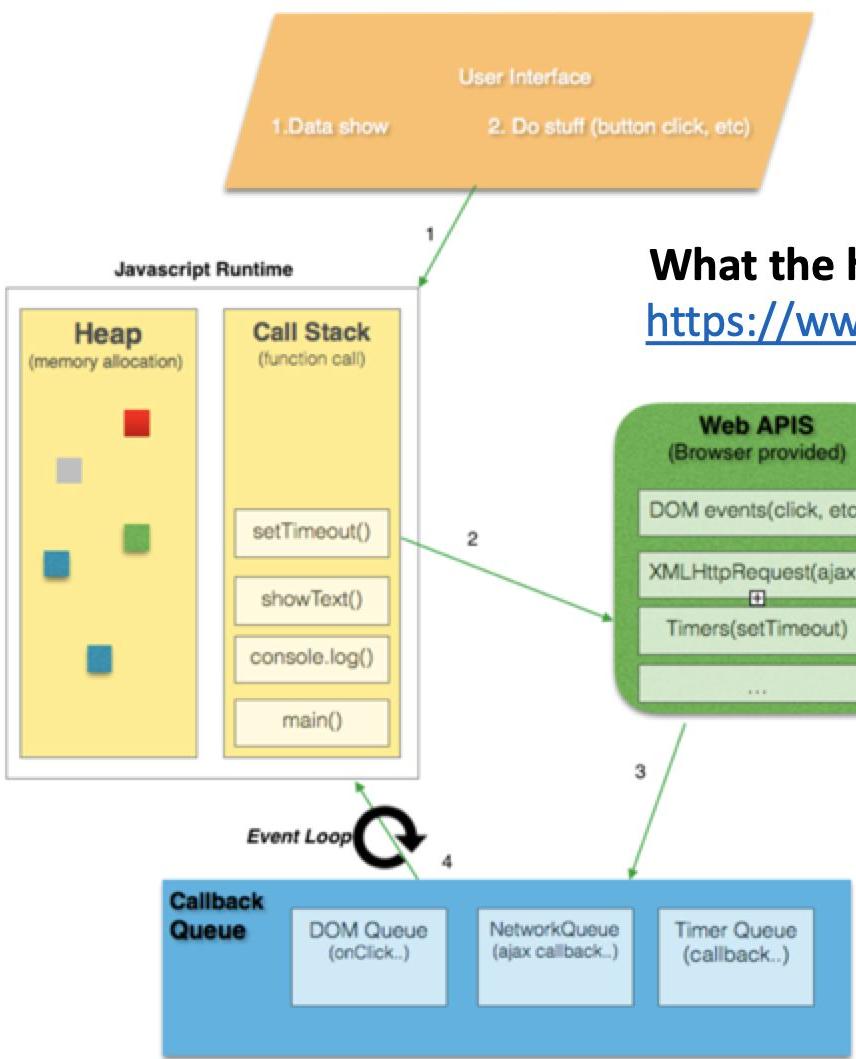
```
1      setTimeout(() => console.log('first'), 0);  
2      console.log('second')
```



```
1  /* jshint esversion: 6 */
2
3  let currentDateTime;
4  let currentTime;
5
6  console.log('Script start: ' + getTheTime());
7
8  function ping() {
9    console.log('Ping: ' + getTheTime());
10 }
11
12 console.log('ping function declared: ' + getTheTime());
13
14 function sayHi(phrase, who) {
15   console.log( phrase + ', ' + who + ', it\'s ' +
16   *   getTheTime());
17 }
18
19 console.log('sayHi function declared: ' + getTheTime());
20
21 setInterval(ping, 500); // Initiate ping events
22 setTimeout(sayHi, 1000, "Hello", "Austen"); // Initiate
23 * message
24
25 console.log('setInterval and setTimeout executed: ' +
26 *   getTheTime());
27
28 function getTheTime () {
29   currentDateTime = new Date();
30   currentTime = currentDateTime.toLocaleTimeString();
31   return currentTime;
32 }
33
34 console.log('Script end: ' + getTheTime());
```

setTime() and setInterval() code examples





What the heck is the event loop anyway?

<https://www.youtube.com/watch?v=8aGhZQkoFbQ>



Callback Hell and the Pyramid of Doom

by Asynchronous JavaScript

```
//TODO: refactor, to avoid the pyramid of doom, by using promises

}); db.getPool().query('DROP TABLE IF EXISTS bid', function (err, rows){
  if (err) return done({ "ERROR": "Cannot drop table bid" });
  console.log("Dropped bid table.");

}); db.getPool().query('DROP TABLE IF EXISTS photo', function (err, rows){
  if (err) return done({ "ERROR": "Cannot drop table photo" });
  console.log("Dropped photo table.");

}); db.getPool().query('DROP TABLE IF EXISTS auction', function (err, rows){
  if (err) return done({ "ERROR": "Cannot drop table auction" });
  console.log("Dropped auction table.");

}); db.getPool().query('DROP TABLE IF EXISTS category', function (err, rows){
  if (err) return done({ "ERROR": "Cannot drop table category" });
  console.log("Dropped category table.");

}); db.getPool().query('DROP TABLE IF EXISTS auction_user', function (err, rows){
  if (err) return done({ "ERROR": "Cannot drop table auction_user" });
  console.log("Dropped auction_user table.");
  done(rows);
});

}); });

}); });

});
```



APIs and **callback hell**

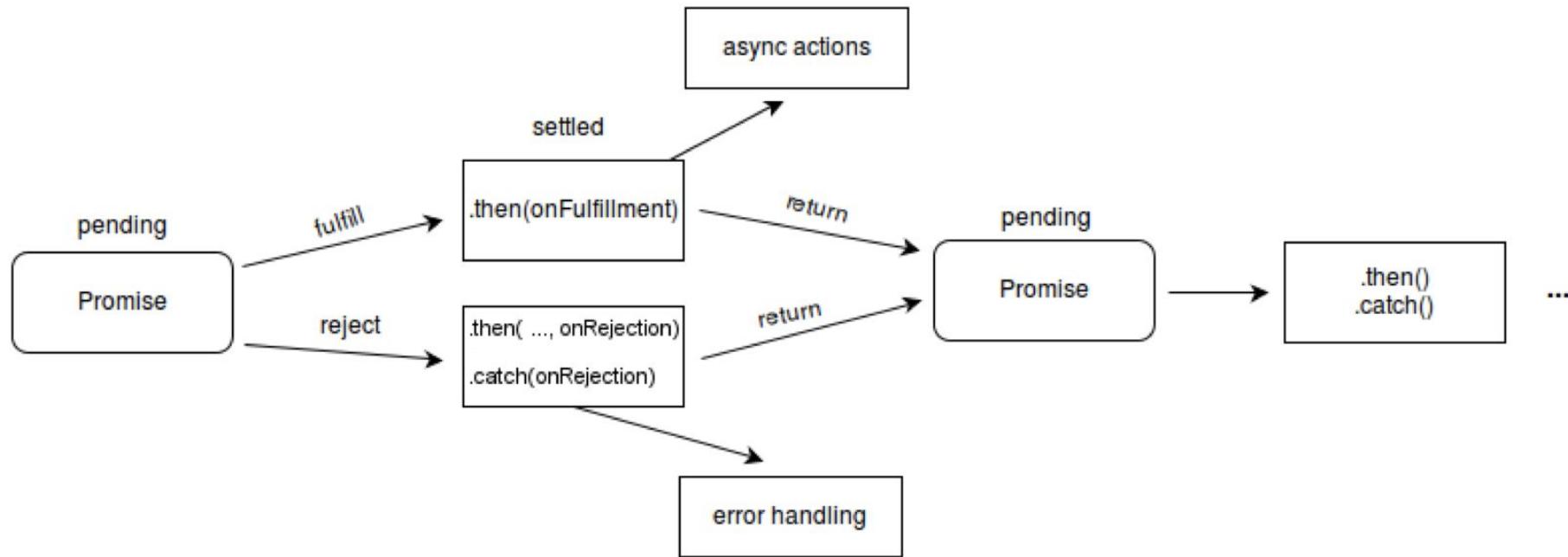
Consider that:

- ◉ An API call from the client may under-fetch data...
(the API is not designed to provide all and only the data the client needs)
- ◉ ... so the client will need to make subsequent API calls.
- ◉ For example:
 - First, an API call to get a list of student IDs in order to select one ID, then
 - an API call to get the list of courses studied by that student, and then
 - another API call to get further information of specific courses
- ◉ This produces a nested (conditional) set of API calls
 - For each call, the client must test whether the call was successful or not



Promises

- The **Promise** object is used for deferred and asynchronous computations.
- Promises allow you to **use synchronous and asynchronous** operations with each other
- A Promise represents an **operation that hasn't completed yet**, but is expected in the future.
 - **pending**: initial state, not fulfilled or rejected.
- Or resolved as either:
 - **fulfilled**: meaning that the operation completed successfully.
 - **rejected**: meaning that the operation failed.





Chaining **Promises**

- ◉ Each Promise is first **pending**, and then (eventually) either **fulfilled** OR **rejected**
- ◉ Chaining Promises allows you to chain dependent asynchronous operations, where each asynchronous operation is itself a Promise
- ◉ Each **Promise** represents the completion of another asynchronous step in the chain.
- ◉ To chain promises, each **Promise** returns another **Promise**
 - Technically, each `then()` returns a **Promise**
- ◉ Chain promises together using `.then()`
- ◉ Can have multiple `.then()`s
- ◉ Handles rejected state/s with `.catch()` (or `.then(null, callback)`)



**ASYNC /
AWAIT**



async ensures a function
returns a Promise

(a kind of function wrapper)

```
async function f() {  
    return 1;  
}  
f().then( () => { console.log(result); } );
```

Note:

async doesn't execute the function immediately

```
1 console.log("Start");           ← Start
2
3 function g(input) {
4   return input + 1;
5 }
6
7 let a_function = g;
8 console.log(a_function);        ← The result of g(2) is : 3
9 console.log("The result of g(2) is :", g(2));
10
11 async function f(input) {      ← function f(_x) {
12   return input + 1;            return _ref.apply(this, arguments);
13 }                                }
14
15 let another_function = f;
16 console.log(another_function);  ← The result of f(2) is: [object Promise]
17 console.log("The result of f(2) is: ", f(2));    ← The result of f() is: 11
18
19 f(10).then(function(result) {   ← The result of another_function (f()) is: 101
20   console.log("The result of f() is: " + result);
21 });
22
23 another_function(100).then(function(result) { ←
24   console.log("The result of another_function (f()) is: " + result);
25 });
26
```



await forces JS to wait for the Promise to resolve

- `await` is only legal inside an `async` function...
- ... and `async` functions are Promises that commit to a future resolution...
- ... so other code can continue to run



Module dependencies.

module.export / require()



Modular JavaScript files

- ◉ **CommonJS**: one specification for managing module dependencies
 - Others exist e.g. RequireJS, ES2015 AMD (Async Module Definition)
- ◉ Node.js adopted CommonJS
 - To use CommonJS on front-end, you'll need to use Browserify (or similar)
- ◉ A module is defined by a **single JavaScript file**
- ◉ Use `module.exports.*` or `exports.*` (but not both) to expose your module's public interface
- ◉ Values assigned to `module.exports` are the module's public interface
 - A value can be lots of things e.g. string, object, function, array
- ◉ You want to expose something? Add it to `module.exports`
- ◉ Import the module using `require()`



Creating modules & reusing existing module

You can create your own modules

myModule.js

module.exports...

And then reuse that module :

myOtherModule.js

```
var something = require('.../../myModule.js');
```



Dependency management

- npm is a package manager for node
- npm is designed to be node-specific
- `npm install` installs packages suitable for the CommonJS-like dependency management used by Node, i.e. the exports/requires approach



Creating modules & reusing existing module (npm)

You can reuse existing modules provided by the node ecosystem

First, install the existing module through npm

```
> npm install aModule
```

And then reuse that module :

```
myOtherModule.js
```

```
var something = require(aModule);
```

Note the differences in parameters for node and home-grown modules

SENG 365 Week 3

TypeScript and Data Persistence





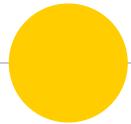
The story so far

- What is a Web Application?
- HTTP
- JavaScript basics
- Asynchronous JavaScript
- Assignment 1



This week

- TypeScript
- Data Persistence in Web Applications



TypeScript

TypeScript Handbook:

<https://www.typescriptlang.org/docs/handbook/intro.html>



Problems with JavaScript

- Dynamically typed
- Type coercion behaves in unexpected ways
 - Poor IDE support
 - <https://blog.campvanilla.com/javascript-the-curious-case-of-null-0-7b131644e274>
- Different ECMAScript versions of the language that are not supported by all browsers

```
var container = "hello";
container = 43;
```

```
null > 0; // false
null == 0; // false
```

```
null >= 0; // true
```



TypeScript

- Developed by Microsoft
- Goal to create safer web code quicker
- Superset of JavaScript
 - All JavaScript code is TypeScript
- Adds:
 - Static typing
 - Type inference
 - Better IDE support
 - Strict null checking



TypeScript files

- TypeScript files use .ts extension
- Any JavaScript file can be converted to TypeScript by simply changing extension from .js to .ts
- The opposite is not true



Static typing in TypeScript

- Basic Types:
 - From JS primitives: boolean, number, bigint, string, array, tuple, object, null, undefined
 - Additional: enum, unknown, any, void, never
- Type declarations for variable
 - Do not change how the code runs
 - Are used by the compiler for type checking
 - Can be explicit or inferred by assignment



Static typing examples

```
let isDone: boolean = false;
```

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
let big: bigint = 100n;
```

```
let color: string = "blue";
color = 'red';
```



Static typing examples

```
let list: number[] = [1, 2, 3];
```

```
let list: Array<number> = [1, 2, 3];
```

```
enum Color {  
    Red,  
    Green,  
    Blue,  
}  
let c: Color = Color.Green;
```



Static typing functions

```
// Parameter type annotation
function greet(name: string) {
  console.log("Hello, " + name.toUpperCase() + "!!");
}

// Would be a runtime error if executed!
greet(42);
```

Argument of type 'number' is not assignable to parameter of type 'string'.

```
function getFavoriteNumber(): number {
  return 26;
}
```

void type is used when no return value



Static typing objects

- Duck typing – based on the shape
- Can be anonymous or named using `interface`

```
function greet(person: { name: string; age: number }) {  
  return "Hello " + person.name;  
}
```

```
interface Person {  
  name: string;  
  age: number;  
}
```

```
function greet(person: Person) {  
  return "Hello " + person.name;  
}
```

Properties can be optional using ?
`age?: number;`



Static typing interfaces, types, and classes

Interface declarations can be used with classes

type is like interface but cannot be extended

See

<https://cutt.ly/NAnFoG9>

```
interface User {  
    name: string;  
    id: number;  
}  
  
class UserAccount {  
    name: string;  
    id: number;  
  
    constructor(name: string, id: number) {  
        this.name = name;  
        this.id = id;  
    }  
}  
  
const user: User = new UserAccount("Murphy", 1);
```



Unions

```
function printId(id: number | string) {
  console.log("Your ID is: " + id);
}
// OK
printId(101);
// OK
printId("202");
// Error
printId({ myID: 22342 });
```

Union types

```
type WindowStates = "open" | "closed" | "minimized";
type LockStates = "locked" | "unlocked";
type PositiveOddNumbersUnderTen = 1 | 3 | 5 | 7 | 9;
```



Unions and typeof

```
function printId(id: number | string) {  
  if (typeof id === "string") {  
    // In this branch, id is of type 'string'  
    console.log(id.toUpperCase());  
  } else {  
    // Here, id is of type 'number'  
    console.log(id);  
  }  
}
```



Strict null checking

```
let x: number = undefined;
```

Generates a compilation error

```
let x: number | undefined;  
if (x !== undefined) x += 1; // this line will compile  
x += 1; // this line will fail compilation
```



Compiling TypeScript

- ◉ Node.JS and browsers do not execute TypeScript
 - It must be compiled to JS first
- ◉ For Node.JS we need to add it to our project:
 - `npm i -D typescript`



TypeScript and Modules

- Node packages can have TypeScript bindings (supports IDE)
- Recall Node uses CommonJS modules (`module.exports`)
- Add `.d.ts` file to package

mymodule.ts

```
const maxInterval = 12;

function getArrayLength(arr) {
  return arr.length;
}

module.exports = {
  getArrayLength,
  maxInterval,
};
```

mymodule.d.ts

```
export function getArrayLength(arr: any[]): number;
export const maxInterval: 12;
```



Data in Web Applications

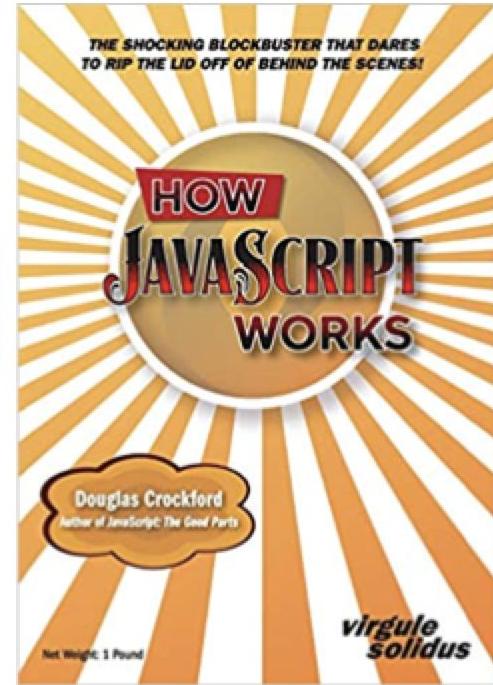
JSON, Relational DB, NoSQL

POJO and JSON

David Crockford's view: <https://json.org/>

A useful tool: <https://json-to-js.com/> with npm version: `npm i -g json-to-js`

Useful tools (but not always accurate): <https://tools.learningcontainer.com/>



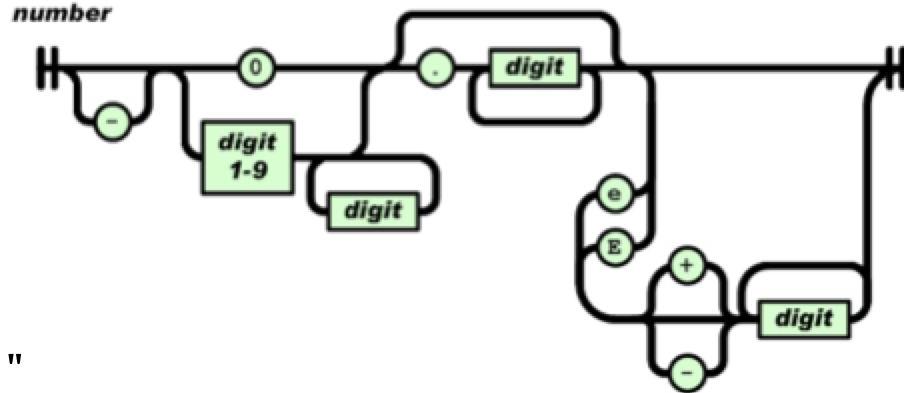


JSON Semi-formal definitions

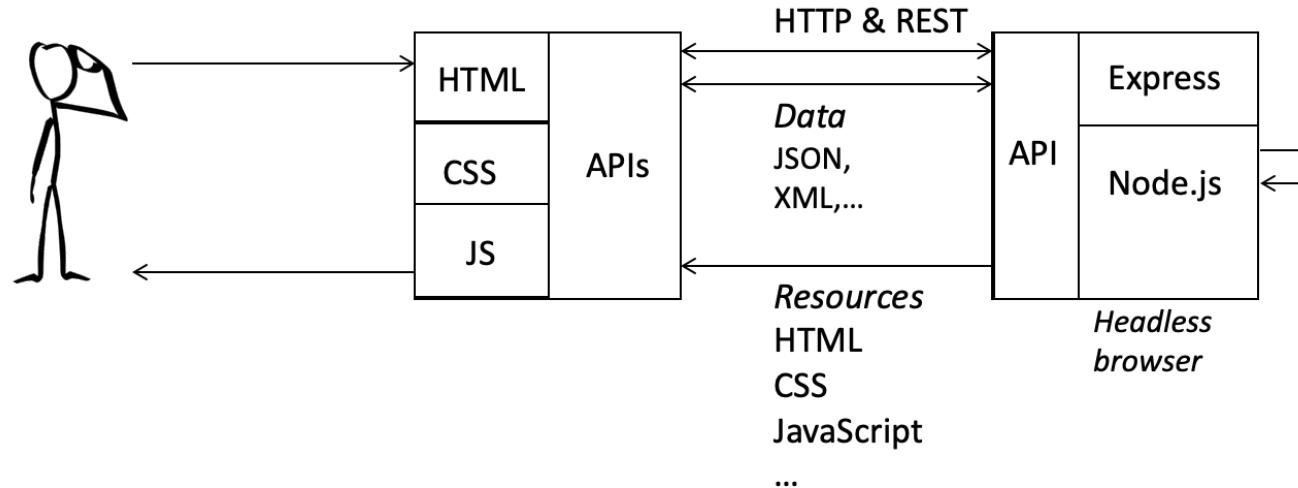
- ◉ JSON is a lightweight **data-interchange** format.
- ◉ A syntax for **serializing data** e.g., objects, arrays, numbers, strings, etc.
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON
- ◉ **Data only**, does not support comments except as a data field
- ◉ Not specific to JavaScript
 - Was originally intended for data interchange between Java and JavaScript
- ◉ No versioning for JSON: why?

JSON: some rules

- All key-names are double-quoted
- Values:
 - Strings are double-quoted
 - Non-strings are not quoted
- Use \ to escape special characters, such as \ and "
- Numbers need to be handled carefully
 - e.g. a decimal must have a trailing digit
 - Correct: 27.0
 - Incorrect: 27.
 - Correct: 27
- Can't – shouldn't – JSONify functions or methods
- See the following for guidance:
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON
 - <https://json.org/>
- And this for an... interesting discussion... on JSON syntax:
 - <https://stackoverflow.com/questions/19176024/how-to-escape-special-characters-in-building-a-json-string>



<https://json.org/>



User	HTTP client	HTTP Server	Database
Human	Machine	Machine	Machine

Reference model



Relational databases

- One of the few cases where a theoretical contribution in academic computer science led innovation in industry
- **Relational model** (E.F. Codd 1970)
 - **Data** is presented as **relations**
 - Collections of **tables** with **columns** and **rows** (tuples)
 - Each **tuple** has the same attributes
 - **Unique key** per row
 - **Relational algebra** that defines operations: UNION, INTERSECT, SELECT, JOIN, etc.

ACID database transactions

- **Atomicity**—“all or nothing” if one part of a transaction fails, then the whole transaction fails
- **Consistency**—the database is kept in a consistent state before and after transaction execution
- **Isolation**—one transaction should not see the effects of other, in progress, transactions
- **Durability**—ensures transactions, once committed, are persistent

“The end of an architectural era?”

- ◉ Traditional RDBMSs
 - ACID properties were requirement for data handling
- ◉ Over the past few decades:
 - Moore’s Law—CPU architectures have changed how they acquire speed
 - New requirements for data processing have emerged
 - Stonebraker et al. (2007), suggest that “one size fits all” DBs not sufficient
 - Still... relational databases are extremely useful in many cases

CAP Theorem

- In distributed computing, **choose two** of:
 - **Consistency**—every read receives the most recent data
 - **Availability**—every read receives a response
 - **Partition tolerance**—system continues if network goes down
- Situation is actually more subtle than implied above
 - Can adaptively choose appropriate trade-offs
 - Can understand semantics of data to choose safe operations

BASE

- Give up consistency (first part of CAP) and we can instead get:
 - **Basic Availability**—through replication
 - **Soft state**—the state of the system may change over time
 - This is due to the eventual consistency...
 - **Eventual consistency**—the data will be consistent eventually
 - ... if we wait long enough
 - (and probably only if data is not being changed frequently)

ACID versus **BASE** example (1/2)

- Suppose we wanted to track people's bank accounts:

```
CREATE TABLE user (uid, name, amt_sold, amt_bought)
CREATE TABLE transaction (tid, seller_id, buyer_id, amount)
```

- ACID transactions might look something like this:

```
BEGIN
```

```
    INSERT INTO transaction(tid, seller_id, buyer_id, amount);
    UPDATE user SET amt_sold=amt_sold + amount WHERE
        id=seller_id;
    UPDATE user SET amt_bought=amt_bought + amount WHERE
        id=buyer_id;
```

```
END
```

ACID versus BASE Example (2/2)

- If we consider amt_sold and amt_bought as *estimates*, transaction can be split:

```
BEGIN
    INSERT INTO transaction(tid, seller_id, buyer_id,
                           amount);
END
BEGIN
    UPDATE user SET amt_sold=amt_sold + amount WHERE
        id=seller_id;
    UPDATE user SET amt_bought=amt_bought + amount WHERE
        id=buyer_id;
END
```

- Consistency between tables is no longer guaranteed
- Failure between transactions may leave DB inconsistent



Key value databases overview

- Unstructured data (i.e., schema-less)
- Primary key is the only storage lookup mechanism
- No aggregates, no filter operations
- Simple operations such as:
 - **Create**—store a new key-value pair
 - **Read**—find a value for a given key
 - **Update**—change the value for a given key
 - **Delete**—remove the key-value pair



Key value databases

Advantages

- Simple
- Fast
- Flexible (able to store any serialisable data type)
- High scalability
- Can engineer high availability

Disadvantages

- Stored data is not validated
 - NOT NULL checks
 - colour versus color
- Complex to handle consistency
- Checking consistency becomes the application's problem
- No relationships—each value independent of all others
- No aggregates (SUM, COUNT, etc.)
- No searching (e.g., SQL SELECT-style) other than via key



Key value database implementations

- Amazon Dynamo (now **DynamoDB**)
- Oracle NoSQL Database, ... (eventually consistent)
- Berkeley DB, ... (ordered)
- Memcache, **Redis**, ... (RAM)
- LMDB (used by OpenLDAP, Postfix, InfluxDB)
- LevelDB (solid-state drive or rotating disk)
- **IndexedDB** (in the browser)



Dynamo Amazon's Highly Available Key-value Store

- Just two operations:
 - `put(key, context, object)`
 - `get(key) → context, object`
- Context provides a connection to DynamoDB
 - contains information not visible to caller
 - but is used internally, e.g., for managing versions of the object
- Objects are typically around 1MiB in size



Dynamo Design

- Reliability is one of the most important requirements
 - Significant financial consequences in its production use
 - Impacts user confidence
- Service Level Agreements (SLAs) are established
- Used within Amazon for:
 - best seller lists; shopping carts; customer preferences; session management; sales rank; product catalog



Redis in memory store

- Whole database is stored in RAM
 - Very fast access
 - Useful for cached data on the server
 - E.g. commonly accessed data from RDBMS can be stored in memory store on same computer as the API server.
- Key-value store where the value can be a complex data structure
 - Strings, Bitarrays, Lists, Sets, Hashes
 - Streams (useful for logs)
 - Binary-safe keys
 - Command set for optimized load, storing, and changing data values



Document databases

- Semi-structured data model
- Storage of documents:
- typically JSON or XML
- could be binary (PDF, DOC, XLS, etc.)
- Additional metadata (providence, security, etc.)
- Builds index from contexts and metadata



Document databases

Advantages

- Storage of raw program types (JSON/XML)
- Indexed by content and metadata
- Complex data can be stored easily
- No need for costly schema migrations
- (Always remember that your DB is likely to need to evolve!)

Disadvantages

- Same data replicated in each document
- Risk inconsistent or obsolete document structures



Document database implementations

- ElasticSearch
- LinkedIn's Espresso
- CouchDB
- MongoDB
- Solr / Apache Lucene
- RethinkDB
- Microsoft DocumentDB
- PostgreSQL (when used atypically)



Graph databases

- **Node** (or **vertex**)—represents an entity
- **Edge**—represents relationship between nodes
- **Bidirectional** (usually illustrated without arrowheads)
- **Unidirectional** (usually illustrated with an arrowhead)
- **Properties**—describe attributes of the node or edge
- Often stored as a **key-value set**
- **Hypergraph** – one edge can join multiple nodes

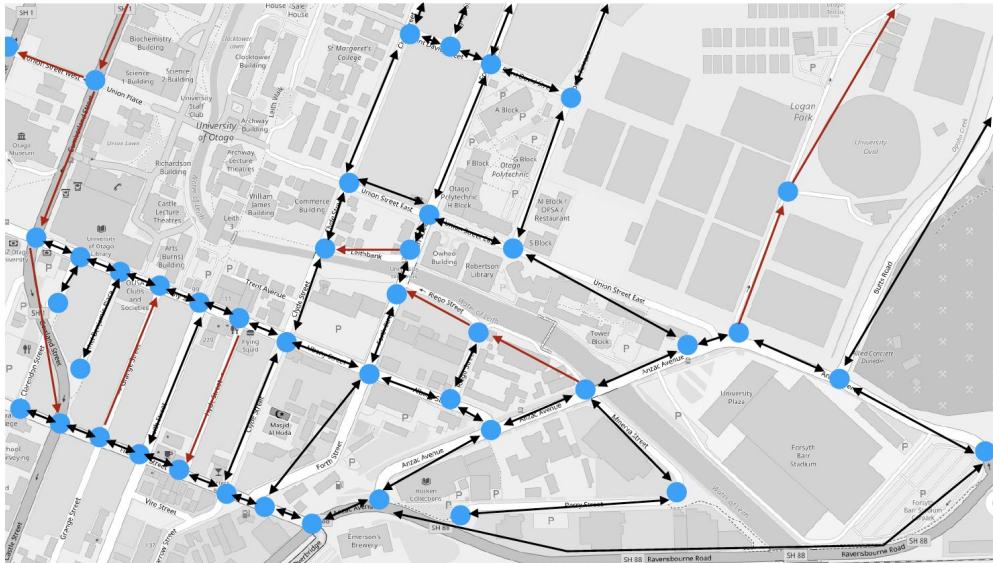
Street map connectivity is a graph

- **Node**

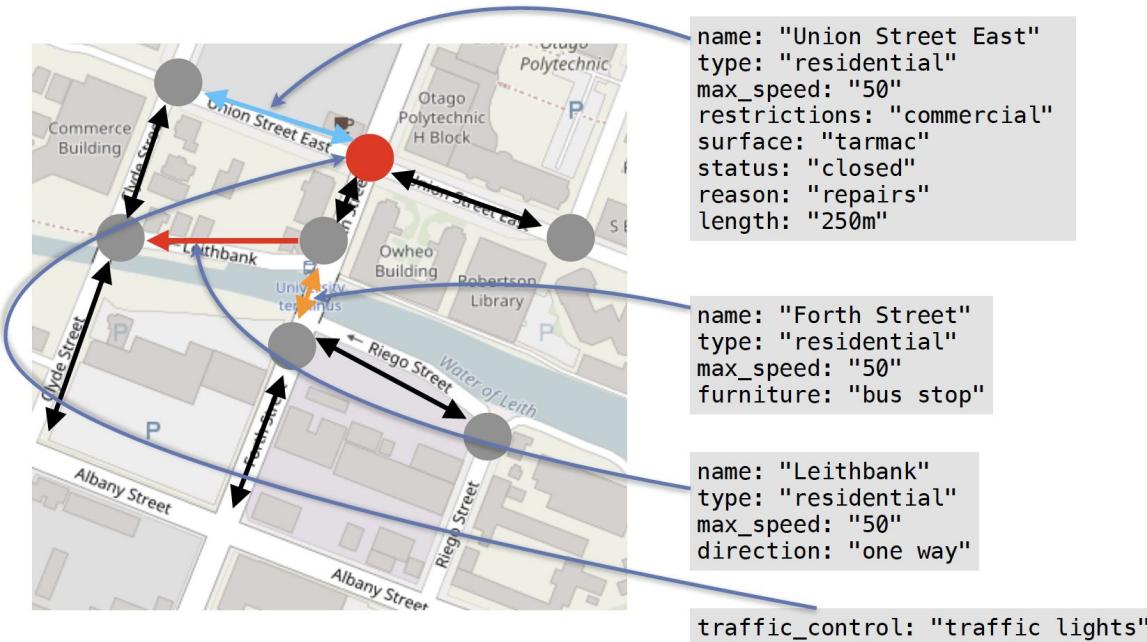
- Traffic junction

- **Edge**

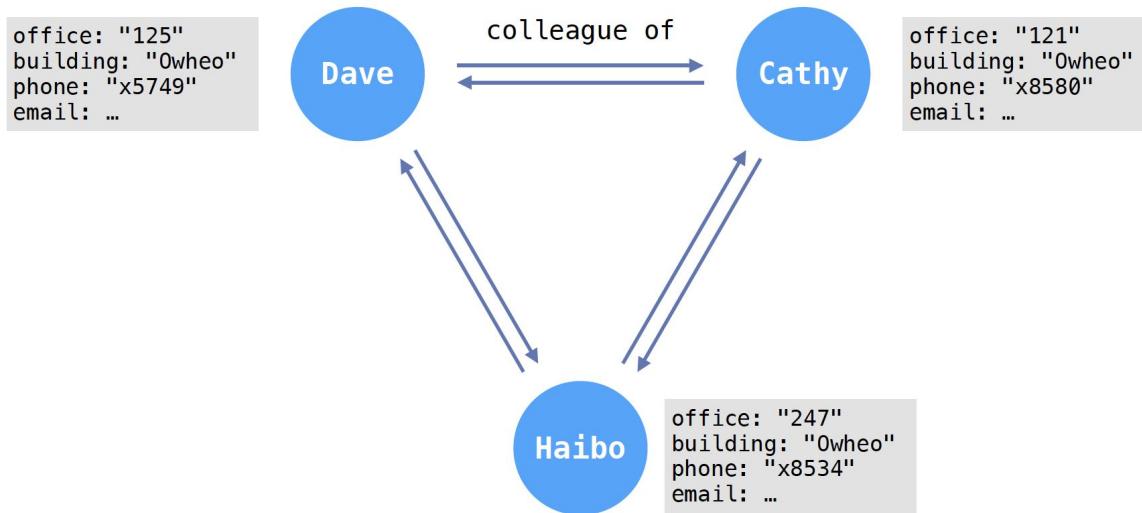
- Shows traffic flow
- Can be uni/bidirectional



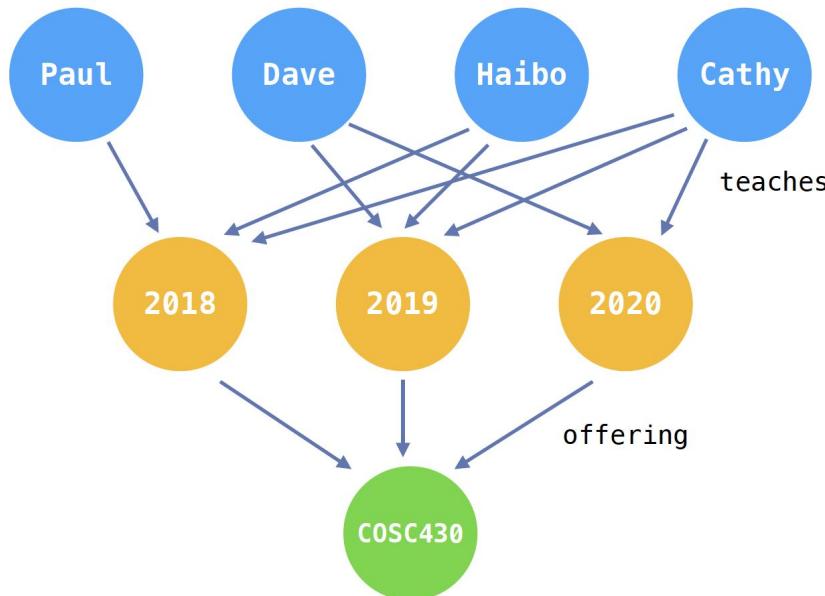
Edges can have properties



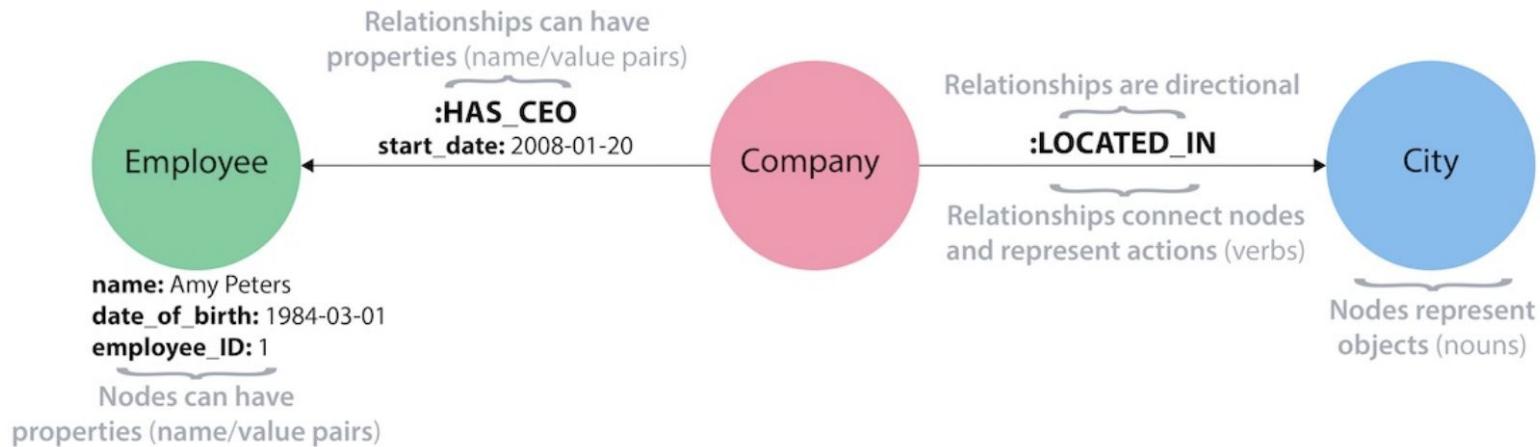
Nodes can have properties



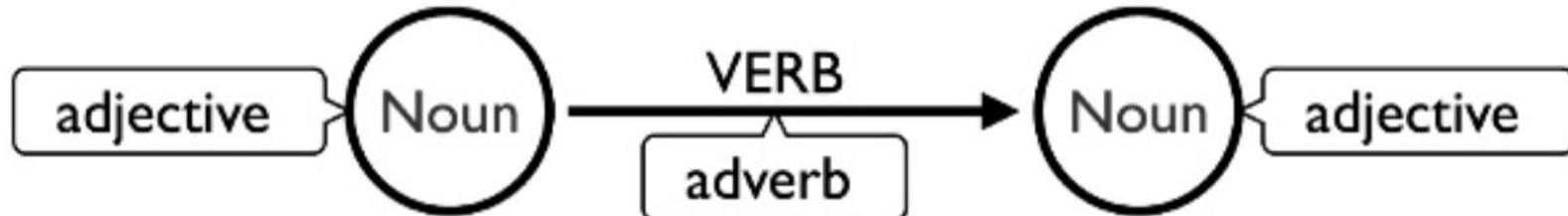
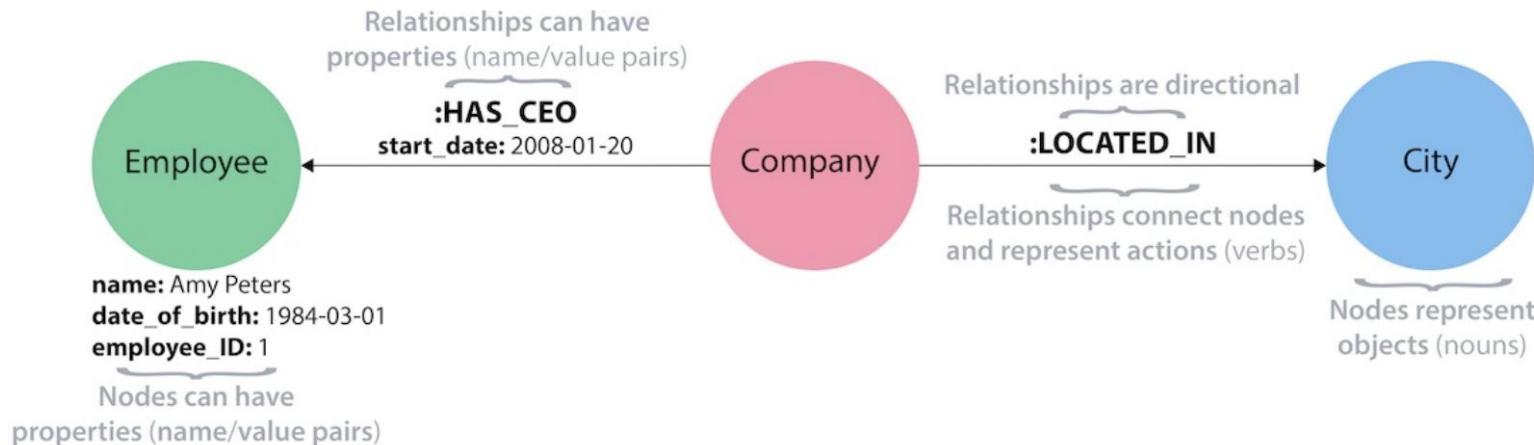
Different types of nodes



Building blocks of property graph model



Building blocks of property graph model



Why do we need graph databases?

- ◉ We can store graphs in RDBMSs, e.g.,
 - Node table
 - Edge table
- ◉ But, joins between nodes and edges are common
 - ... as the number of hops in a graph increases, this becomes increasingly expensive
- ◉ Some problems best suit direct representation in graphs
 - E.g. social graph



Designing graph databases

- Typical mapping from application's data to a graph:
 - **Entities** are represented as **nodes**
 - **Connections** are represented as **edges** between nodes
 - **Connection semantics** dictate **directions** of edges
 - Entity **attributes** become node **properties**
 - Link strength / weight / quality maps to relationship properties
- Other metadata will also be include in property sets
 - e.g., information about data entry and revision



Graph database implementations

- Neo4j
 - <https://neo4j.com/developer/graph-database/>
- Amazon Neptune
- JanusGraph (scalable, distributed graph database)
- ArangoDB
- OrientDB
- RedisGraph (in memory)
- RDF-specific
 - Virtuoso, BlazeGraph, AllegroGraph
- Others... see <https://tinkerpop.apache.org>

Graph DB Query languages

- **Cypher** – developed for neo4j but used by other systems
 - <https://neo4j.com/developer/cypher/>
 - **Declarative language** (like SQL for graph databases)
 - Create, Read, Update, Delete operations on the elements of the graph
 - **Match patterns** in the graph

`(node)-[:RELATIONSHIP]->(node)`

`(node {key: value})-[:RELATIONSHIP]->(node)`

- **Alternatives:**

- SPARQL – querying RDF graphs
- Gremlin – graph traversal language for Apache Tinkerpop
- PGQL – Oracle – mix of SQL SELECT-style with graph matching

Cypher MATCH and RETURN keywords

The screenshot shows the Neo4j browser interface with the following details:

- Query Bar:** neo4j\$ Match (m:Movie) where m.released > 2000 RETURN m limit 5
- Result Summary:** *(5) Movie(5)
- Nodes:** Five orange circular nodes representing movies:
 - The Polar Express
 - Somethi...
 - The Matrix Revol...
 - The Matrix Reloa...
 - Rescue...
- Graph View:** A visualization of the nodes and their connections.
- Navigation:** Includes icons for Graph, Table, Text, and Code, along with standard browser navigation and search tools.
- Footer:** Displaying 5 nodes, 0 relationships.

Cypher MATCH and RETURN keywords

neo4j\$ `MATCH (p:Person)-[d:ACTED_IN]-(m:Movie) WHERE m.released > 2010 RETURN p,d,m`

neo4j\$ `MATCH (p:Person)-[d:ACTED_IN]-(m:Movie) WHERE ...`

Graph Person(4) Movie(1)
*(5) *(4) ACTED_IN(4)

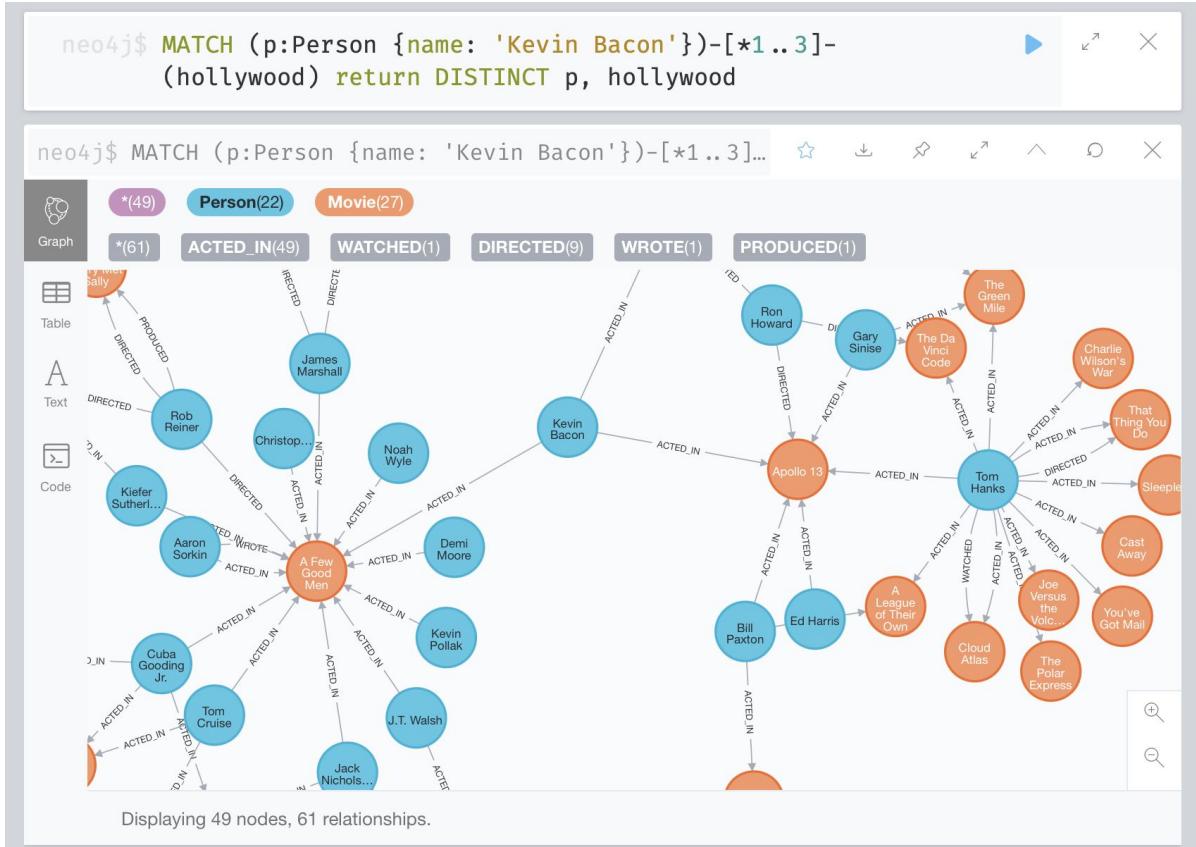
Table

Text

Code

Displaying 5 nodes, 4 relationships.

Complex graph queries



Connecting to neo4j from nodeJS

Shell

[Copy to Clipboard](#)

```
npm install neo4j-driver
```

JavaScript

[Copy to Clipboard](#)

```
const neo4j = require('neo4j-driver')

const driver = neo4j.driver(uri, neo4j.auth.basic(user, password))
const session = driver.session()
const personName = 'Alice'

try {
  const result = await session.run(
    'CREATE (a:Person {name: $name}) RETURN a',
    { name: personName }
  )

  const singleRecord = result.records[0]
  const node = singleRecord.get(0)

  console.log(node.properties.name)
} finally {
  await session.close()
}

// on application exit:
await driver.close()
```

SENG 365 Week 4

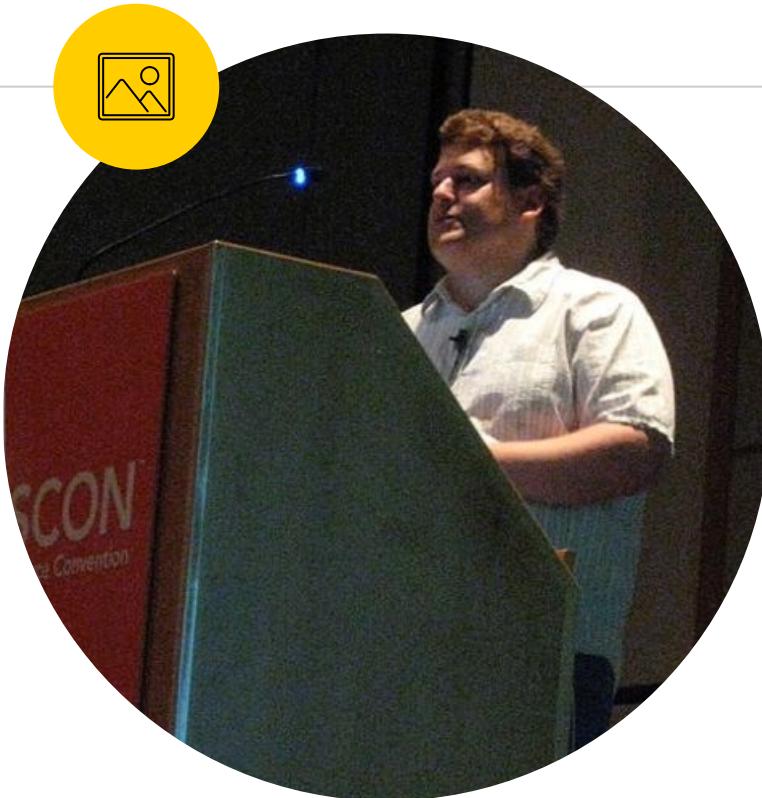
REST, APIs, GraphQL





This week

- RESTful APIs
- Managing state in web applications
- API versioning



REST: **Representational State Transfer**

Developed by Roy Fielding

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.



HTTP

- ◉ HTTP is a **stateless protocol** originally designed for document retrieval
 - HTTP requests and responses are self-contained
 - Not dependent on previous requests and responses
- ◉ HTTP **common commands**
 - GET – retrieve a named resource (shouldn't alter visible server state)
 - HEAD – like GET but only gets headers
 - POST – submits data to the server, usually from an HTML form
- ◉ Some **less common** commands
 - OPTIONS – get supported methods for given resource
 - POST – create a new resource
 - PUT – submit (changes to) a specified resource
 - DELETE – delete a specified resource
 - PATCH – modify an existing resource



Idempotence and safe methods

- ◉ **Safe** methods are HTTP methods that do not modify resources
 - Like read-only methods
- ◉ **Idempotent** methods: the *intended state* (e.g. data changed) is the same whether the method is called once or many times
 - Depends on the current state of the system
- ◉ Clarification
 - This is about best practice & standards
 - You can break these standards, but shouldn't
 - Concerns the server, not the client

HTTP method	Should be safe	Should be Idempotent
GET	Yes	Yes
HEAD	Yes	Yes
PUT (GET first)	No	Yes
POST	No	No
DELETE	No	Yes
PATCH	No	No



Why does idempotency matter? Side-effects

“Idempotency and safety (nullipotentcy) are guarantees that server applications make to their clients ... An operation doesn't automatically become idempotent or safe just because it is invoked using the GET method, if it isn't implemented in an idempotent manner.

A poorly written server application might use GET methods to update a record in the database or to send a message to a friend ... This is a really, really bad design.

Adhering to the idempotency and safety contract helps make an API fault-tolerant and robust.”



CRUD and REST

- To **Create** a resource on the server, you should use POST
- To **Retrieve** a resource, you should use GET
- To change the state of a resource or to **Update** it, ... use PUT
- To remove or **Delete** a resource, you should use DELETE



Bad practice

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

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

- What's this HTTP request doing?



How good are these examples?

1

```
GET /adduser?name=Robert HTTP/1.1
```

2

```
POST /users HTTP/1.1
Host: myserver
Content-Type:
application/xml
<?xml version="1.0"?>
<user>
  <name>Robert</name>
</user>
```

3

```
POST /users HTTP/1.1
Host: myserver
Content-Type:
application/json
{
  "name": "Robert"
}
```



CRUD and REST cont.

- For a partial change, use PATCH (like diff)
- Include only elements that are changing
- Null (value) to delete an element
- See also “best practices”
 - <http://51elliot.blogspot.co.nz/2014/05/rest-api-best-practices-3-partial.html>
- JSON merge patch
 - <https://tools.ietf.org/html/rfc7386>



A REST service is

- Platform-independent
 - Server is Unix, client is a Mac...
- Language-independent
 - C# can talk to Java, etc.
- Standards-based (*runs on top of HTTP*), and
- Can easily be used in the presence of firewalls



A **REST service** is

- RESTful systems typically, but not always:
 - communicate over the Hypertext Transfer Protocol (HTTP)
 - with the same HTTP verbs (GET, POST, PUT, DELETE, etc.) used by web browsers
 - to retrieve web pages and send data to remote servers.
- RESTful apps are (a subset of) web apps



(HTTP | REST) **resources**

- ◉ Nouns not verbs (because resources are things not actions)
- ◉ Instances or Collections
- ◉ Resource instance typically identified by :id
 - Integer
 - sequential numbering issues, reuse of ids (generally bad idea)
 - some systems have troubles with long integers (>53 bits in JavaScript)
 - UUID
- ◉ Hypertext As The Engine Of Application State (HATEOS)
 - <http://restcookbook.com/Basics/hateoas/>



REST **features**

- REST offers **no built-in security features**, encryption, session management, QoS guarantees, etc.
- These can be added by building on top of HTTP
- For encryption, REST can be used on top of HTTPS (secure sockets)



REST vs **SOAP**

- REST displaced SOAP, because...
- ... REST is considerably easier to use
 - e.g. SOAP has a 'heavy' infrastructure
- ... works nicely with AJAX / XHR
 - e.g. XML is verbose; JSON is more concise
- ... has some network advantages
 - accepted through firewalls



Stateless requests

- A complete, independent request doesn't require the server to retrieve any kind of application context or state.
- A RESTful Web service application includes within the HTTP headers and body of a request: all of the parameters, context, and data needed by the server-side component to generate a response.
- The entire resource is returned, not a part of it.
- Statelessness:
 - Improves Web service performance
 - Simplifies the design and implementation of server-side components...
 - because the absence of state on the server removes the need to synchronize session data with an external application.



RESTful URLs

- ◉ **Hide the server-side scripting technology** file extensions (.jsp, .php, .asp), if any, so you can port to something else without changing the URLs
 - Hide all implementation details!
- ◉ **Be consistent** in the singularity / plurality of resource names
 - Use **singular** or **plural**, but don't mix them
 - /student & /course; not /student and /courses
- ◉ Keep everything **lowercase**
- ◉ Substitute **spaces for hyphens**
- ◉ Instead of using the 404/Not Found code if the request URI is for a partial path, always provide a **default page or resource** as a response



Issues with REST

- Tightly coupled to HTTP
- Request-response
 - Can't push/alert or broadcast
- Multiple request-responses needed
 - Implied tree-structure
 - Underfetching and overfetching
 - Latency increases for full set of request-response



State and Statelessness



Mini-overview

- State timescales
- Session (state) information
- GET parameters e.g. after the ?
- HTTP bodies & cookies
 - Types
 - Sequence
 - Limitations
 - Session IDs



State timescales

- ◉ Individual HTTP request (stateless)
- ◉ Business transaction
- ◉ Session
- ◉ Preferences
- ◉ Record state





Session state information

- For web applications (in contrast to public web pages) there is a need to **maintain stateful information** about the client
- The application (client & server) needs to maintain consistency on **which client** (authentication) and **what actions** (authorisation)
- Longer-lived transactions
 - Anything that's built up of **multiple HTTP calls** (stateless)
 - May be stored on server, or on client (or synchronized)



GET ? parameters

- Maintain some kind of **session variable** in the **parameter** to the HTTP request
- Variable does not contain the username and password, but a unique ('random') identifier
- Include variable as a parameter in each network request e.g.,
`GET www.example.com?sessionid=<var>`
- Why is this 'bad practice'?
- Why may something like this be needed, at times?



Cookies

- Use cookies to maintain session information
- The server issues a unique ('random') identifier in the cookie to the client, for that username & password
- Client sends back the cookie with **each** network request to the server
- e.g. in the POST data
- Note that the username and password are not sent (once the user is logged on)



Cookies

- ◉ A **small piece of data** initially sent by the server to the client.
 - Comprises **name-value pairs**
 - Also has **attributes** (that are not sent back to the server)
 - **Expiry 'date'** (duration)
- ◉ Used to maintain state information
 - e.g. items in a shopping basket
(although this example may be better kept on the server)
 - e.g. browser activity such as a 'path' through a registration process

Types of Cookies

First-party cookie	A cookie set by the server to which the browser primarily connects.
Session cookie	Exists only for the duration of that browser session, and the browser typically deletes the cookie
Persistent cookie (aka tracking cookie)	Persistent data. The cookie is not deleted when the browser closes. Can be used by advertising to track user behaviour. Can be used to store credentials e.g. log in details.
Secure cookie	A cookie that can only be transmitted over an encrypted connection, such as HTTPS.
HTTPOnly cookie	Can only be transmitted through HTTP/S, and are not accessible through non-HTTP APIs such as JavaScript.
Third-party cookie	Cookies set by third-parties that serve content to the page e.g. advertising.



Cookie sequence

- Request from browser

```
GET /index.htm HTTP/1.1  
Host: www.example.com
```

- Response from server

```
HTTP/1.1 200 OK  
Content-type: text/html  
Set-cookie: sessionToken=a1b2c3; Expires = [dat]
```

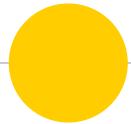
- Follow-up request from browser

```
GET /profile.htm HTTP/1.1  
Host: www.example.com  
Cookie: sessionToken=a1b2c3
```



Limitations of cookies

- Each browser maintains its own ‘cookie jar’
- A cookie does not identify a person
- A cookie identifies the combination of:
 - User account
 - Web browser
 - Device
- A cookie requires that the browser is cookie-enabled and is set to allow cookies



API Versioning



API versioning: overview

- Compatibility between API provider and API consumer
- Semantic versioning
- Specifying API versions in HTTP requests (and handling of those)
- JSON
- Publishing APIs

Interesting (though a bit dated) discussion at SO:

<https://stackoverflow.com/questions/389169/best-practices-for-api-versioning>



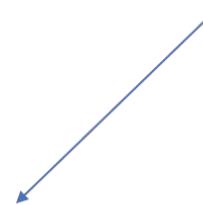
Why version APIs?

- Maintain compatibility of APIs with API consumers, as APIs change
- Add / amend functionality/services
- Performance improvements e.g., reduce the number of HTTP requests
- Market testing of ideas e.g., A/B testing
- System testing e.g. dev, test, prod
- Subsets of clients, e.g.,
 - B2B vs B2C
 - Geographical / political region
- Rollback of (unacceptable) API
- From client perspective:
 - API is backward compatible if client can continue through service changes
 - Forward compatible if client can be changed without needing service change



Semantic versioning: MAJOR.MINOR.PATCH

1. Caching - want different versions cached differently
2. Number or name?
 - If number, what format? Semver? Counter?



Semantic Versioning – semver.org

Given a version number MAJOR.MINOR.PATCH, increment the:

- MAJOR version when you make **incompatible** API changes,
- MINOR version when you add functionality in a backwards-compatible manner, and
- PATCH version when you make backwards-compatible bug fixes.



Specifying API version in HTTP request

- ◉ Query parameter
 - ?v=xx.xx or ?version=xx.xx or ?Version=2015-10-01
 - e.g., Amazon, NetFlix
- ◉ URI
 - api/v1/
 - e.g. Facebook: <https://graph.facebook.com/v2.2/me/adaccounts>
 - Semantically messy (implies version refers to version of object)
- ◉ Header
 - Accept header – hard to test – can't just click on link or type URL
 - Custom request header – duplicates Accept header function
 - E.g., GitHub: <https://developer.github.com/v3/media/>



Publishing an API

- **Documentation** – current, accurate, easy, guide/tutorial/directed (management tool generated)
- **Direct access** (no SDK required)
 - e.g., through Postman or curl (say, curl -L
`http://127.0.0.1:4001/v2/keys/message-XPUT -d value="Hello world")`
- **SDKs/Samples in developer preferred languages**
 - Any SDK is just libraries to access REST/SOAP API, nothing more. Potentially an impediment to simply making use of the straight API.
 - Straightforward install and use
- **Free/Freemium** use for developers
- **Instant API keys**
- **Simple sandbox** to try things out for developers
- Before API available, establish **API landing page** on web to discover interest and potential user types

SENG 365 Week 5

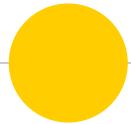
GraphQL and API Testing





This week

- More info on assignment
- GraphQL
- API testing



More info on Assignment 1



Getting started

- Some of the endpoints rely on other endpoints
- E.g. you cannot do a POST request to create a new film until you have logged in
 - You will get a 401 unauthorized error
- Where to start?
 - Implementing user endpoints
 - Other GET requests that do not rely on user authentication



Example routes code (in JS)

```
1 const venues = require('../controllers/venues.controller');
2 const authenticate = require('../middleware/authenticate');
3
4 module.exports = function (app) {
5     app.route(app.rootUrl + '/venues')
6         .get(venues.search)
7         .post(authenticate.loginRequired, venues.create);
8
9     app.route(app.rootUrl + '/venues/:id')
10        .get(venues.viewDetails)
11        .patch(authenticate.loginRequired, venues.modify);
12
13    app.route(app.rootUrl + '/categories')
14        .get(venues.getCategories);
15};
```



Example controller code (in JS)

```
45 exports.viewDetails = async function (req, res) {
46     try {
47         const venue = await Venues.viewDetails(req.params.id);
48         if (venue) {
49             res.statusMessage = 'OK';
50             res.status(200)
51                 .json(venue);
52         } else {
53             res.statusMessage = 'Not Found';
54             res.status(404)
55                 .send();
56         }
57     } catch (err) {
58         if (!err.hasBeenLogged) console.error(err);
59         res.statusMessage = 'Internal Server Error';
60         res.status(500)
61             .send();
62     }
63 };
```



Example model code (in JS)

```
142 exports.viewDetails = async function (venueId) {
143     const selectSQL = 'SELECT venue_name, city, short_description, long_description, date_added, ' +
144         'address, latitude, longitude, user_id, username, Venue.category_id, category_name, category_description ' +
145         'FROM Venue ' +
146         'JOIN User ON admin_id = user_id ' +
147         'JOIN VenueCategory ON Venue.category_id = VenueCategory.category_id ' +
148         'WHERE venue_id = ?';
149
150     try {
151         const venue = (await db.getPool().query(selectSQL, venueId))[0];
152         if (venue) {
153             const photoLinks = await exports.getVenuePhotoLinks(venueId);
154             return {
155                 'venueName': venue.venue_name,
156                 'admin': {
157                     'userId': venue.user_id,
158                     'username': venue.username
159                 },
160                 'category': {
161                     'categoryId': venue.category_id,
162                     'categoryName': venue.category_name,
163                     'categoryDescription': venue.category_description
164                 },
165                 'city': venue.city,
166                 'shortDescription': venue.short_description,
167                 'longDescription': venue.long_description,
168                 'dateAdded': venue.date_added,
169                 'address': venue.address,
170                 'latitude': venue.latitude,
171                 'longitude': venue.longitude,
172                 'photos': photoLinks
173             };
174         } else {
175             return null;
176         }
177     } catch (err) {
```



Authentication

```
27 exports.loginRequired = async function (req, res, next) {
28     const token = req.header('X-Authorization');
29
30     try {
31         const result = await findUserIdByToken(token);
32         if (result === null) {
33             res.statusMessage = 'Unauthorized';
34             res.status(401)
35                 .send();
36     } else {
37         req.authenticatedUserId = result.user_id.toString();
38         next();
39     }
40 } catch (err) {
41     if (!err.hasBeenLogged) console.error(err);
42     res.statusMessage = 'Internal Server Error';
43     res.status(500)
44         .send();
45 }
46 };
```



Some advice #1

- We are testing against the **API specification**
- Be **clear** about what you are trying to achieve with **each function**
- Ensure npm packages have been added to **package.json**
- Remember to do an **npm install** when doing a clean test deploy
- Be aware of your **npm dependencies**
 - Dependencies in **dev** vs dependencies for **prod** e.g. nodemon
- Remember the prefix to the URL, **/api/v1**
- Check against the **latest version** of the API specification
 - Am I using the correct parameters? Are they formatted correctly?



Some advice #2

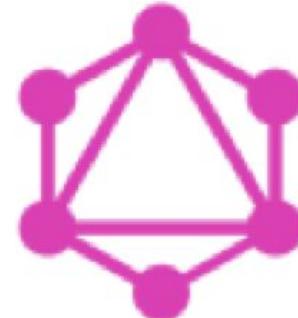
- How are you handling photos?
 - Do you need to add a photo directory to git?
 - `/storage/photos` is tracked, but the contents are not...
 - Make sure that you use correct mime type for images, e.g. `image/png`
 - Use `mz/fs` (or similar) to handle file reading and writing of image files from filesystem:
<https://www.npmjs.com/package/mz>
- Test against the reference server



Some advice #3

- Encrypting password in database
 - Best practice to use existing library, e.g. `bcrypt`
 - <https://www.npmjs.com/package/bcrypt>
 - We will test that you are not storing the password in plain text
- Generate authentication token
 - Several options: e.g. `rand-token`:
 - <https://www.npmjs.com/package/rand-token>

{REST} GraphQL



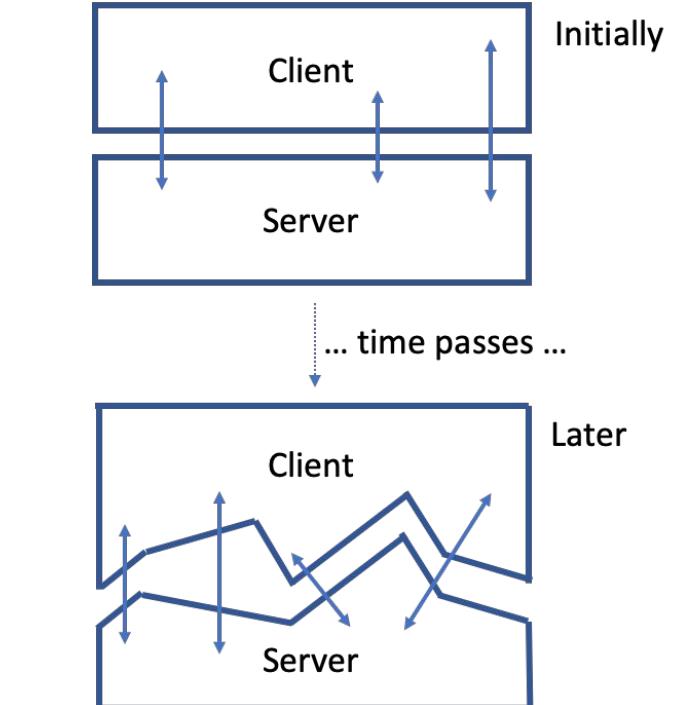


Endpoints and client views

- Endpoints tend to be designed and structured according to the views expected to be needed on the front-end
 - e.g. we design request parameters (query & body) and the response's JSON structure to fit the view
- That's an efficient design...

... EXCEPT THAT...

- Views change
- Users want different information, more information, less information, more and less views
- The fit between endpoint/s and view/s therefore disintegrates





RESTful APIs and their limitations

- Fetching complicated data structures requires **multiple round trips** between the client and server.
- For mobile applications operating in **variable network conditions**, these multiple roundtrips are highly undesirable.

An example set of requests
`/auctions/{id}`
`/auctions/{id}/bids`
`/users/...`
`/auctions/{id}/photos`



Overfetching and underfetching

Overfetch: Download more data than you need

- e.g. you might only need a list of usernames, but `/users` downloads (as a JSON object) more data than just usernames
- And endpoint provides more than you need

Underfetch: download less than you need so must then do more (the n+1 problem)

- e.g. you need a list of most recent three friends for a username, so for each item in `/users` you need to get information from `/user/friends`, but then only take the first three entries



RESTful APIs and their limitations cont.

- REST endpoints are usually **weakly-typed** and lack machine-readable metadata.

An example of the confusion
`eventStartTime integer`
Why integer and not Date?
Mapping from integer to date and time?

POST /events API, is `startingTime` the same as the `event_startingtime` in the `events` table?



GraphQL

- A specification for:
 - How you specify data (cf. strong-typing)
 - How you query that data
- There are reference implementations of the GraphQL specification
 - <https://github.com/graphql/graphql-js> (Node.js)
- Extra lab on LEARN (not pre-req for assignment)



GraphQL simple example

Comments

- Character is a GraphQL Object Type that has fields
- name and appearsIn are the fields
- String is a scalar type (a base type that's irreducible)
- [Episode] ! is an array [] that's non-nullable (due to the !)
- Each type Query specifies an entry point for every GraphQL query.

Example (of API)

```
type Character {  
    name: String!  
    appearsIn: [ Episode ]!  
}  
  
type Query {  
    hero: Character  
}
```



GraphQL vs REST

GraphQL

- Define objects and fields that can be query-able
- Define **entry points** for a query
- The client application can dynamically ‘compose’ the content of the query
- A much more flexible interface to the server side.

REST

- **Endpoints** that are set and inflexible
- Pre-defined fixed endpoints that
 - Require pre-defined inputs
 - Return pre-defined data structures
- Those endpoints are then ‘set’...
 - ... until version x.y.z of the API



GraphQL vs REST response codes

GraphQL

- All GraphQL queries return 200 response code, even errors.
 - E.g. malformed query, query does not match schema, etc.
- Errors are returned in user-defined field
- Network errors can still return 4xx/5xx
 - E.g. GraphQL server is down

REST

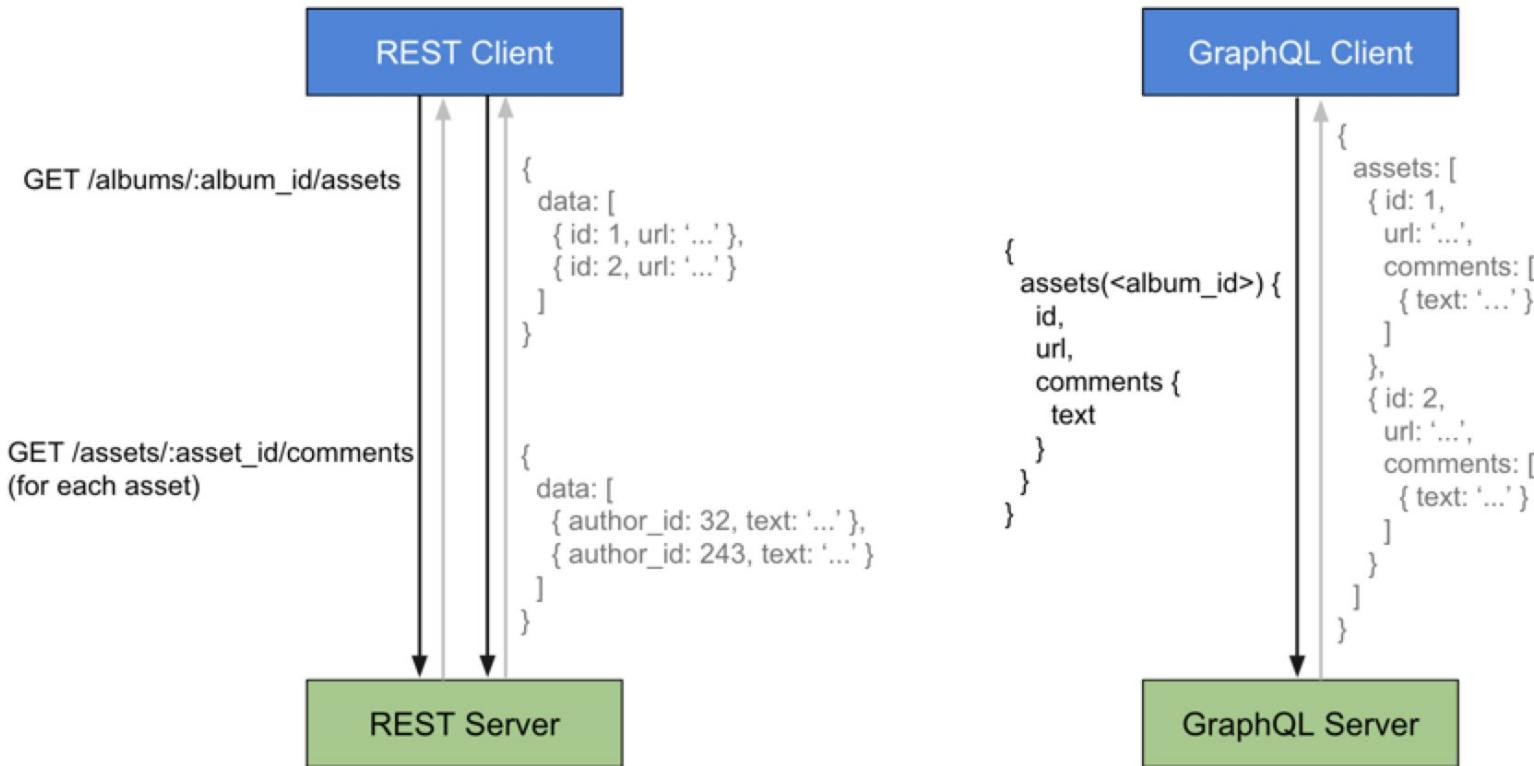
- HTTP response code indicates success / error
- 2xx, 4xx, 5xx, etc.

```
{  
  "data": {  
    "getInt": 12,  
    "getString": null  
  },  
  "errors": [  
    {  
      "message": "Failed to get string!",  
      // ...additional fields...  
    }  
  ]  
}
```



GraphQL and data

- ◉ Does not require you to think in terms of graphs
 - You think in terms of **JSON-like structures** for a query (see earlier slide)
- ◉ Is not querying the database directly
 - Rather is a '**language**' (**specification**) for composing queries to a server
- ◉ Still requires some kind of **pre-defined data** and **queries** on the server-side
 - Objects, fields and allowable queries
 - But these pre-definitions are more 'atomic' in their nature



REST vs GraphQL requests



GraphQL uses GET and POST

GET

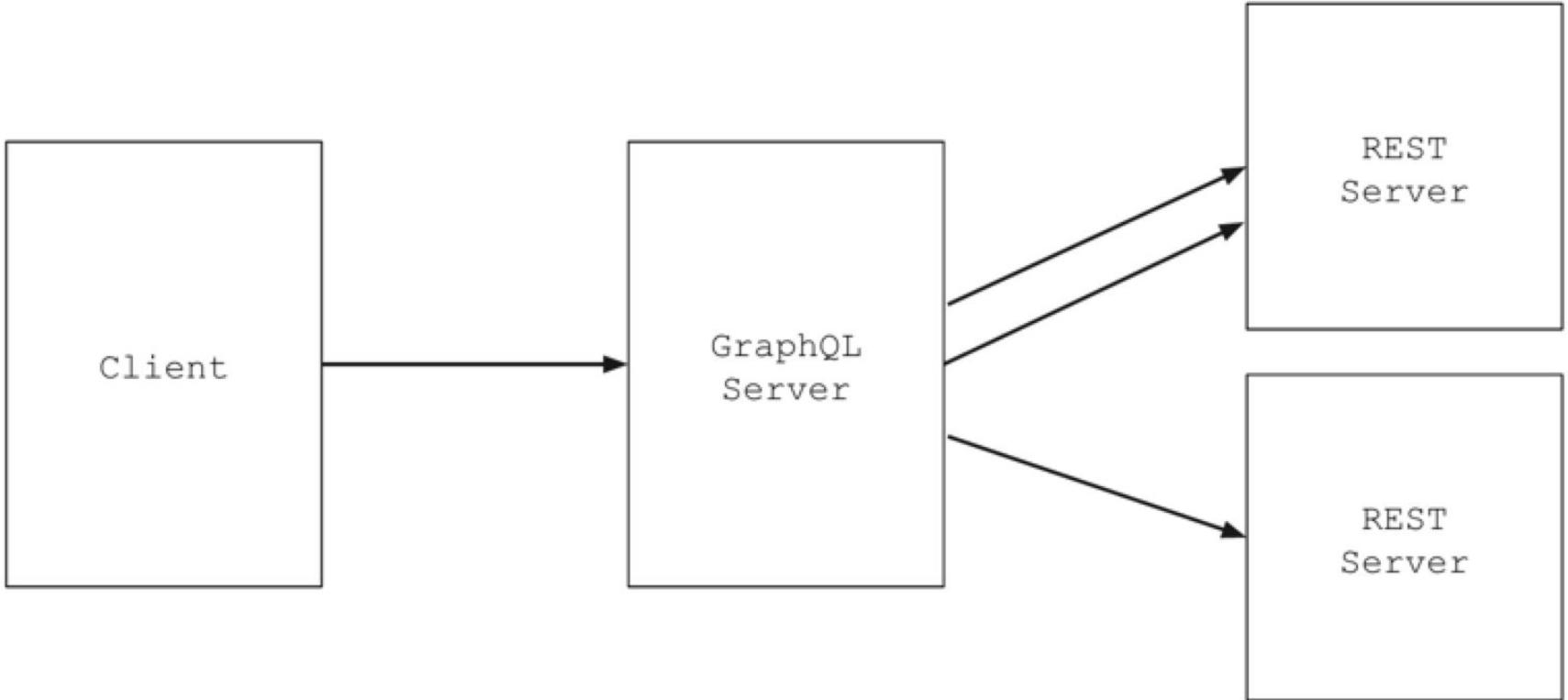
GraphQL query is specified using the URL query parameters

```
http://myapi/graphql?query={me{name}}
```

POST

Specify the query in the HTTP body, using JSON

```
"query": "...",
"operationName": "...",
"variables": {
  "myVariable": "someValue",
  ...
}
```



GraphQL can sit in front of REST API(s)





GraphQL additional resources

- GraphQL Introduction
 - <https://graphql.org>
- Express + GraphQL
 - <https://www.npmjs.com/package/express-graphql>
- Apollo GraphQL Server
 - <https://www.apollographql.com/docs/apollo-server/>
- From REST to GraphQL
 - <https://0x2a.sh/from-rest-to-graphql-b4e95e94c26b>



Automated API Testing



API testing

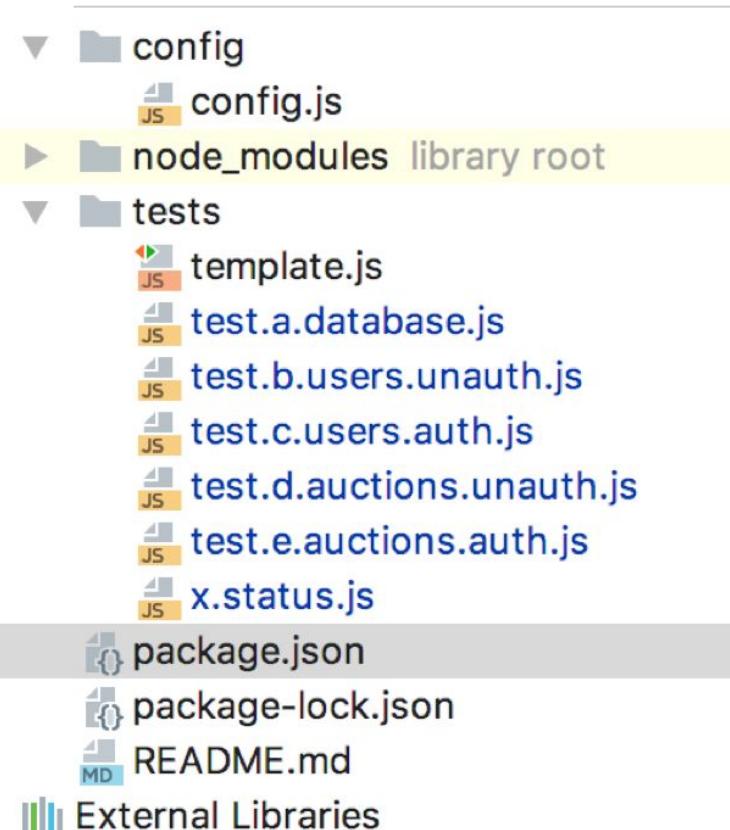


- **Postman** tests – Javascript console for testing API endpoints
- **Mocha + Chai**
 - Packages for automated testing Node.JS code
 - Can be used with continuous integration/deployment (CI/CD) environments, such as GitLab Runner
 - Mocha – Asynchronous testing environment
 - Chai – Assertion library
 - <https://www.digitalocean.com/community/tutorials/test-a-node-restful-api-with-mocha-and-chai>



Mocha + Chai setup

- Can have one test file
- For multiple test files:
 - Mocha runs test files in order of occurrence (depends on OS's file systems)
 - Depends on how defined in `package.json`
- Each test (even multiple tests in one test file):
 - Is intended to be independent
 - Runs asynchronously





Separate test project

In package.json

```
...
"scripts": {
    "start": "mocha ./tests/test.*.js --reporter spec --log-level=warn",
    "test": "mocha ./tests/test.a.file.js --reporter spec
--log-level=warn",
},
...
...
```

Given the above:

`npm start` will run all my test files

`npm test` will run a particular test file (that I have specified)



Asynchronous behavior when testing

- You can setup pre- and post-conditions
 - `before()`, `beforeeach()`, `after()`, etc
- Mocha, Chai and Chai-HTTP can handle callbacks, and Promises (and `async/await`)
 - Don't get these mixed up in a given test
 - Avoid the use of `return` together with `done()`



A single test using a Promise

```
describe('Test case:/POST/login with parameters in query string', () => {
  it('Should return 200 status code, id and authorisation token', function () {
    return chai.request(server_url)
      .post('/users/login')
      .query(
        {
          username: 'testUsername4',
          email: "user4@testexample.com",
          password: "testpassword"
        }
      )
      .then(function(res) {
        expect(res).to.have.status(200);
        expect(res).to.be.json;
        expect(res.body).to.have.property('id');
        expect(res.body).to.have.property('token');
        authorisation_token = res.body['token']; //use in subsequent test
        user_id = res.body['id']; //use in subsequent test
      })
      .catch(function (err) {
        expect(err).to.have.any.status(400, 500);
        throw err; // there is any error
      });
    });
});
```



A single test using old-style callbacks

```
describe('Test case: ' + test_case_count + ': POST /users', () => {
  it('Callback with done(): Should return 400 or 500 as there was a duplicate entry', (done) => {
    chai.request(server_url)
      .post('/users')
      .send(
        {
          username: "testUsername4",
          givenName: "testGivenName",
          familyName: "testFamilyName",
          email: "user@testexample.com",
          password: "testpassword"
        }
      )
      .then(function (res) {
        expect(res).to.have.any.status(201); // is this line really needed?
        done(new Error("Status code 201 returned unexpectedly")); //test completed but failed
      })
      .catch(function (err) {
        expect(err).to.have.any.status(400,500);
        done(); // test completed as it should / as it was expected to complete
      });
  });
});
```



Tests are **asynchronous**

- ◉ With the assignment, for example, you would be testing a **network request** to a server that is then making a **database request**
- ◉ **You don't know when** the network request or the database request will **complete**
 - Therefore you don't know when the test will complete
- ◉ You **shouldn't assume** that **test n+1** will complete before **test n+2** starts
 - Which is why you have `before()`, `beforeeach()`, `after()` etc.
- ◉ Need to be careful with the **dependencies between tests**
- ◉ Need to be careful on how you **report the progress of tests**, because the report **may not output synchronously** with completion of the test itself



Testing for **expected success** and **expected failure**

- ◉ Often we test to corroborate that something completes as we expected
 - e.g. that `user/login` is successful as expected: the user logs in
- ◉ We also need to test that the system rejects/doesn't complete as expected
 - e.g. that `user/login` is unsuccessful as expected: the user is not logged in
- ◉ Need to think carefully about:
 - `.then()`, `catch()`, `done()`, `done(err)`, and/or `throw err;`



Passing tests does not always mean intended behavior

	Actual behavior: successful	Actual behavior: failed
Intended behavior: successful	The test passed	The test failed
Intended behavior: failure	The test failed	The test passed

SENG 365 Week 6

Security and Intro to client side





This week

- Primer on Security issues
- Introduction to client-side technologies and concepts

- Assignment 1 queries



Primer on Security for Web Apps



OWASP

The Open Web Application Security Project

<https://owasp.org>

<https://www.meetup.com/OWASP-New-Zealand-Chapter-Christchurch/>



Open Web Application Security Project (updated 2020)

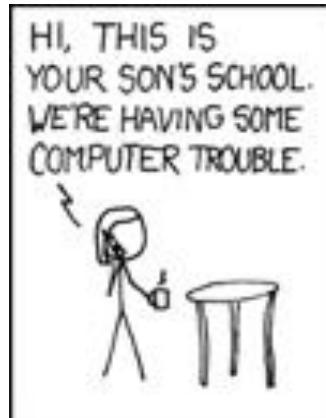
Top 10 security problems

1. Injection
2. Broken Authentication
3. Sensitive Data Exposure
4. XML External Entities (XXE)
5. Broken Access Control
6. Security Misconfiguration
7. Cross-Site Scripting (XSS)
8. Insecure Deserialization
9. Using Components with Known Vulnerabilities
10. Insufficient Logging and Monitoring

Injection

Injection flaws allow attackers to relay malicious code through an application to another system e.g. SQL injection.

https://www.owasp.org/index.php/Injection_Flaws





Injection

- **Any time** an application uses **an interpreter of any type** there is a danger of introducing an injection vulnerability.
- When a web application passes information from an **HTTP request** through as part of an external request, it must be carefully scrubbed
- **SQL injection** is a particularly widespread and dangerous form of injection...



Command Injection

- Assume that we have a Java class (on the server) that gets input from the user via a HTTP request, and that class goes on to use the Java Runtime object to make an MS-DOS call

```
Runtime rt = Runtime.getRuntime();
// Call exe with userID
rt.exec("cmd.exe /C doStuff.exe " + "-" + myUid);
```



Command Injection

```
Runtime rt = Runtime.getRuntime();
// Call exe with userID
rt.exec("cmd.exe /C doStuff.exe " + "-" + myUid);
```

When myUid = Joe69, we'd get the following OS call:

```
> doStuff.exe -Joe69
```

When myUid = Joe69 & netstat -a, we'd get:

```
> doStuff.exe -Joe69
```

```
> netstat -a // "&" is command appender in MS-DOS
```



Basis of all injections...

- ◉ All injection flaws are **input-validation** errors.
 - i.e. you're not checking the input properly
- ◉ Input is **not just text fields**
- ◉ All **external input** is a source of a **threat**.
 - The input contains the data with the threat
 - Examples: text fields, list boxes, radio buttons, check boxes, cookies, HTTP header data, HTTP post data, hidden fields, parameter names and parameter values



Validate ... and re-validate

- ◉ An input field is likely to be **validated on the client side**, e.g., that an IDNumber textfield contains a number rather than a number and malicious code.
 - What happens to that data **between the client and the server?**
- ◉ Sometimes it's **hard to validate**.
- ◉ Sometimes there are **multiple clients** e.g. you're offering a public web service to clients.
- ◉ Should you safely assume that the input data is valid because it was **previously validated**?



Authentication

- Definitions
 - **Authentication**: establish claimed identity
 - **Authorisation**: establish permission to act
 - Authentication precedes authorisation
- **Why** authenticate?
 - Control access to resources
 - Log user activity
 - Non-repudiation
- **How** can we authenticate?
 - Three factors: something you know, have, or are



Securing passwords

- ◉ Hash username and password
- ◉ Require users to change their passwords regularly
- ◉ Use multi-factor authentication
 - Username & password
 - Code sent by phone
- ◉ Salt the username and password
 - Add additional elements to the ID information
- ◉ Use HTTPS (HTTP + TLS)

HTTP is a “stateless” protocol

- Means credentials have to go with every request
- Should use SSL for everything requiring authentication

Session management flaws

- SESSION ID used to track state since HTTP doesn't
 - and it is just as good as credentials to an attacker
- SESSION ID is typically exposed on the network, in browser, in logs, ...

Beware the side-doors

- Change my password, remember my password, forgot my password, secret question, logout, email address, etc...

Typical Impact

- User accounts compromised or user sessions hijacked

Verify your architecture

- Authentication should be simple, centralized, and standardized
- Use the standard session id provided by your container
- Be sure SSL protects both credentials and session id at all times

Verify the implementation

- Forget automated analysis approaches
- Check your SSL certificate
- Examine all the authentication-related functions
- Verify that logoff actually destroys the session
- Use OWASP's WebScarab to test the implementation

Follow the guidance from

- https://www.owasp.org/index.php/Authentication_Cheat_Sheet



DOM-based XSS Injection

- DOM Based XSS allows an attacker to use the **Document Object Model** to introduce **hostile code** into vulnerable client-side JavaScript embedded in many pages.
- **Browser interprets** .js, HTML, the DOM etc
- DOM based XSS is extremely difficult to mitigate against because of its **large attack surface** and **lack of standardization** across browsers.
 - **Untrusted data** should only be treated as **displayable text**. Never treat untrusted data **as code** or markup within JavaScript code.
 - Always **JavaScript encode** and **delimit untrusted data** as quoted strings when entering the application



OWASP Tutorials

Learn more here (or take SENG 406)

<https://owasp.org/www-project-web-security-testing-guide/>



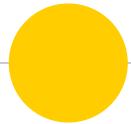
Summary of server-side

Lectures

- HTTP requests & responses
- APIs, endpoints & API-driven design
- The server itself
- e.g. Node.js & express & node packages
- Data persistence e.g. MySQL

Labs and additional tutorials

- JavaScript, TypeScript
- Node.js + Express
- Data persistence
- APIs
- GraphQL
- OWASP



Introduction to

Client-side

March 27 2023

Kia Ora, Aotearoa!

newsable
Renting vs buying:
The pros and cons



generally famous
Why my daughter
made it to Hollywood



newsable
Cats and cucumbers:
A feline fear story



generally famous
Mike King's mental
health parenting tips



health

Widower with terminal cancer fighting to be there for kids

Non-smokers Graham and Mery Brooke-Smith were both diagnosed with stage 4 lung cancer in 2022. She died in November, but he still has hope.

@ 9:29am Torika Tokalau



celebrities
Hayley Holt on
sobriety, her son
Raven and losing a
babv



politics
Watch live: Spy
bosses face
questions at
Parliament



property

Hillside suburb sets record as wealthy buyers pay for top spots

Bucking national trends, home values are still rising in parts of Christchurch - which now has 16 suburbs where a house will cost you \$1 million.

Liz McDonald



wellington

The simplified parking experience that takes



Advertise with Stuff

What kind of contents ‘appear’ in the browser?

1. What the user sees?
2. How the content is described?
(‘view source page’)

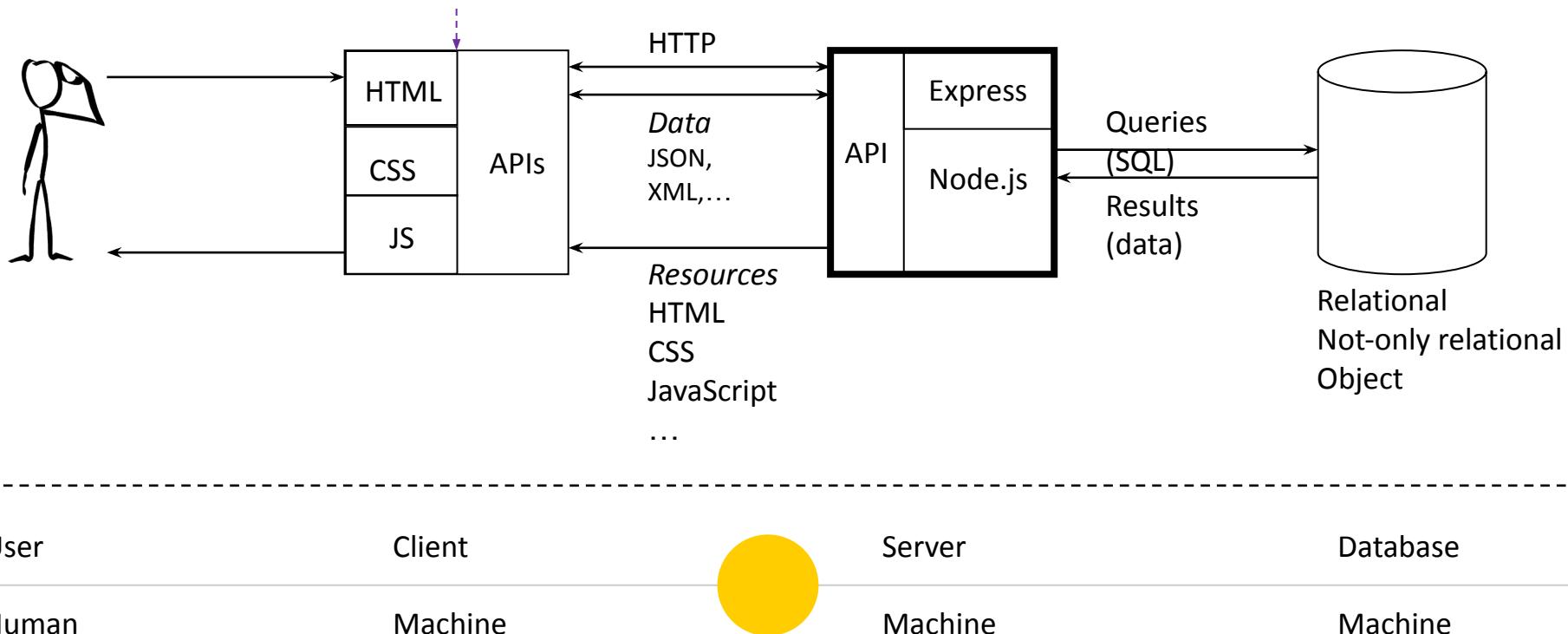
How is that content presented?

- Colours, styles, etc.

What is the user experience in the browser?



What does the client-side app need to do to assemble the data from the API endpoints into coherent, interactive ‘pages’ for the user?





Introduction to the main client-side technologies

JS



Client-side: JavaScript

But note...

“Unlike most programming languages, the [core|original] **JavaScript** language has no concept of input or output. It is designed to run as a scripting language **in a host environment**, and it is up to the **host environment** to provide mechanisms* for communicating with the outside world.”

https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript

* e.g. more APIs

... the full quote:

“Unlike most programming languages, the **JavaScript** language has **no concept of input or output**. It is **designed to run** as a scripting language **in a host environment**, and it is up to the **host environment** to provide mechanisms for **communicating with the outside world**.

The most common host environment is the browser, but JavaScript interpreters can also be found in a huge list of other places, including Adobe Acrobat, Adobe Photoshop, SVG images, Yahoo's Widget engine, server-side environments such as Node.js, NoSQL databases like the open source Apache CouchDB, embedded computers, complete desktop environments like GNOME (one of the most popular GUIs for GNU/Linux operating systems), and others.”

JavaScript + browser \neq JavaScript + Node.

With JS + browser:

- You're dealing with a user!
- Input and output via HTML & CSS (& DOM)
- Deployment of your app is different:
 - Which browser, or version.
 - Network-connection quality.
 - Whether cookies are enabled.
 - The computing power of the hosting machine.

- Different APIs in browser
 - AJAX (XHR)
- Dependencies on libraries
 - 'importing' JS libraries is different
 - e.g. CDN vs npm install
 - What about node packages...?
- Project structure is different
 - JS, HTML, CSS, other assets
- Similar terminology, different meaning
 - e.g. "routes" & "routing"



Client-side: HTML

What is HTML?

“HTML (HyperText Markup Language) is the most basic building block of the Web. It describes and defines the **content** of a **webpage**. Other technologies besides HTML are generally used to describe a webpage's appearance/presentation (**CSS**) or functionality (**JavaScript**).”

(<https://developer.mozilla.org/en-US/docs/Web/HTML>)

- ◉ Be careful about the difference between **content** and **data**.
- ◉ Also, be careful about your concept of a **webpage**. What constitutes a ‘web page’ has changed over time.

Yeah, okay, but what is HTML?

- HTML is a **declarative** ‘language’, comprising:
- A declaration of a document type (HTML), together with a *hierarchical* structure of (nested) HTML elements, where
 - **elements** are identified by **tags**, and
 - elements typically contain some kind of **content** (to display), and where
 - elements may have **attributes**, in which
 - attributes define **characteristics** of elements,
 - attributes often have **values** (for the characteristics),
 - attributes allow **cross-referencing** to CSS and JavaScript, and
 - attributes may be **custom-defined**.

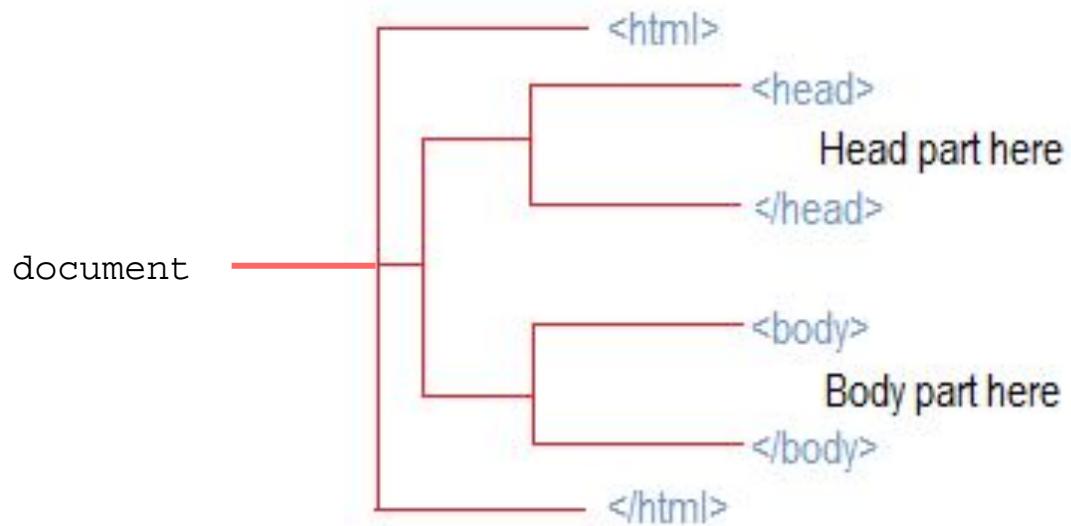
(HTML5 introduced many new elements)

Basic example of HTML page

HTML comprises:

- Document type declaration
- Elements, organised into a hierarchy, e.g.
 - <html>
 - <head>
 - <body>
- Attributes of elements, with values
 - lang="en"

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>A title</title>
  </head>
  <body>
    </body>
</html>
```

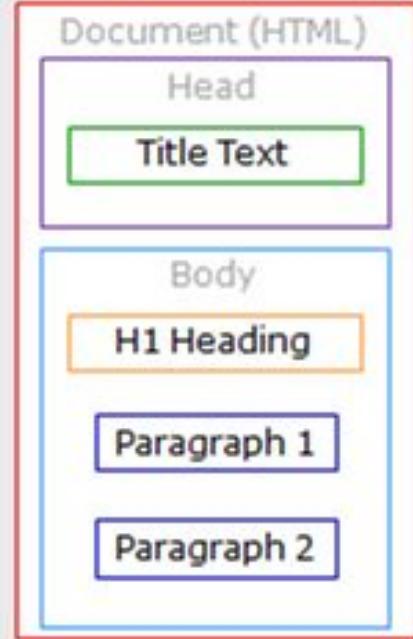


<http://www.corelangs.com/html/introduction/img/html-page-structure.png>

```
<HTML>
  <HEAD>
    <TITLE>Title Text</TITLE>
  </HEAD>

  <BODY>
    <H1>H1 Heading</H1>
    <P>Paragraph 1</P>
    <P>Paragraph 2</P>
  </BODY>

</HTML>
```





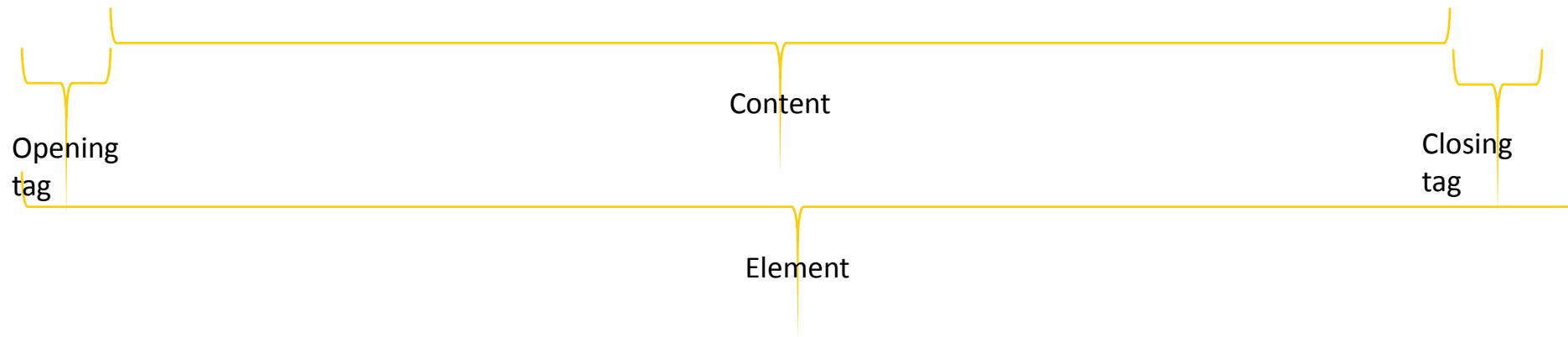
HTML elements

For example, if we wanted to write the following on a web page:

Javascript callbacks are turtles all the way down.



```
<p>Javascript callbacks are turtles all the way down.</p>
```





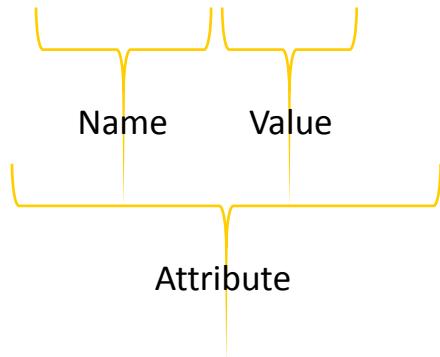
HTML elements & their attributes

For example, if we wanted to write the following on a web page:

Javascript callbacks are turtles all the way down.



```
<p class="comment">Javascript callbacks are turtles...</p>
```





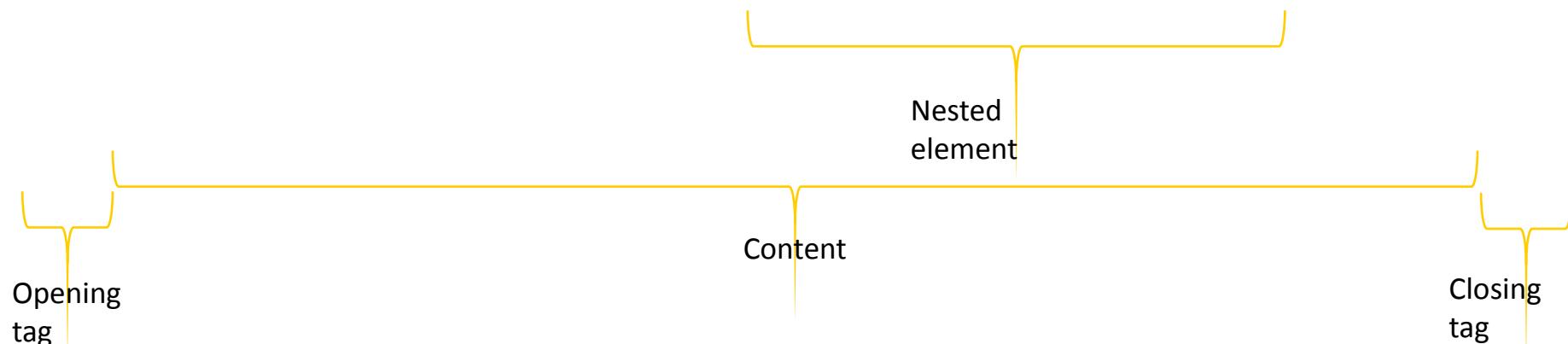
Nested HTML elements

For example, if we wanted to write the following on a web page:

Javascript callbacks are **turtles** all the way down.



```
<p>Javascript callbacks are <strong>turtles</strong> ...</p>
```





One way to change HTML content: JavaScript

HTML: before

```
<p id="four">Oh, cruel world.</p>
```

JavaScript

```
document.getElementById("four").innerHTML = "Hello world!" ;
```

HTML: after

```
<p id="four">Hello world!</p>
```

Custom attributes (we'll come back to this)

- ◉ You can define your own attributes for elements
- ◉ HTML5 offers `data-*` attribute
 - Where * is a string of characters of your choice
 - But potential for name clashes with other JavaScript libraries
 - I define `data-student` in my `student.js` library, and
 - You define `data-student` in your `super-student.js` library
- ◉ Frameworks use custom-defined attributes as part of their two-way binding ‘magic’
 - Angular has `ng-*` attributes
 - Vue has `v-*` attributes

Pointers elsewhere

Validate your HTML at:

<https://validator.w3.org/>

Further information on **HTML5** elements:

<https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5>

CSS



Client-side: CSS

A (declarative) ‘language’ for specifying how contents are presented to a user.

What is CSS?

A set of rules for specifying how content of a web page should look, where:

- ◉ A rule typically comprises:
 - a **selector**, and
 - a **set of properties and values** for how HTML content should be presented
- ◉ There are selectors for:
 - **Attributes** e.g. select on attribute name
 - **Element/s** e.g. select on element type
- ◉ CSS can be contained in:
 - An external style sheet (recommended)
 - An internal style sheet (sometimes okay)
 - An inline style (not recommended)

Examples of rules

Example 1

- Select (all) `<h1>` elements, and
- Set three properties: `color`, `background-color`, and `border`

Example 2

- Select (all) `<p>` elements
- Set one property: `color`

```
h1 {  
    color: blue;  
    background-color: yellow;  
    border: 1px solid black;  
}  
  
p {  
    color: red;  
}
```

CSS can be contained in:

- An external style sheet
(recommended)
- An internal style sheet
(sometimes okay)
- An inline style (not recommended)

```
<head>
<meta charset="utf-8">
<title>Blah</title>
<link rel="stylesheet" href="style.css">
</head>
```

CSS can be contained in:

- An external style sheet (recommended)
- An internal style sheet (sometimes okay)
- An inline style (not recommended)

```
<head>
<title>Blah blah</title>

<style>
  h1 {
    color: blue;
    background-color: yellow;
    border: 1px solid black;
  }

  p {
    color: red;
  }
</style>

</head>
```

CSS can be contained in:

- An external style sheet
(recommended)
- An internal style sheet
(sometimes okay)
- **An inline style
(not recommended)**

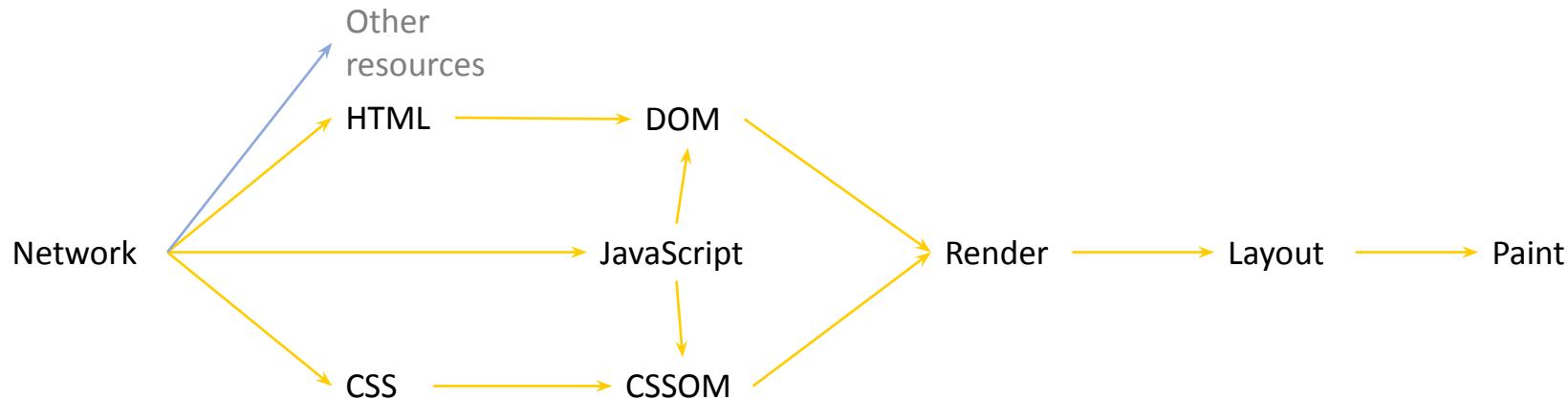
```
<body>  
  <h1 style="color:  
    blue;background-color:  
    yellow;border: 1px solid  
    black;">Hello World!</h1>  
  
  <p style="color:red;">Javascript  
  callbacks are turtles all the way  
  down.</p>  
  
</body>
```



How the client-side
technologies fit together



JavaScript, HTML, CSS, DOM...



Advice

Put CSS at the top in the HTML HEAD

Put JavaScript at the bottom of the page

HTML, CSS and JavaScript

- HTML has elements
 - Many pre-defined
 - Define your own:
 - custom attributes
- Elements can be referenced by
 - Element type e.g. <p>
 - Unique identifier e.g. id=“...”
 - Attribute class e.g. class=“...”
 - (Other ways...?)
- CSS has rules that
 - ‘Apply’ presentation to referenced elements, through selectors
- JavaScript
 - gets (and sets) ...
 - Element content
 - Element attributes & values
 - ... based on references to those elements
- HTML document is the primary source
 - Contains HTML (duh)
 - Contains CSS or reference to CSS
 - Contains JS or reference to JS

Content vs data

- ◉ **Data** as what is ‘in’ JavaScript data structures e.g. arrays, objects.
 - And therefore what is in your database too
- ◉ **Content** as what is ‘in’ the HTML elements.
- ◉ Data needs to be **‘injected’** into HTML content e.g. JavaScript setter
- ◉ User entered information needs to be **retrieved** from the rendered fields on screen e.g. JavaScript ‘getters’

Looking ahead:

- ◉ **Frameworks** help us do this through **two-way binding**.
 - ‘Injecting’ data into content; and retrieving content into data

SENG 365 Week 7

**Single Page Applications and
JavaScript Frameworks**





Outline

- Background on Single Page Applications
- Design patterns
- Document Object Model (DOM)
- Introduction to JavaScript frameworks



Timeline of JavaScript Frameworks

- ◉ **1995** - JavaScript introduced
 - Poor browser compatibility for several years
- ◉ **2004** - standardized AJAX (Google Gmail)
 - Beginning of single-page applications (SPAs)
- ◉ **2006** - jQuery
 - Write JavaScript code without worrying about the browser version
 - AJAX support
- ◉ **2010s** - MVC* Data-view binding frameworks introduced
 - Client-side rendering
- ◉ **2010** - AngularJS, Ember.js, Backbone
- ◉ **2013** - React (*2015 - Redux*)
- ◉ **2014** - Vue.js
- ◉ **2016** - Svelte (compilation to JS)

The move to Single Page Applications

- ◉ Users want **responsiveness & interactivity**; improved user experience
 - Compare user experience with native apps & stand-alone apps
- ◉ **Managing the interactions** with a user is *much more complex* than managing communication with server
 - Think of all those events (e.g. onClicks) to handle

Single Page Applications (SPAs)

“Single page apps are distinguished by their ability to redraw any part of the UI without requiring a server round trip to retrieve HTML. This is achieved by separating the data from the presentation of data by having a model layer that handles data and a view layer that reads from the models.”

<http://singlepageappbook.com/goal.html>

Some features of Single Page Applications

Separate:

- Data
- ‘Content’ & Presentation: HTML & CSS
(and other resources)

Reduce communication with the server/s by:

- **Occasional download** of resources e.g. HTML, CSS and JavaScript
- Asynchronous ‘background’ fetching of data: **AJAX / XHR or web sockets**
- **Fetch data only** e.g. JSON
- Fetch data from different servers:
CORS

‘Page’ navigation

- **Client-side JavaScript** handles the **routing** instead of the browser itself
 - Managing **page history**
 - Routing within the SPA

Reconceive web application

Re-balance workload across:

- ◉ **Server-side** application
 - e.g. Node.js
- ◉ **Client-side** application/
 - Front-end libraries and frameworks e.g. React

Communication between client & server

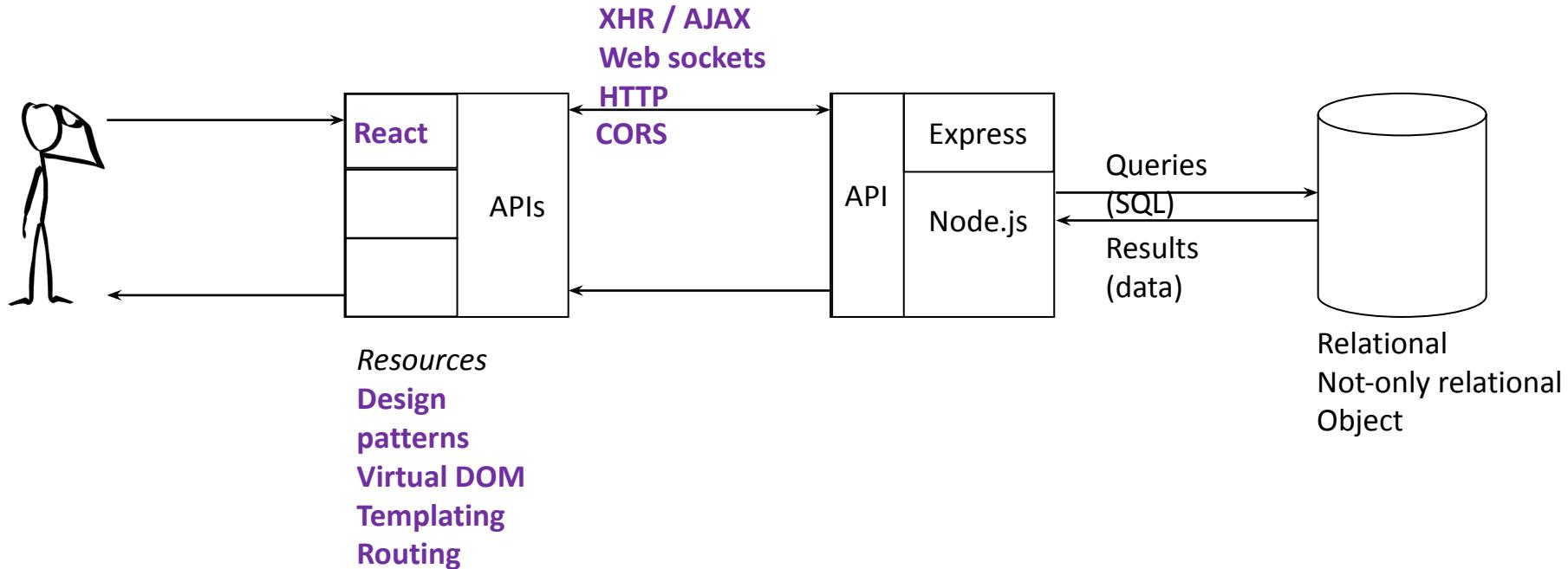
- API-driven/specified
- e.g. AJAX & JSON

Considerations:

- ◉ **Manage application assets / resources**
 - e.g. dependency management
- ◉ **Application design and implementation**
 - Modularisation
 - Design patterns
- ◉ **Templating**



The balance of work between client and server changes. And there are new technologies

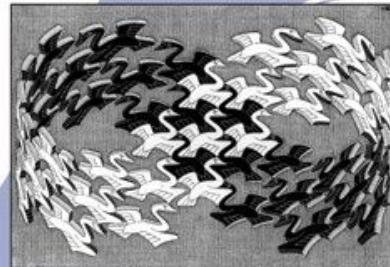


Design patterns

Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

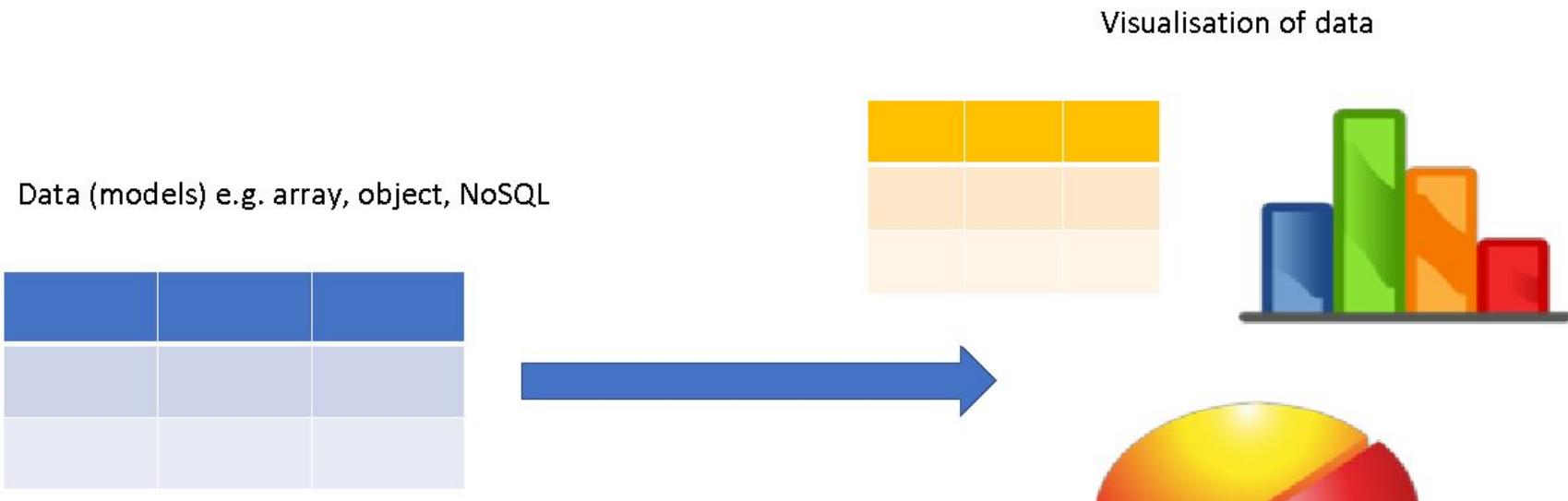
Foreword by Grady Booch



Background: ***Separation of concerns***

- A **design principle** for separating software code into distinct ‘sections’, such that each section addresses a separate concern.
- A **strategy** for handling complexity
 - Of the problem
 - Of the solution e.g. software code
- **Examples** of separation of concerns:
 - Object-oriented programming
 - Classes, objects, methods
 - Web computing
 - HTML: structure/organisation of content
 - CSS: presentation
 - JavaScript: functionality

Example: data and views of data

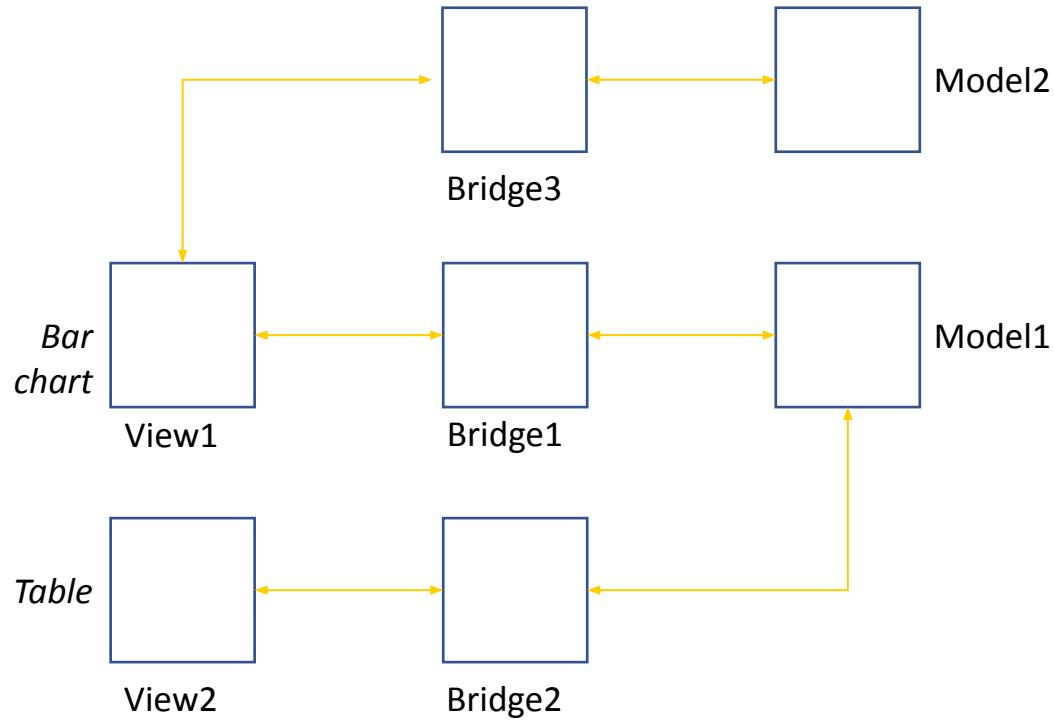


Separation of concerns:

- The management of the data e.g. CRUD
- The visualisation of the data



A generic approach



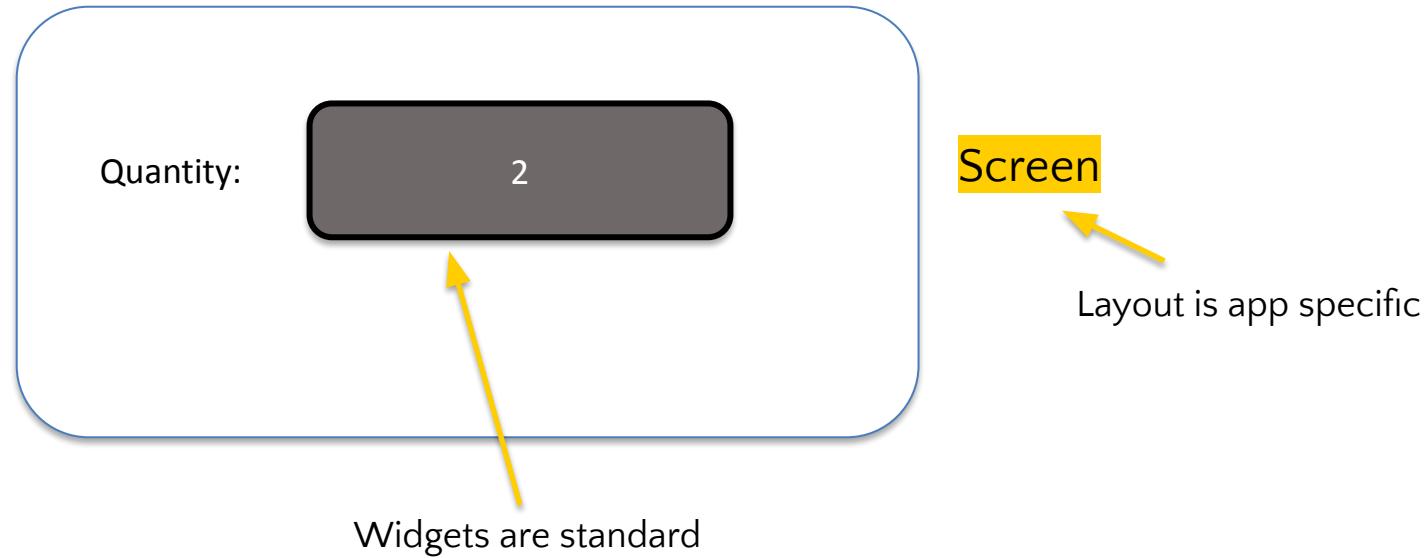
- We want to **display data** to the user **in different ways** e.g. as a bar chart or as a table of data.
- We want to keep our **data independent** of the **way it is presented**.
- We may want to **compute additional values** depending on how and why we present data e.g. add a total in our table.
- We need some **connection** – some ‘bridge’ – **between views and data**.
- We can *reuse*:
 - **models** and
 - **views**.

Variations on a theme of Model-View-{Bridge}* MV...

- Model View Controller (MVC)
- Model View Adaptor (MVA)
- Model View Presenter (MVP)
- Model View ViewModel (MVVM)

Commentary

(Some) disagreements on what
exactly defines these design patterns



Quantity:

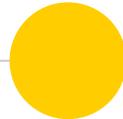
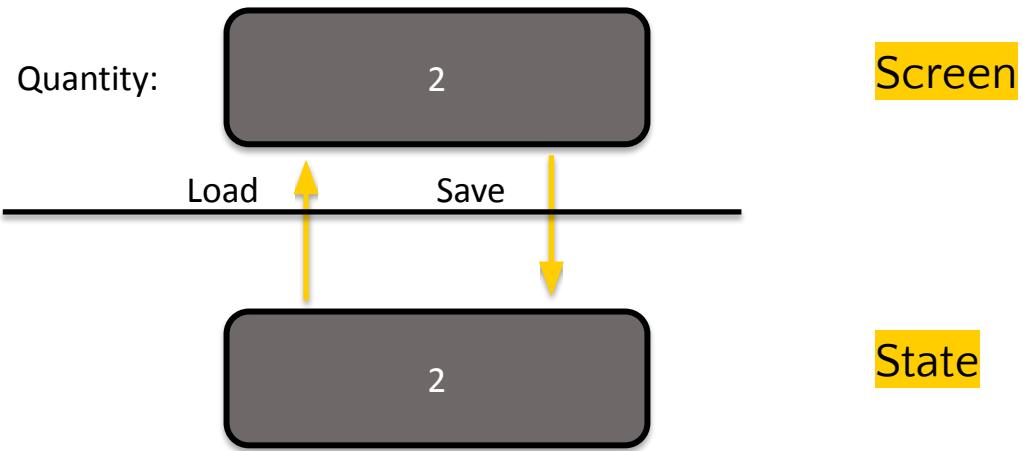
2

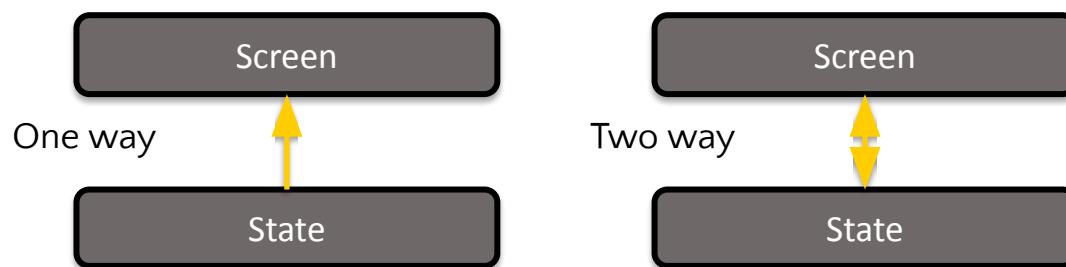
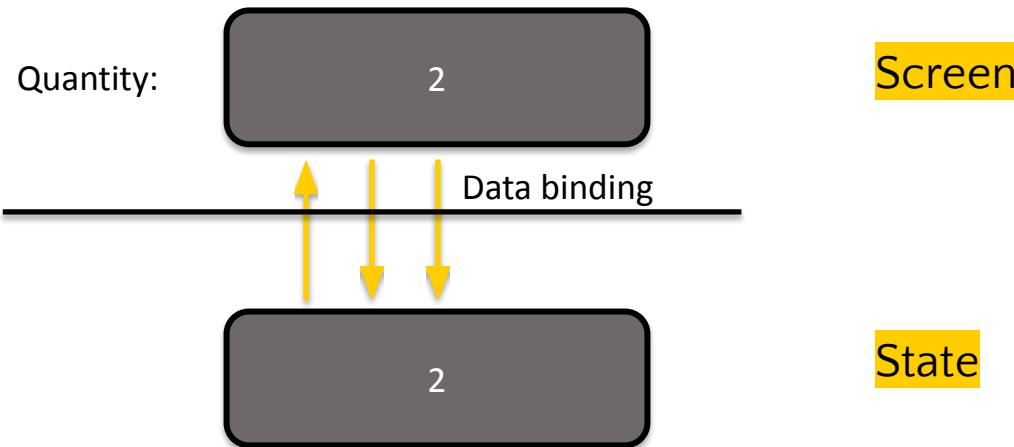
Screen

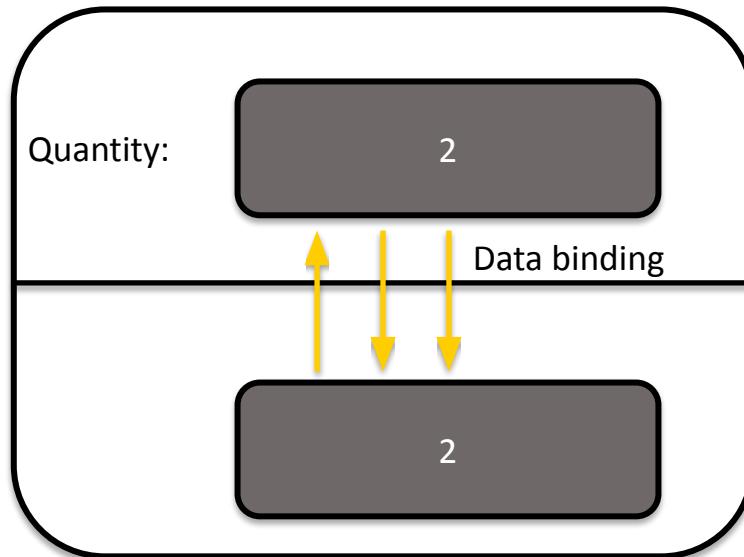
2

State





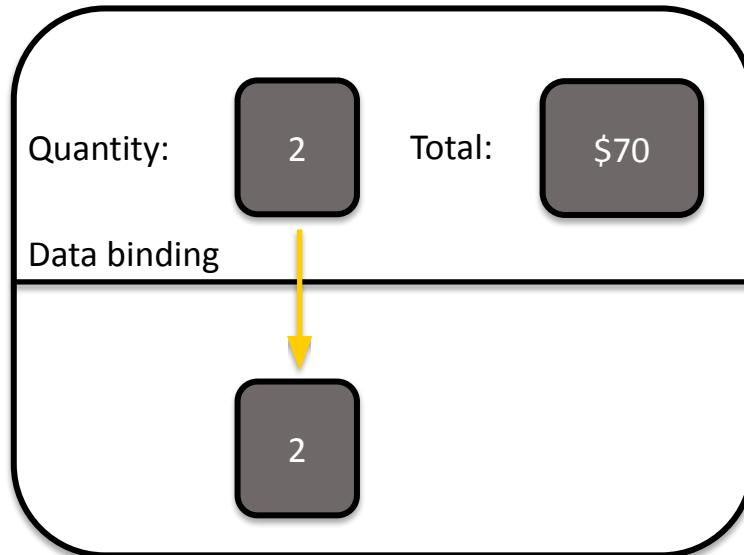




Screen

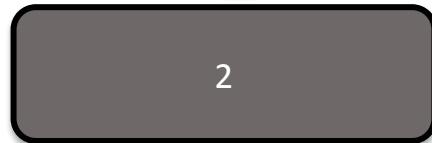
Session
state

Record
state

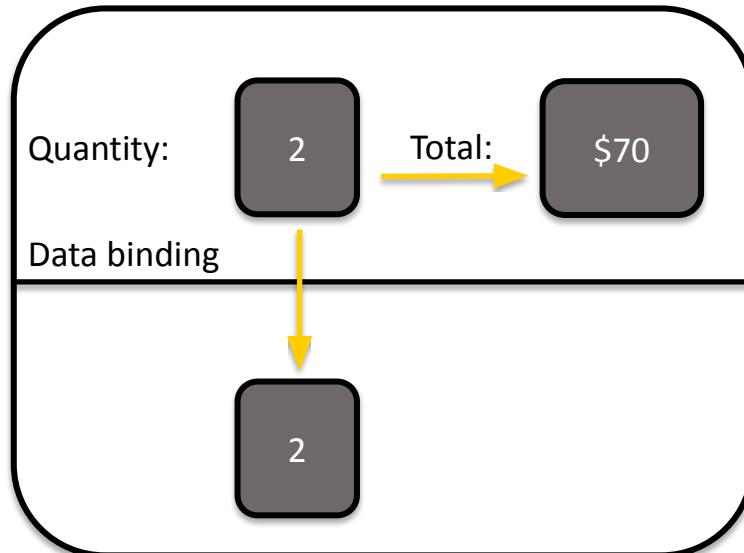


Screen

Session
state

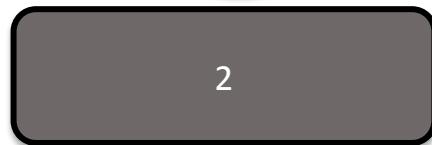


Record
state

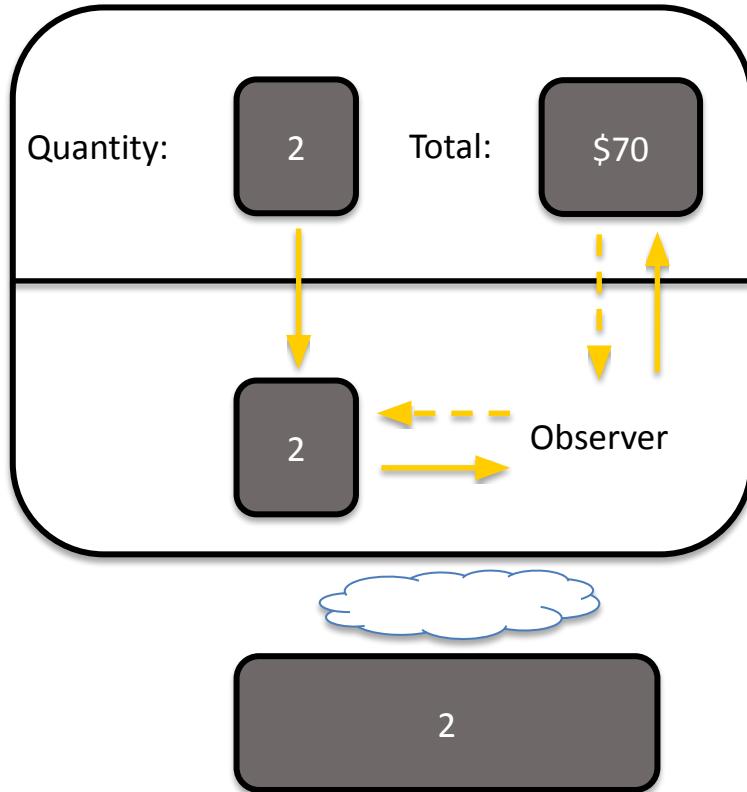


Screen

Session
state



Record
state



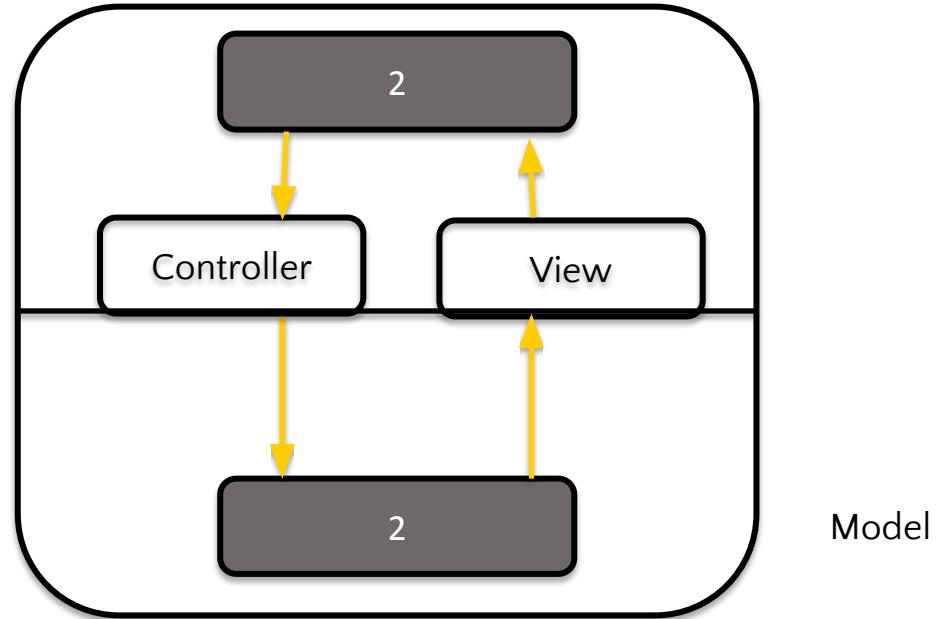
Screen = Presentation

*Specific to the UI
Independent of Domain*

Session state = Model

*Independent of UI
Specific to the Domain*

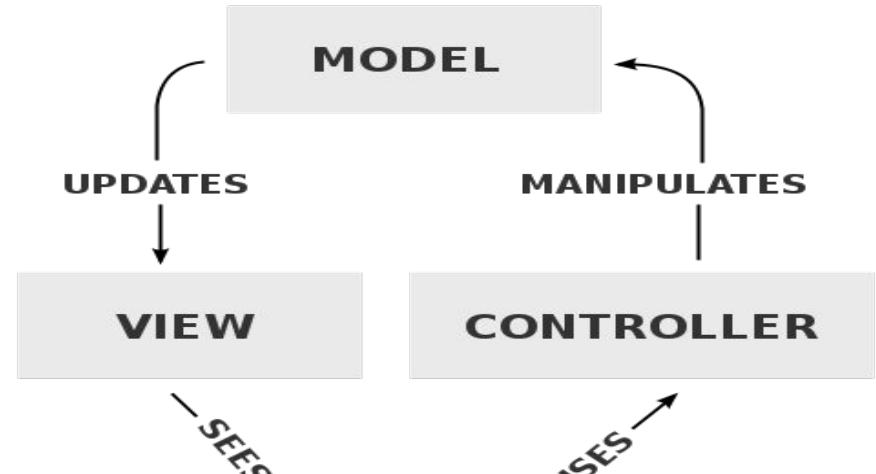
Record state = Data

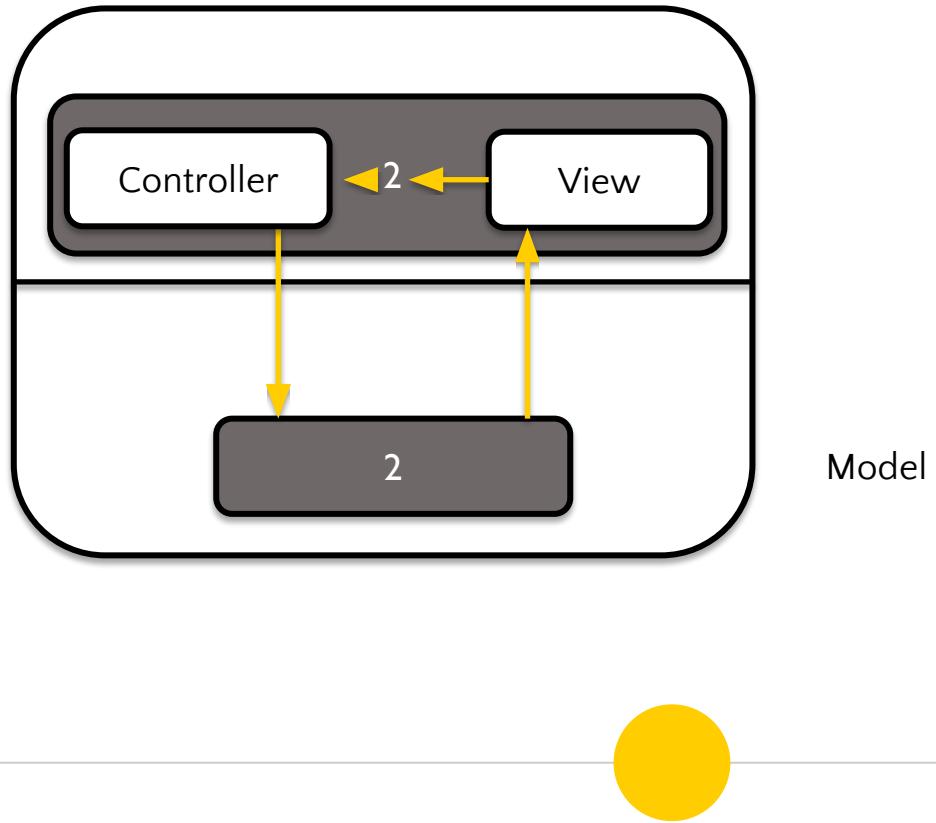


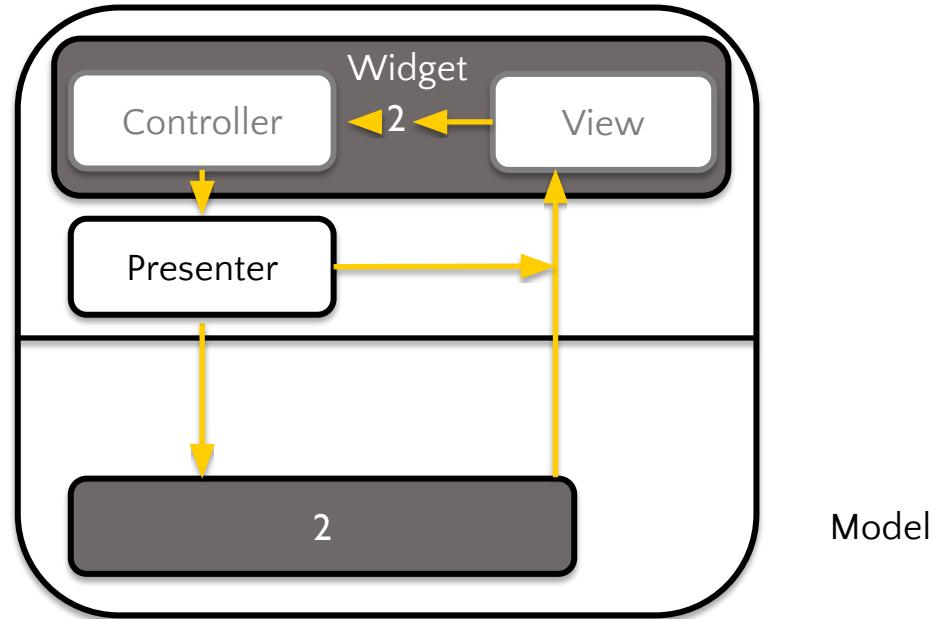
MVC:

- + Multiple views
- + Synchronized views
- + Pluggable views and controllers
- + Exchangeable look and feel

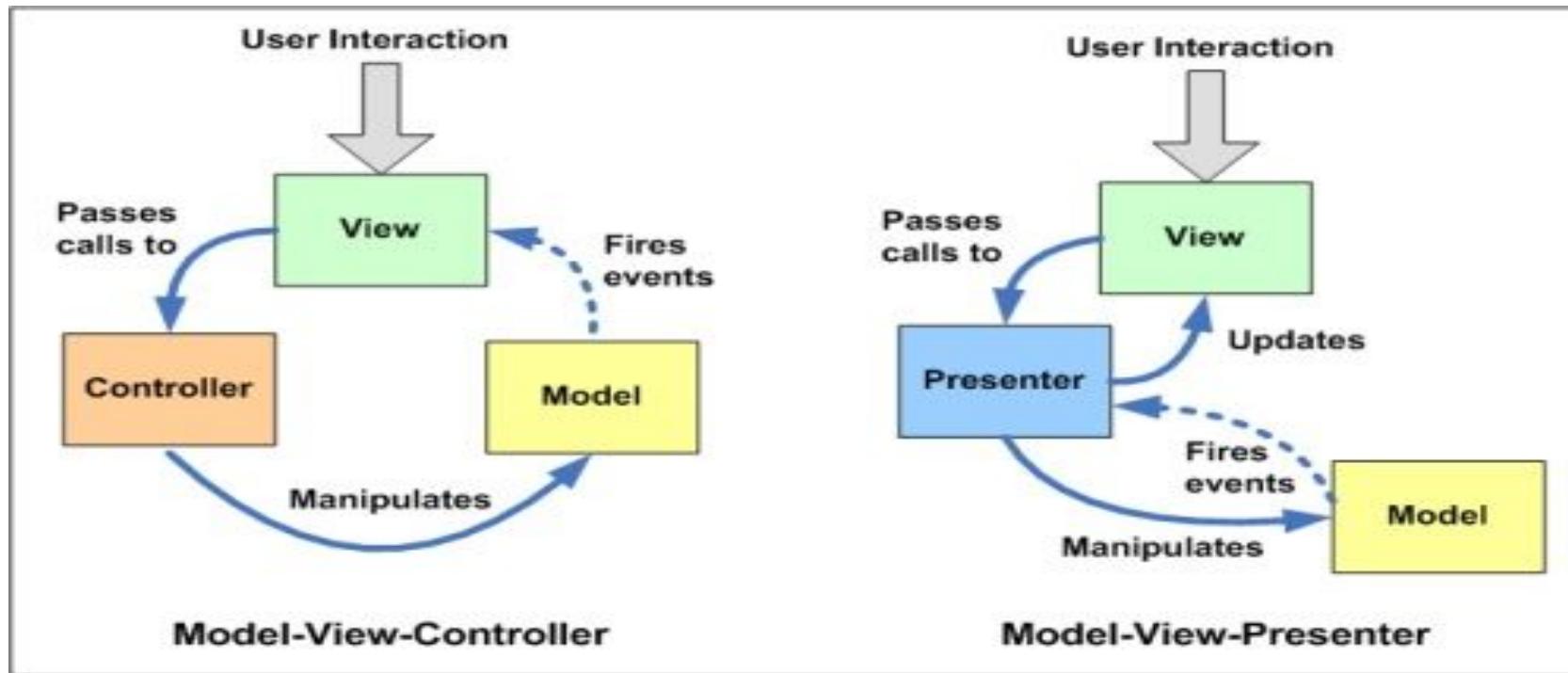
- Complexity
- Can have lots of updates/notifications
- Links between controller and view
- Coupling of controller/view and model
- Mix of platform-dependent/independent code within controller and view (porting)







MVC & MVP

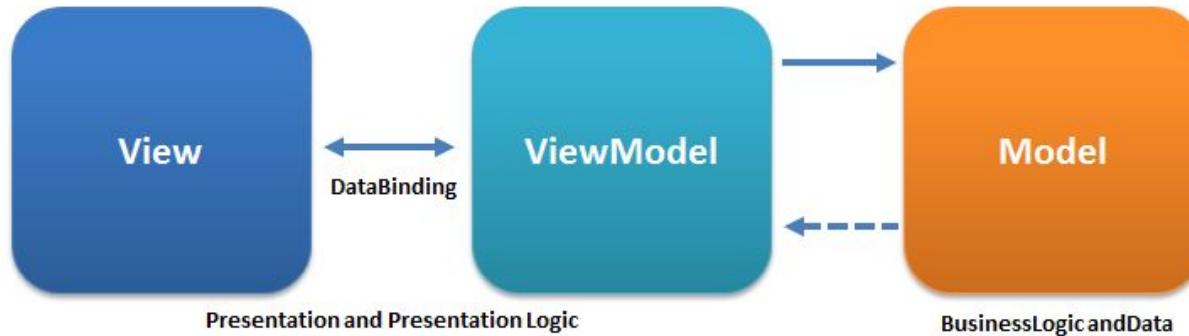


MVC versus MVP

- **MVC** is Model-View-Controller
- **MVP** is Model-View-Presenter
- Much hot air expended in defining / comparing / contrasting these.
 - **MVC**: view is stateless with not much logic. Renders a representation of model(s) when called by controller or triggered by model. Gets data directly from model.
 - **MVP**: view can be completely isolated from model and rendering data from presenter, or can be a MVC view, or somewhere in between
- But many variants.
- Categorisation not very important; you mostly just use whatever tools the framework gives you.
- The term MV* may be safer to avoid arguments!

MVVM

Model/View/ViewModel



- ViewModel is just the data currently required by the view
 - In a web context, Model may be on server, View and ViewModel on client
- *Data binding* synchronises view and viewModel bidirectionally
 - Uses lower-level "hidden" mechanism



Design pattern summary

- **Developing web applications is challenging**
- **One strategy** to development, and to integration, is to '**divide and conquer**'
- **Separate out concerns**
 - 3-tier architecture of browser, server and data store
- Separate out concerns
 - MV* as a **design concept**
 - **Differences of opinion** on what the M, the V and the * were
 - **Differences on how M, V and * interact** with each other
 - Looked at MVC, MVP, MVVM and MVW

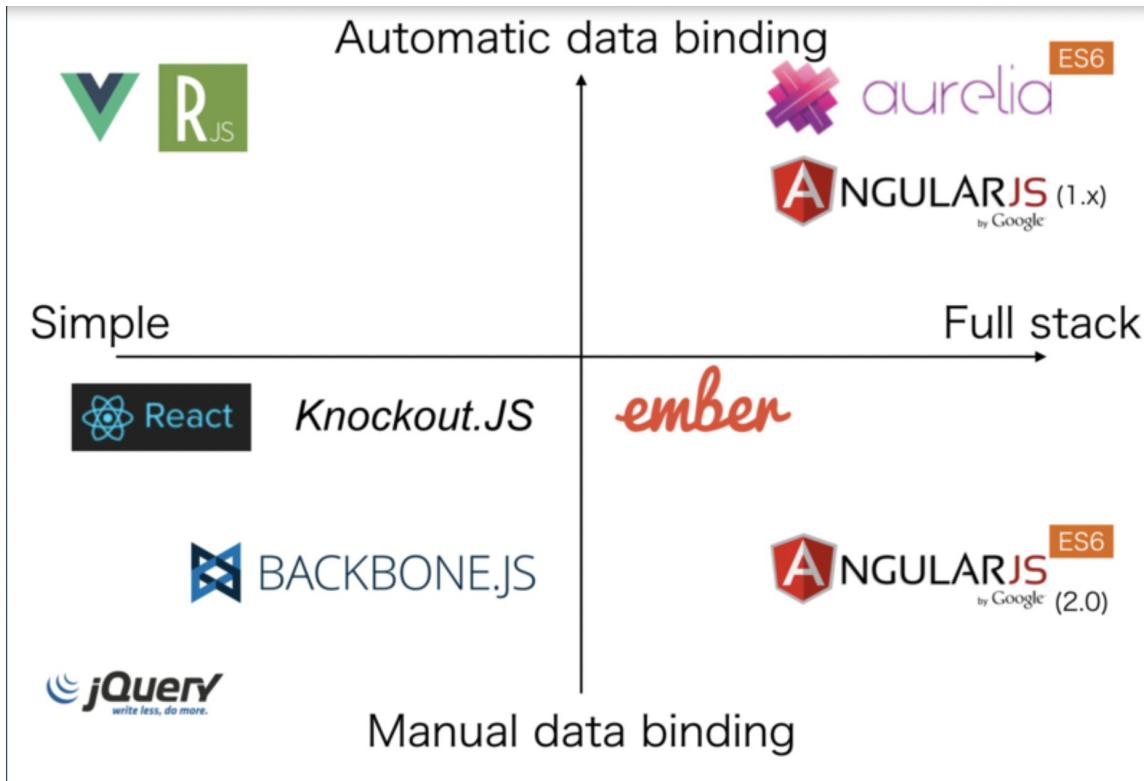


Data-binding frameworks

Angular, React, Vue.JS etc.



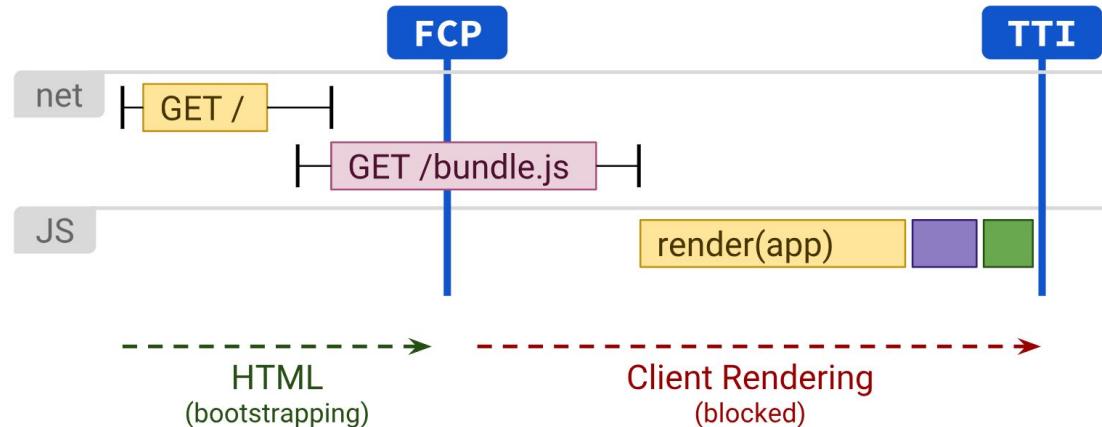
Data binding frameworks





Client-side rendering

- Renders the page using JS in the browser
- Logic, data-fetching, templating, and routing handled by browser code
- First-contentful page (FCP) as JS bundle loaded
- Time-to-interactive (TTI) after the render function is executed





Templating

```
1 <div id="counter">  
2   Counter: {{ counter }}  
3 </div>
```

html

```
1 const Counter = {  
2   data() {  
3     return {  
4       counter: 0  
5     }  
6   }  
7 }  
8  
9 Vue.createApp(Counter).mount('#counter')
```

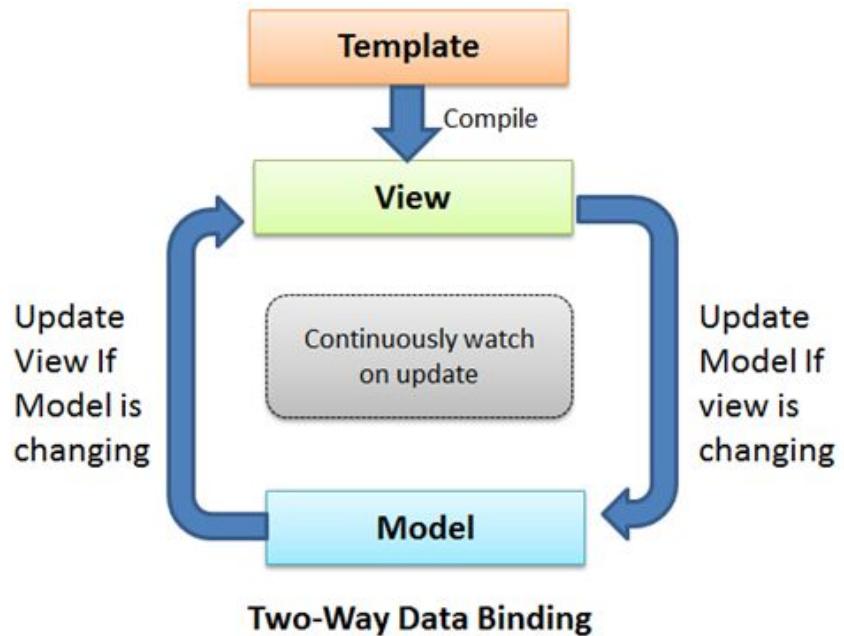
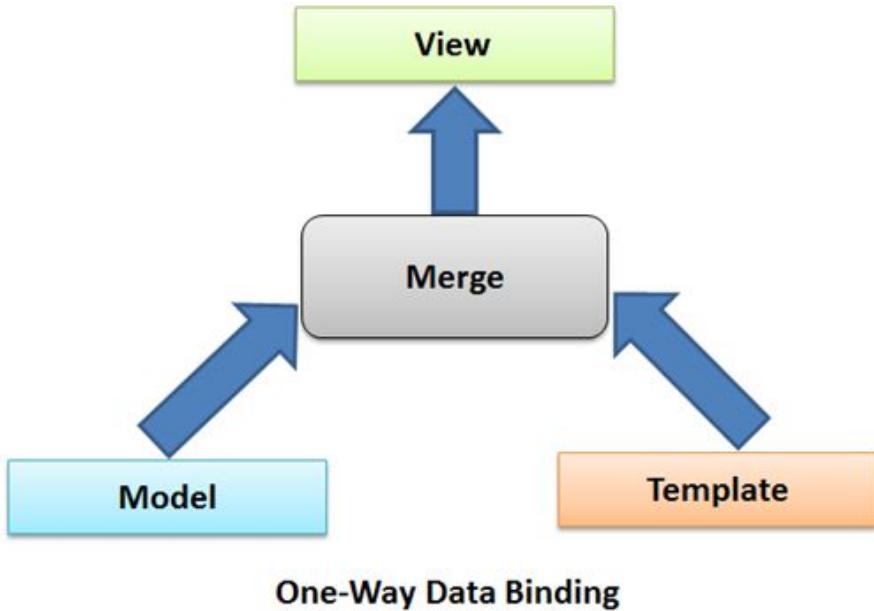
js

Counter: 0

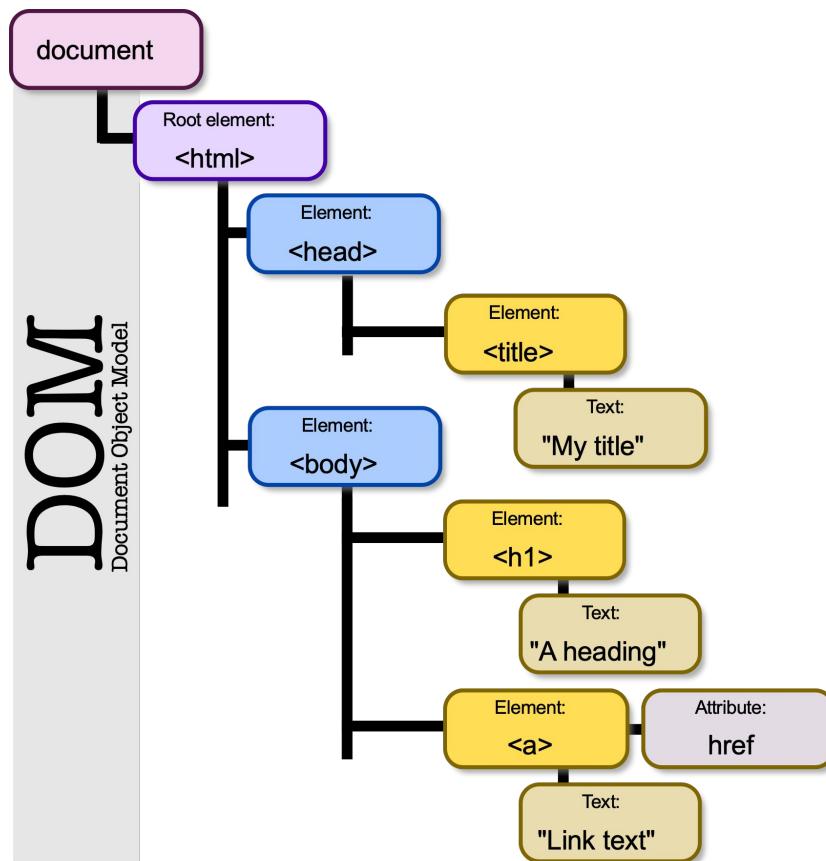
- Frameworks such as Angular and Vue use **templating** to map JS variables to HTML elements
- Data-binding in HTML using “moustache” syntax:
 - Most frameworks use curly braces: {{data}}
- JS, CSS, HTML separate files / sections



Types of data binding



One way binding can also go the other way: view -> data



"The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document." (W3C). The **HTML DOM** : standard model for HTML documents

Examples of DOM ‘sizes’

Website	DOM count
www.bbc.com/news	2298
Facebook.com	1570
Facebook.com with a few scrolls	8300
Yahoo performance site	700
Trademe.co.nz	3200

1. Load page
2. Open JavaScript console
3. `document.getElementsByTagName('*').length`

DOM performance

- The **DOM (Document Object Model)** can become **excessively large** in an application, e.g. Facebook, when you've scrolled down a bit
- A large number of **DOM nodes to traverse**
- **Performance impact** e.g. you have to modify a large number of nodes (even with a tree structure)

Virtual DOM

- The **Virtual DOM**: an abstraction of the DOM
 - Modify the virtual DOM; update the real DOM from the virtual DOM when needed
 - Libraries / frameworks (e.g. React, Vue.js) will use a virtual DOM in the background
 - You don't need to work directly with the DOM

<https://bitsofco.de/understanding-the-virtual-dom/>

Updating the Virtual DOM

When

The data has changed and it needs to be updated: but how do we know that the data has changed?

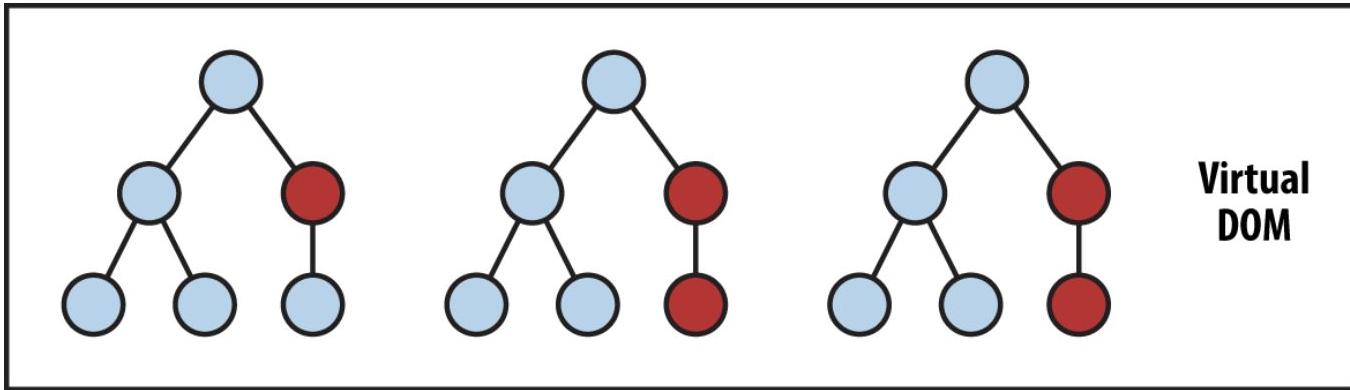
1. **Dirty checking**: poll the data at a regular interval to check the data structure recursively.
2. **Observable**: observe for state change. If nothing has changed, don't do anything. If something has changed, we know exactly what to update.

Updating the Virtual DOM

How

How do we make changes efficiently?

- Need efficient **diff** algorithms.
- **Batch** DOM read/write operations.
- **Efficient update** of sub-tree only.



State Change

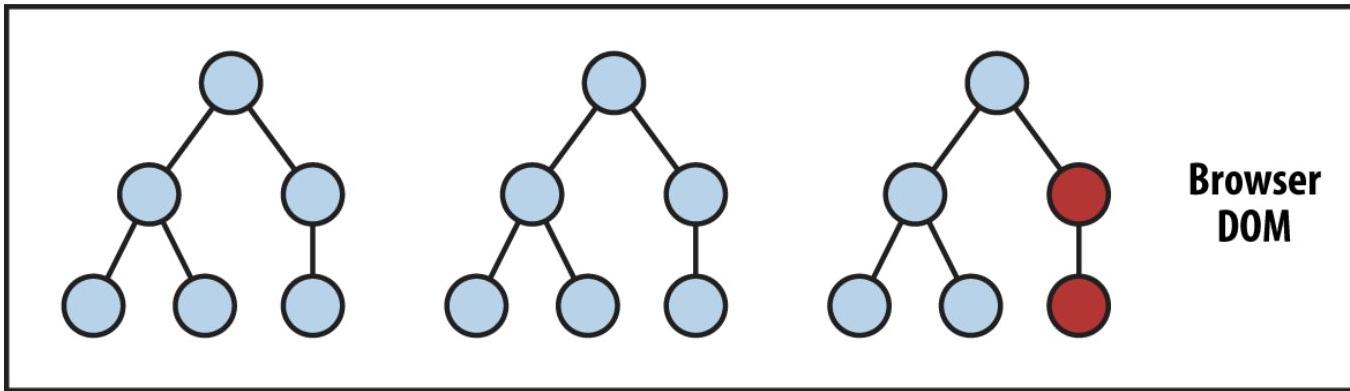


Compute Diff



Re-render

**Virtual
DOM**

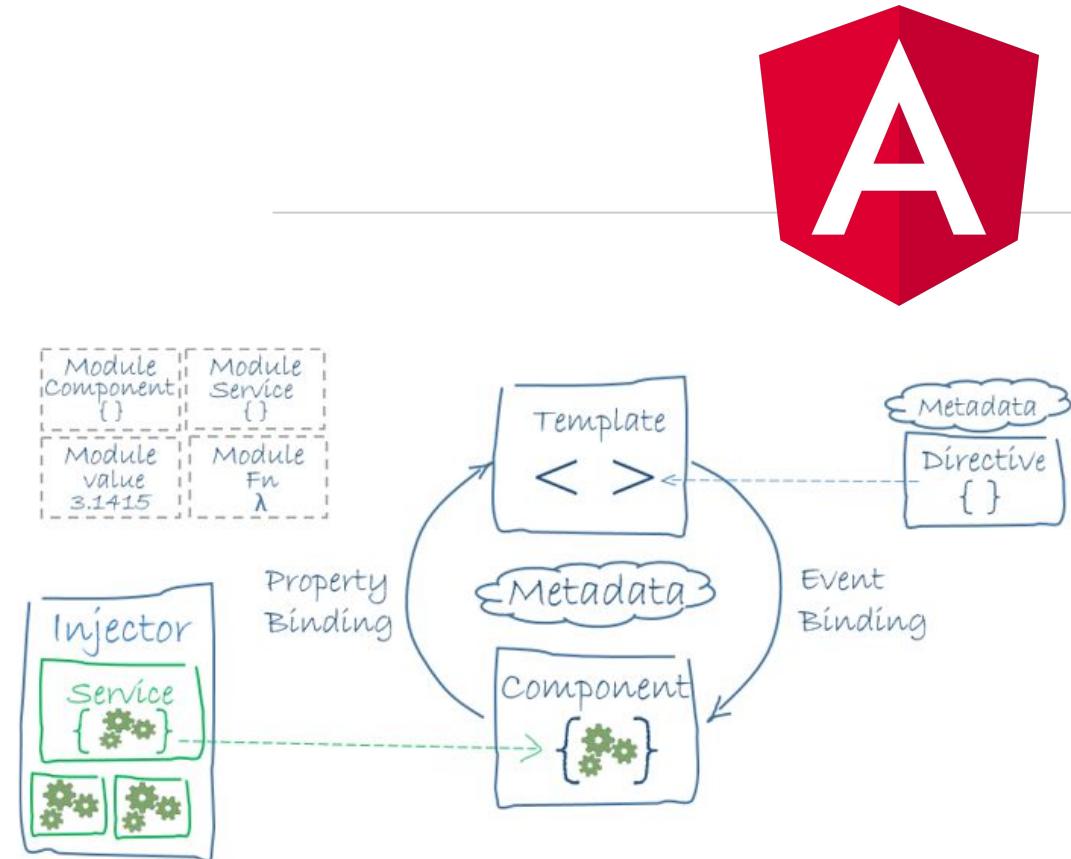


**Browser
DOM**



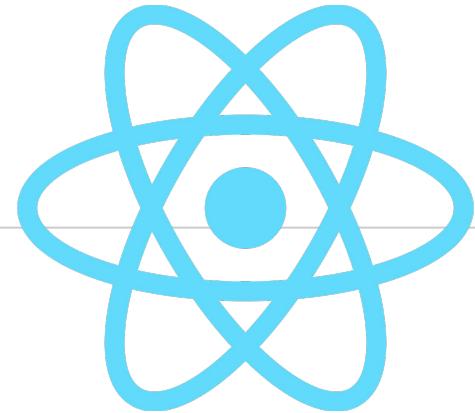
Angular

- Platform for enterprise-scale web application development
- Typescript only (no JS)
- Relatively complex to learn
- Two-way binding
- Dirty-checking to know when DOM should change





React



- Unlike Angular **does not use templating**
- Rather defines reusable **components** in **JSX**
 - Brings the HTML inside the JavaScript
 - Each component has a lifecycle
- Most popular framework at present
 - Thousands of third-party libraries
- Uses a **virtual DOM**
- More about React in coming weeks



VueJS



- Designed by Google dev who found Angular too heavy-weight
- Uses component with lifecycles concept & virtual DOM from React
- But templating in HTML, instead of JSX
 - Closer to native HTML than React, whereas JS is the “starting point” in React



Svelte

- Svelte is different again, it parses .svelte files and compiles into JavaScript
 - Uses abstract syntax tree to generate JS and CSS
- Compiled JS mounts the component, handles events, and patches the DOM directly (no virtual DOM)
 - All HTML elements are created by JS
- No framework code, so small and fast code



<https://lihautan.com/the-svelte-compiler-handbook/>

SENG 365 Week 8

React: JSX and Components



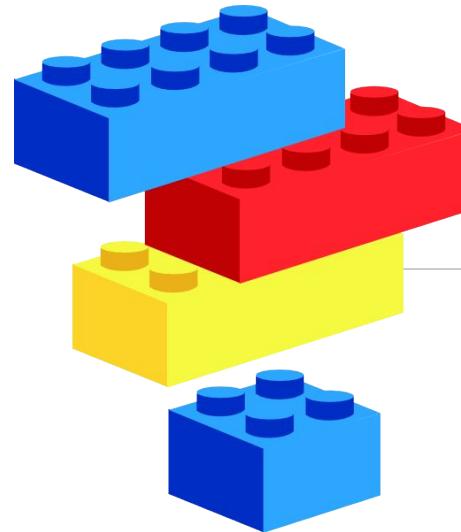


This week

- JSX
- Class and Function components



JSX and Components





Creating a React project

- Several toolchains available
 - Create React App easiest way to start with a project from scratch
 - See also: <https://reactjs.org/docs/create-a-new-react-app.html>
 - You can also modify an existing project (w/o toolchain) by adding React JS using script tags and an appropriate JSX preprocessor

```
npx create-react-app my-app  
cd my-app  
npm start
```

What's up?...

Notifications Messages



User Status

⌚ 15 minutes ago

♥ 26 Likes

💬 6 Comments

↳ 3 Shares



♥ 16 💬 3

Title

Bacon ipsum dolor amet pork chop pig commodo, biltong ham hock adipisicing venison rump spare ribs ut. Incididunt alcatra lorem sint tail strip steak

Username commented on your photo:

Bacon ipsum dolor amet pork chop pig commodo, biltong ham hock.

⌚ just now



Username added a new gallery.



⌚ 2h



Username is now following you.

⌚ 4h



Username is now following you.

⌚ 5h



Username

Username

Username

Username

Username

Username

Username



Many ways to group files

```
common/  
  Avatar.js  
  Avatar.css  
  APIUtils.js  
  APIUtils.test.js  
  
feed/  
  index.js  
  Feed.js  
  Feed.css  
  FeedStory.js  
  FeedStory.test.js  
  FeedAPI.js  
  
profile/  
  index.js  
  Profile.js  
  ProfileHeader.js  
  ProfileHeader.css  
  ProfileAPI.js
```

```
api/  
  APIUtils.js  
  APIUtils.test.js  
  ProfileAPI.js  
  UserAPI.js  
  
components/  
  Avatar.js  
  Avatar.css  
  Feed.js  
  Feed.css  
  FeedStory.js  
  FeedStory.test.js  
  Profile.js  
  ProfileHeader.js  
  ProfileHeader.css
```

No one right way

Think about **import** statements

Consistency



Anatomy of a **React** app

```
import ReactDOM from "react-dom"  
import App from "./App"  
  
ReactDOM.render(<App />, document.getElementById("root"))
```



Anatomy of a **React** app

```
import React from 'react'  
import ReactDOM from 'react-dom/client'  
  
import App from './App'  
  
ReactDOM.createRoot(document.getElementById('root')).render(<App />)
```

This has changed slightly in React 18



Anatomy of a **React** app

```
import ReactDOM from "react-dom"  
import App from "./App"  
  
ReactDOM.render(<App />, document.getElementById("root"))
```

ReactDOM is a JavaScript library that renders **JSX** to elements in the document object model (DOM)



Anatomy of a **React** app

```
import ReactDOM from "react-dom"  
import App from "./App"  
  
ReactDOM.render(<App />, document.getElementById("root"))
```

Application components (e.g. App) are written in JSX

Imported components can have `.js` or `.jsx` extension and it does not need to be indicated in the `import` statement



Anatomy of a **React** app

```
import ReactDOM from "react-dom"
import App from "./App"

ReactDOM.render(<App />, document.getElementById("root"))
```

The second parameter is the HTML element that the compiled JSX should be attached to (in this case the element with id `root`)



JSX basics

- JSX is a syntax extension of JavaScript
- Looks a bit like a mix of JS and HTML
- Compiled to HTML before running in the browser
- Your project needs a JSX preprocessor (this is already installed by Create React App)

```
npm install babel-cli@6 babel-preset-react-app@3
```



JSX examples

Single element

```
const title = <h1>Welcome all!</h1>
```



JSX examples

Single element with attributes (like HTML)

```
const example = <h1 id="example">JSX Attributes</h1>;
```



JSX examples

Multiline and nested expressions

- Requires surrounding brackets: ()
- Must be **only one** outermost tag (e.g.)

```
const myList = (
  <ul>
    <li>item 1</li>
    <li>item 2</li>
    <li>item 3</li>
  </ul>
);
```



JSX examples

Can contain evaluated JavaScript

- Denoted by curly brackets: { }

```
let expr = <h1>{10 * 10}</h1>;
// above will be rendered as <h1>100</h1>
```



JSX conditionals

Can be tricky and there is more than one approach

1. JavaScript Boolean short circuit evaluation

```
// All of the list items will display if
// baby is false and age is above 25
const tasty =
  <ul>
    <li>Applesauce</li>
    { !baby && <li>Pizza</li> }
    { age > 15 && <li>Brussels Sprouts</li> }
    { age > 20 && <li>Oysters</li> }
    { age > 25 && <li>Grappa</li> }
  </ul>
);
```



JSX conditionals

Can be tricky and there is more than one approach

2. Ternary operator <expr> ? <expr> : <expr>

```
// Using ternary operator
const headline = (
  <h1>
    { age >= drinkingAge ? 'Buy Drink' : 'Do Teen Stuff' }
  </h1>
);
```



Arrays and JSX collections

Use `map` function to generate a collection from an array

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li>{number}</li>
);

<ul>{listItems}</ul>
```

But this generates a warning message that
the `list items` should have a `key`



List item keys in JSX

- Keys are necessary because they tell React when a render needs to happen because a list element has changed or is added/removed.

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li key={number.toString()}>
    {number}
  </li>
);
```



Key as id field in list object

```
function Car(props) {
  return <li>I am a { props.brand }</li>;
}

function Garage() {
  const cars = [
    {id: 1, brand: 'Ford'},
    {id: 2, brand: 'BMW'},
    {id: 3, brand: 'Audi'}
  ];
  return (
    <>
      <h1>Who lives in my garage?</h1>
      <ul>
        {cars.map((car) => <Car key={car.id} brand={car.brand} />)}
      </ul>
    </>
  );
}
```



JSX compilation

JSX is syntactic sugar for JavaScript that calls `React.createElement`

```
const App = () => {
  const now = new Date()
  const a = 10
  const b = 20

  return (
    <div>
      <p>Hello world, it is {now.toString()}</p>
      <p>
        {a} plus {b} is {a + b}
      </p>
    </div>
  )
}
```



```
const App = () => {
  const now = new Date()
  const a = 10
  const b = 20
  return React.createElement(
    'div',
    null,
    React.createElement(
      'p', null, 'Hello world, it is ', now.toString()
    ),
    React.createElement(
      'p', null, a, ' plus ', b, ' is ', a + b
    )
  )
}
```



React.createElement

Takes three parameters:
type, props, children

It returns a JavaScript
object

The screenshot shows the Chrome DevTools Elements tab. At the top, there are icons for back, forward, and refresh, followed by tabs for Elements, Console, Sources, and Network. Below the tabs, there are buttons for play/pause, stop, and filter, along with a dropdown menu set to 'top'. The main area displays the structure of a React element:

```
> React.createElement('div', {}, 'Hello')
└─ { $$typeof: Symbol(react.element), type: 
    $$typeof: Symbol(react.element)
    key: null
    props: { children: "Hello" }
    ref: null
    type: "div"
    _owner: null
    └─ __proto__: Object }
```



React.createElement

React elements can be nested in children

ReactDOM.render is passed one of these nested objects

```
React.createElement('div', { }, React.createElement('p', {}, 'A p inside a div'))  
▼ $$typeof: Symbol(react.element), type: "div", key: null, ref: null, props: {...}, ...  
  $$typeof: Symbol(react.element)  
  key: null  
  ▼ props:  
    ▼ children:  
      $$typeof: Symbol(react.element)  
      key: null  
      ▼ props:  
        children: "A p inside a div"  
        ► __proto__: Object  
        ref: null  
        type: "p"  
        _owner: null  
        ► __proto__: Object  
      ► __proto__: Object  
    ref: null  
    type: "div"  
    _owner: null  
    ► __proto__: Object
```



Defining your own Components

Class component

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```



Defining your own Components

Class component

```
class Welcome extends React.Component {  
  render() {  
    return <h1>Hello, {this.props.name}</h1>;  
  }  
}
```

Function component

```
function Welcome(props) {  
  return <h1>Hello, {props.name}</h1>;  
}
```

Equivalent



Composing Components

`props` stands for properties and is a
read-only object that is passed to
the component

Components in JSX have to start
with a capital letter to differentiate
them from HTML tags

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}
```



What does this render?

```
const Hello = (props) => {
  return (
    <div>
      <p>
        Hello {props.name}, you are {props.age} years old
      </p>
    </div>
  )
}

const App = () => {
  const name = 'Peter'
  const age = 10

  return (
    <div>
      <h1>Greetings</h1>
      <Hello name="Maya" age={26 + 10} />
      <Hello name={name} age={age} />
    </div>
  )
}
```



What does this render?

```
const Hello = (props) => {
  return (
    <div>
      <p>
        Hello {props.name}, you are {props.age} years old
      </p>
    </div>
  )
}

const App = () => {
  const name = 'Peter'
  const age = 10

  return (
    <div>
      <h1>Greetings</h1>
      <Hello name="Maya" age={26 + 10} />
      <Hello name={name} age={age} />
    </div>
  )
}
```

Greetings

Hello Maya, you are 36 years old

Hello Peter, you are 10 years old



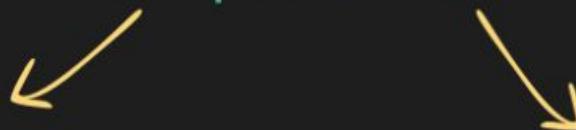
Component lifecycle and managing state



Making components stateful



Props vs State



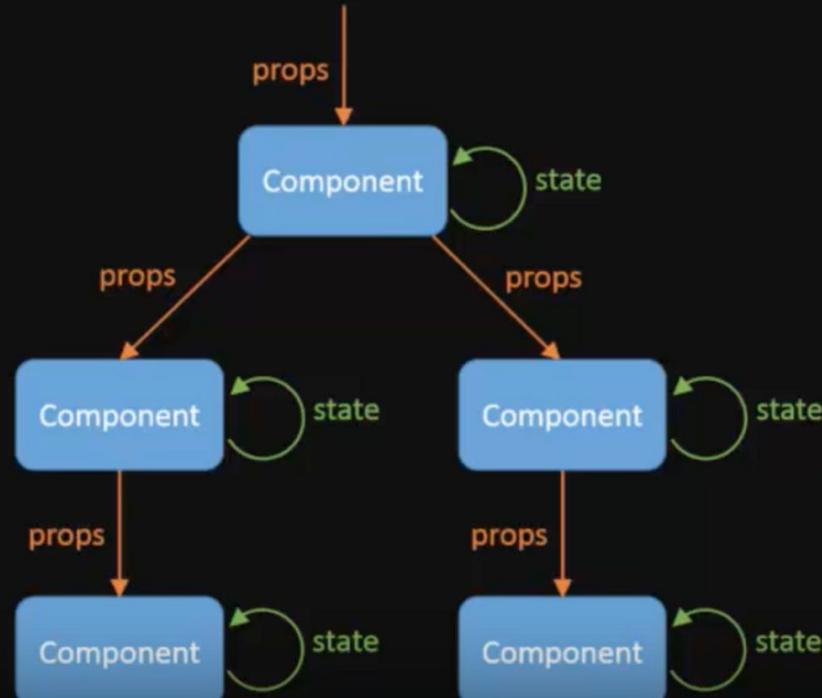
- ✓ props are read-only
- ✓ props can not be modified
- ✓ state changes can be asynchronous
- ✓ state can be modified using `this.setState`



Data flow

Data Flow - React

Components can either be passed data (PROPS), or materialize their own state and manage it over time (STATES)





State and Lifecycle in Class Components

Class components extend `React.Component`.

`this.state` is the component's state object

It is initialized in the component's constructor, which takes `props` as a parameter. It should always call `super(props)`; at the beginning.

The state object is how we change the view (made by the `render` method) based on events.

```
class Clock extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {date: new Date()};  
  }  
}
```



State and Lifecycle in Class Components

Every React has a lifecycle, accessed through component methods, e.g.:

- `componentDidMount`
- `componentDidUpdate`
- `componentWillMount`

`this.state` should not be set directly, instead **use `this.setState()`**

`this.setState()` is a *request* to change the state, not updated immediately

The object sent to `setState` is *merged* with the existing state in a batch operation

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }
}
```



State and Lifecycle in Class Components

The state can be referenced with `this.state` in the `render()` method

You should **not** update the state in `render()`

Why?

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}
```



State and Lifecycle in Class Components

The state can be referenced with `this.state` in the `render()` method

You should **not** update the state in `render()`

Why?

The state update will trigger a new render of the view.

In this example the state is updated every second using `setInterval`. How about events triggered by user actions?

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

SENG 365 Week 9

React: Event Handling, Hooks, Global State





This week

- Event handling
- Hooks
- Global storage
- Routing
- Debugging React apps



Event handling and hooks



Event handling

- Event handling in a component is similar to DOM events but with some differences
 - Camel case notation: `onClick`, `onSubmit`, etc.
 - Pass a function as the event handler
- The function can be a method in the component class but cannot be called unless `this` is explicitly bound.



Digression: function binding

```
const module = {
  x: 42,
  getX: function() {
    return this.x;
  }
};

const unboundGetX = module.getX;
console.log(unboundGetX()); // The function gets invoked at the global scope
// expected output: undefined

const boundGetX = unboundGetX.bind(module);
console.log(boundGetX());
// expected output: 42
```



Accessing a component method for an event (option 1)

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // This binding is necessary to make `this` work in the callback
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(prevState => ({
      isToggleOn: !prevState.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}
```

Bind `this` in the component constructor.



Accessing a component method for an event (option 2)

```
class LoggingButton extends React.Component {
  // This syntax ensures `this` is bound within handleClick.
  // Warning: this is *experimental* syntax.
  handleClick = () => {
    console.log('this is:', this);
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
```

Use public class fields syntax

This syntax is experimental but enabled by default by Create React App



Accessing a component method for an event (option 3)

```
class LoggingButton extends React.Component {
  handleClick() {
    console.log('this is:', this);
  }

  render() {
    // This syntax ensures `this` is bound within handleClick
    return (
      <button onClick={() => this.handleClick()}>
        Click me
      </button>
    );
  }
}
```

Use arrow function in callback

Not as performant because creates new function with each render



Function components with state

- Function components have been around in React for a long time for simple components
- Stateless before React 16.8
- Hooks were added in React 16.8
- Now most of what you could do with class components can be done with function components
- Syntax much lighter, easier to read code



Hooks in Function Components

```
import React, { useState } from 'react';

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Import `useState` hook

`count` is assigned the initial value which is zero.

`setCount` is assigned to a function used to modify the state.

Note: must use arrow function notation, otherwise if we execute `setCount(count + 1)` in the return statement it will get in an endless render loop.



Hooks in Function Components

```
const App = () => {
  const [ counter, setCounter ] = useState(0)

  const increaseByOne = () => setCounter(counter + 1)

  const setToZero = () => setCounter(0)

  return (
    <div>
      <div>{counter}</div>
      <button onClick={increaseByOne}>
        plus
      </button>
      <button onClick={setToZero}>
        zero
      </button>
    </div>
  )
}
```

If we don't want to use an arrow function in the render, declare a **const** set equal to the arrow function in the component body.

(Note in this example App component function is also using the **const** function expression form, instead of **function** keyword)



Accessing previous state explicitly in setter function

```
function Counter({ initialCount }) {
  const [count, setCount] = useState(initialCount);
  return (
    <div>
      Count: {count}
      <button onClick={() => setCount(initialCount)}>Reset</button>
      <button onClick={() => setCount((prevCount) => prevCount - 1)}>-
      </button>
      <button onClick={() => setCount((prevCount) => prevCount + 1)}>+
      </button>
    </div>
  );
}
```



useState can be used to set multiple state variables

```
function App() {  
  const [sport, setSport] = useState('basketball');  
  const [points, setPoints] = useState(31);  
  const [hobbies, setHobbies] = useState([]);  
}
```

Compare with `this.state` in class components.

Updating a state variable *replaces* the variable, in contrast to merging as in `this.setState()`



useReducer

```
const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}
```

Pass in a reducer of type `(state, action) => newState`

Returns state and a dispatch method that can be used to trigger different actions.

Useful to maintain state of complex objects

In React source `useState` is just a special case of the `useReducer` hook



useEffect hook

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Creates side effects in components

Tells component that it needs to do something *after* render

Guarantees DOM has been updated

Allows similar functionality to lifecycle methods in a class component



useEffect optimization

Second parameter is a dependency array: specifies what state variables must change to execute side effect

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]); // Only re-run the effect if count changes
```



Rules for Hooks

- ◉ Only call at Top Level (not inside loops, conditions, etc.)
- ◉ Only call from React functions **or** custom hooks (next slide)
- ◉ Many more pre-defined hooks:
<https://reactjs.org/docs/hooks-reference.html>



Custom hooks

```
import { useState, useEffect } from 'react';

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });

  return isOnline;
}
```

JS function name starts with **use**

Must follow Rules of Hooks
(from previous slide)



Using custom hooks

```
function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

Two different components
using the same hook **do**
not share state

```
function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

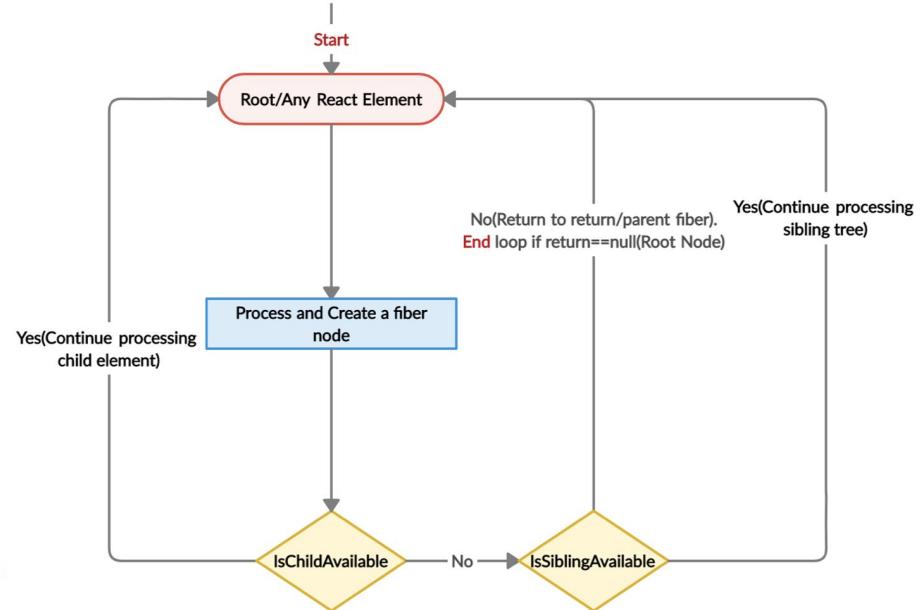
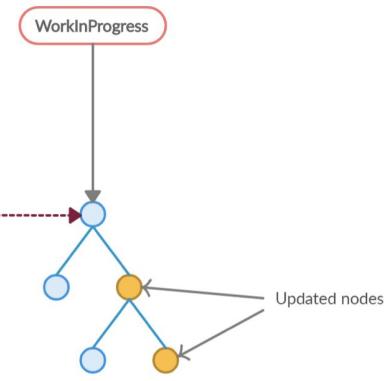
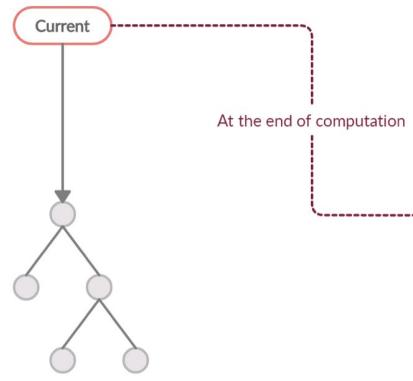


How do Hooks work under the hood?

- In React, **reconciliation** and **render** are two different phases
- **Reconciliation** is the tree diffing algorithm in React used to say what changed (virtual DOM)
- **Render** uses that info to update the app view
- Reconciliation is implemented using **fibers**
 - a JavaScript object that contains information about a component, its input, and its output
 - a kind of virtual stack frame



Fiber tree traversal





Hooks are called in render

- Hooks use a “dispatcher” object
- When you call useState, useEffect, ... it passes the call to the dispatcher object
- The renderer is what executes the component function (based on the result of reconciliation)
- The renderer knows the context of the component and works through linked list of hooks



More resources on React under the hood

- How does React actually work? (video)
 - <https://www.youtube.com/watch?v=7YhdqIR2Yzo>
- What is React Fiber?
 - <https://www.youtube.com/watch?v=OympFlwQFJw>
 - <https://blog.logrocket.com/deep-dive-react-fiber/>
- How do Hooks work?
 - <https://www.youtube.com/watch?v=1VVfMVQabx0>

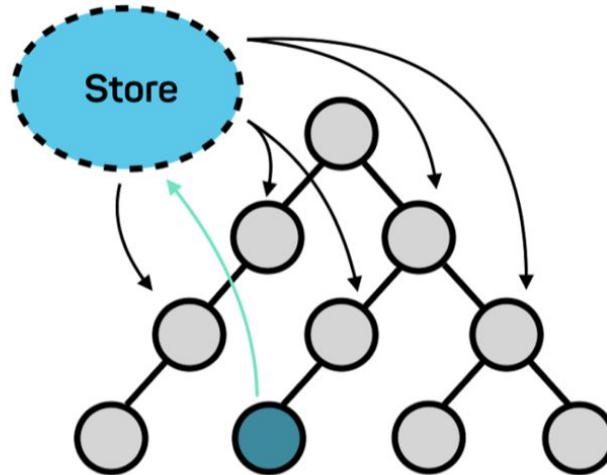
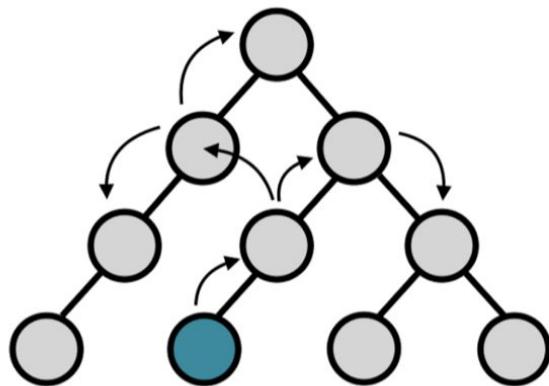


State and props revisited

- In React, **state** is internal to components
- **props** are used to pass information from parents to children
- Children can pass information to parents, by passing a parent setter function as a prop in a child element
- But what about complex apps with shared state across components not in parent-child relationship?



Global state



Component initiating change

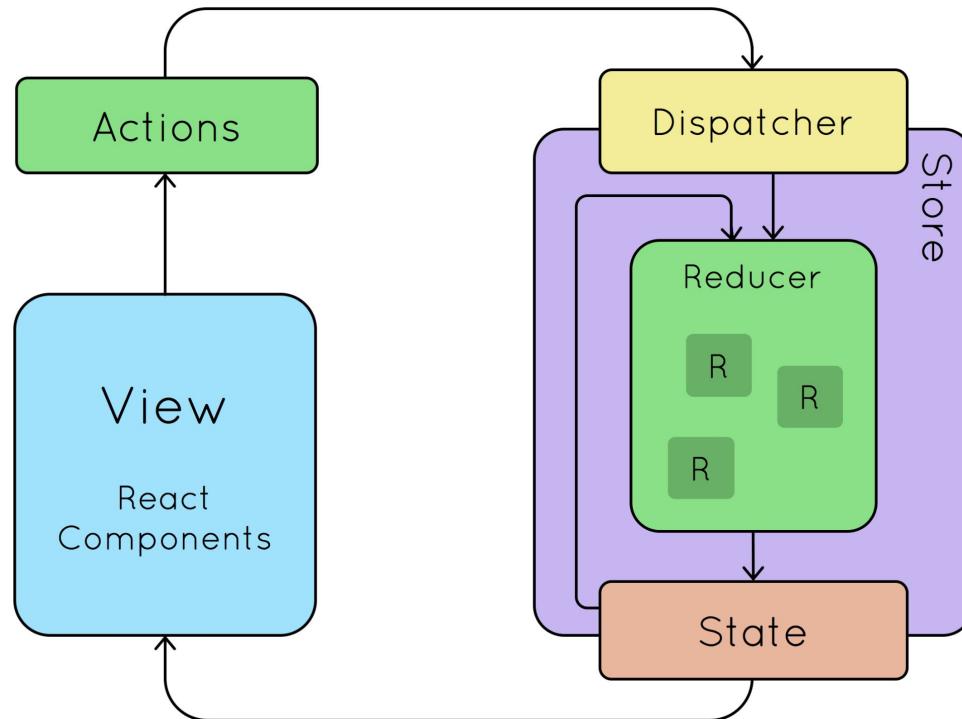


Global state management

- ◉ Several libraries exist to manage global state
- ◉ Define actions to update the state, can be async functions (e.g., including a data fetch from server)
- ◉ Redux is the most popular
 - Verbose, lots of boilerplate
 - Best to use Redux-Toolkit for React >16.8 with function components and hooks
- ◉ We use Zustand in the labs
- ◉ `useContext` hook is useful for sharing global variables such as CSS themes, but can be inefficient for large apps (triggering updates of all components)



Redux model

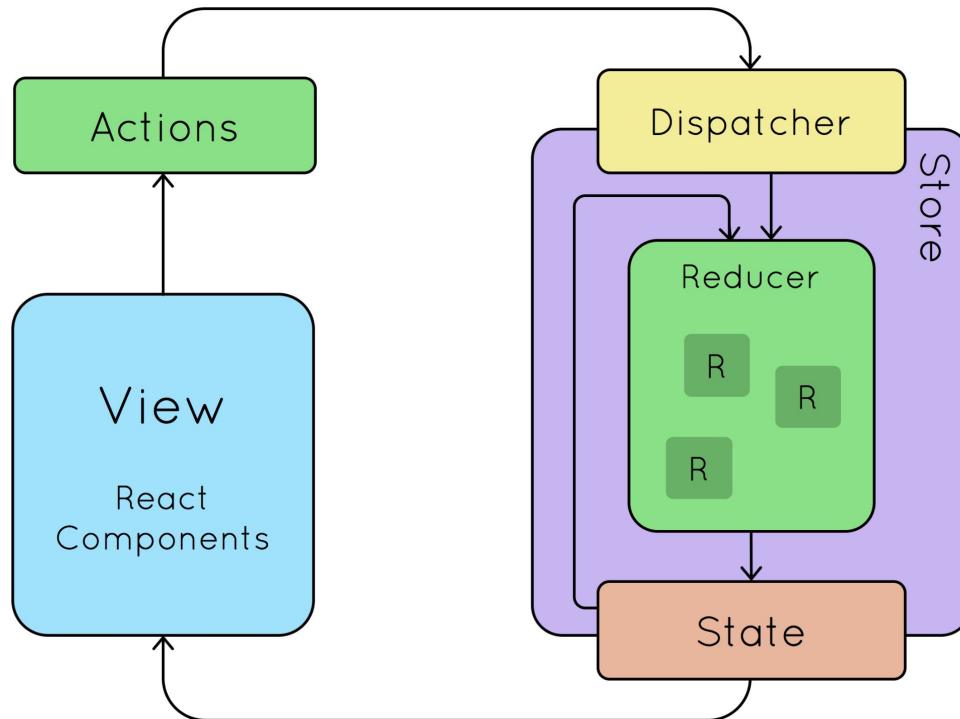


Actions = JavaScript objects that *describe* the action to be taken

```
//action to add a todo item
{ type: 'ADD_TODO', text: 'This is a new todo' }
//action that pass a login payload
{ type: 'LOGIN', payload: { username: 'foo', password: 'bar' } }
```



Redux model

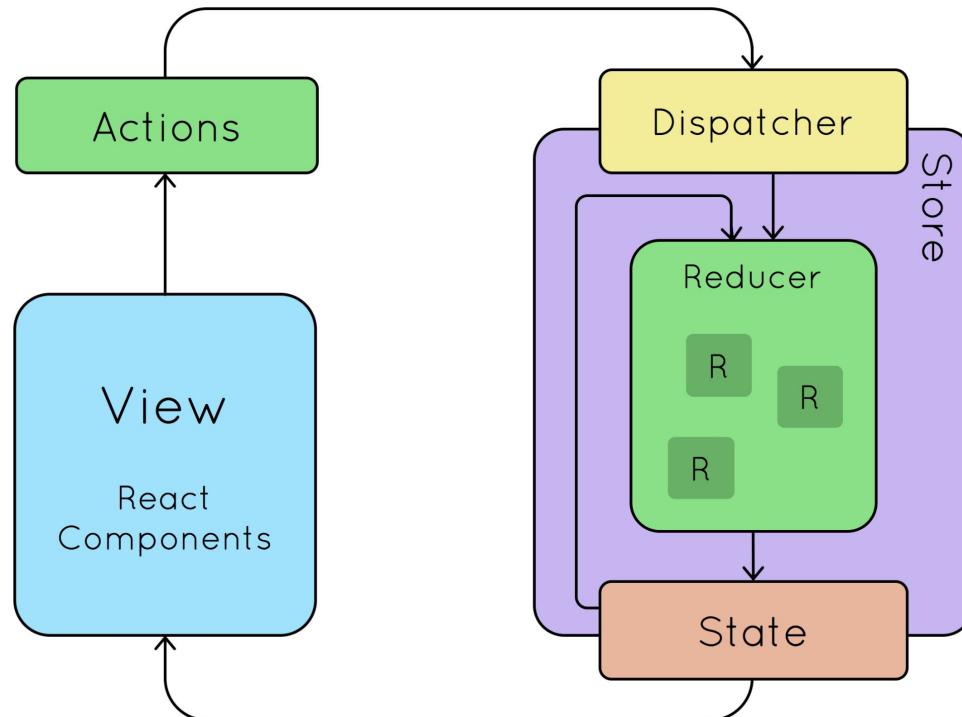


Reducer = *pure functions* that take an action and update the state

```
//takes in the current state and action
//updates the value based on the action's type
function counterReducer(state = { value: 0 }, action) {
  switch (action.type) {
    case 'INCREASE':
      return { value: state.value + 1 }
    case 'DECREASE':
      return { value: state.value - 1 }
    default:
      return state
  }
}
```



Redux model



Redux-toolkit: slices combines all this to create less boilerplate code

```
import { createSlice } from '@reduxjs/toolkit'

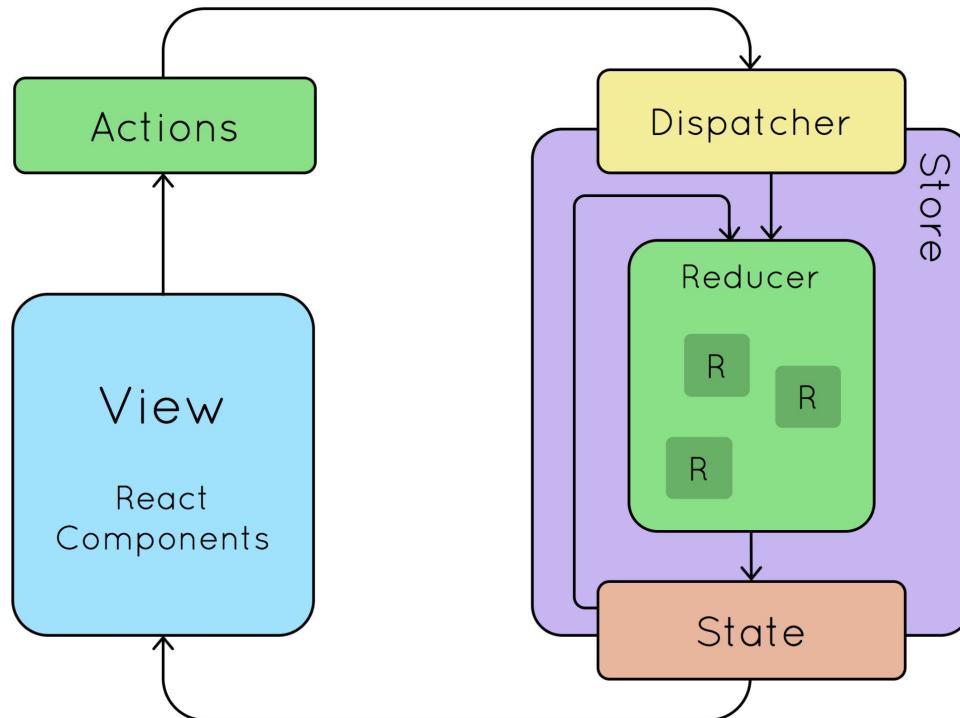
export const counterSlice = createSlice({
  name: 'counter',
  initialState: {
    value: 0
  },
  reducers: {
    increase: state => {
      state.value += 1
    },
    decrease: state => {
      state.value -= 1
    }
  }
}

// each case under reducers becomes an action
export const { increase, decrease } = counterSlice.actions

export default counterSlice.reducer
```



Redux model



Redux-toolkit: useSelector and useDispatch hooks are used in the component

```
export function Counter() {
  const count = useSelector(state => state.counter.value)
  // in our slice, we provided the name property as 'counter'
  // and the initialState with a 'value' property
  // thus to read our data, we need useSelector to return the state.counter

  const dispatch = useDispatch()
  // gets the dispatch function to dispatch our actions

  return (
    <div>
      <button onClick={() => dispatch(increase())}>
        Increase
      </button>
      <p>{count}</p>
      <button onClick={() => dispatch(decrease())}>
        Decrease
      </button>
    </div>
  )
}
```



Zustand

The screenshot shows a React application with a dark theme. On the right side, there is a floating modal window with a dark gray background and rounded corners. Inside the modal, the number '1' is displayed in large white font. Below it is a white button with the text 'one up'. To the left of the modal, the main content area contains the following code:

```
import create from 'zustand'

const useStore = create(set => ({
  count: 1,
  inc: () => set(state => ({ count: state.count + 1 })),
})) 

function Controls() {
  const inc = useStore(state => state.inc)
  return <button onClick={inc}>one up</button>
}

function Counter() {
  const count = useStore(state => state.count)
  return <h1>{count}</h1>
}
```

The code uses the Zustand library to manage a single 'count' state across components. The `useStore` hook provides a function to increment the count. The `Controls` component contains a button that triggers this increment. The `Counter` component displays the current value of the count.



useContext

1. Create a context object

```
const UserContext = createContext()
```



useContext

1. Create a context object
2. Wrap components in context Provider (user is a state variable)

```
const UserContext = createContext()  
  
<UserContext.Provider value={user}>  
  <h1>{'Hello ${user}!'}</h1>  
  <Component2 user={user} />  
</UserContext.Provider>
```



useContext

1. Create a context object
2. Wrap components in context Provider, and pass state variable (e.g. user) as value
3. In child component access the variable with useContext

```
const UserContext = createContext()

<UserContext.Provider value={user}>
  <h1>{'Hello ${user}!'}</h1>
  <Component2 user={user} />
</UserContext.Provider>

const user = useContext(UserContext);

return (
  <>
    <h1>Component 5</h1>
    <h2>{'Hello ${user} again!'}</h2>
  </>
);
```



Routing



Routing basics

- In a traditional non-SPA website, when you navigate to a new route (e.g. by clicking a link) it loads a new page from the server
- In SPA:
 - The page is not re-loaded
 - The link is intercepted and any new content needed from the server is made by client-side code
 - Do not need to re-render entire page, including framework JS (e.g. React), potentially saving data transfer/time.
 - Bonus: can update the page dynamically, e.g. using CSS transitions



React Router basic concepts

URL - URL is address bar

Location - object based on
`window.location`

History - object that
subscribes to the browser's
history stack

The history stack is changed
via POP, PUSH, REPLACE
operations

Client-side routing -
programmatically changing
the history stack without
making a server request

Route match - when the URL
matches a pattern a specific
route is rendered



Location object

pathname, search and hash
come from the url

state is the current top of the
history stack

key is a unique key (useful for
client-side caching etc.)

```
{  
  pathname: "/bbq/pig-pickins",  
  search: "?campaign=instagram",  
  hash: "#menu",  
  state: null,  
  key: "aefz24ie"  
}
```



Route config and matching

```
<Routes>
  <Route path="/" element={<App />}>
    <Route index element={<Home />} />
    <Route path="teams" element={<Teams />}>
      <Route path=":teamId" element={<Team />} />
      <Route path=":teamId/edit" element={<EditTeam />} />
      <Route path="new" element={<NewTeamForm />} />
      <Route index element={<LeagueStandings />} />
    </Route>
  </Route>
  <Route element={<PageLayout />}>
    <Route path="/privacy" element={<Privacy />} />
    <Route path="/tos" element={<Tos />} />
  </Route>
  <Route path="contact-us" element={<Contact />} />
</Routes>
```

- The route config is a tree of route elements.
- Matches can be exact or dynamic (e.g. :teamId)
- path prop defines the match
- element prop defines the component to be rendered



Higher-order Components

```
class Welcome extends React.Component {
  render() {
    return (
      <div>Welcome {this.props.user}</div>
    );
  }
}

const withUser = (WrappedComponent) => {
  return class extends React.Component {
    render() {
      if (this.props.user) {
        return (
          <WrappedComponent { ...this.props} />
        )
      }
      return <div>Welcome Guest!</div>
    }
  }
}

export default withUser(Welcome);
```

Wrapping components with
props proxy



Higher-order Components

Wrapping components with
inheritance inversion

```
class Welcome extends React.Component {
  render() {
    return (
      <div>Welcome {this.props.user}</div>
    );
  }
}

const withUser = (WrappedComponent) => {
  return class extends React.Component {
    render() {
      if (this.props.user) {
        return (
          <WrappedComponent { ...this.props} />
        )
      }
      return <div>Welcome Guest!</div>
    }
  }
}

const withLoader = (WrappedComponent) => {
  return class extends WrappedComponent {
    render() {
      const { isLoading } = this.props;
      if (!isLoading) {
        return <div>Loading...</div>;
      }
      return super.render();
    }
  }
}

export default withLoader(withUser(Welcome));
```



Debugging React Apps



Tips on debugging

- When you run in dev hot reloading is enabled
- Keep browser open with web page (e.g. `localhost:3000`) and console visible while developing
- Sometimes when it crashes (shows error message in browser) you need to reload the page



console.log

- `console.log` is still a great way to view the state of variables when components load
- Remember to remove these before production, though!

```
console.log('props value is', props)
```

Use comma, not + when concatenating complex objects to strings





debugger in Chrome developer

The screenshot shows the Chrome Developer Tools interface with the 'Sources' tab selected. A modal window is open, displaying the message "Paused in debugger" with three buttons: play (▶), step over (↑), and step into (↓). The main pane shows the code editor with a file named 'index.js'. The code contains several lines of logic involving arrays 'allClicks' and 'setAll', and functions 'handleRightClick' and 'handleLeftClick'. Line 102 contains the word 'debugger', which is highlighted in blue. The right sidebar shows the 'Debugger pause' panel with options for 'Watch', 'Call Stack', and 'App'. Below this, a list of function names is visible, including 'mountIndeterminateComponent', 'react-dom.deve', 'beginWork', 'react-dom.deve', and 'performUnitOfWork'.

```
93     setAll(allClicks.concat('L'))  setAll = f (), allClicks = []
94     setLeft(left + 1)  setLeft = f (), left = 0
95   }
96
97 const handleRightClick = () => {  handleRightClick = f handleRightClick
98   setAll(allClicks.concat('R'))  setAll = f (), allClicks = []
99   setRight(right + 1)  setRight = f (), right = 0
100  }
101
102 debugger
103
104 return (
105   <div>
106     <div>
```

Can also set
breakpoints manually.

Once it has stopped
you can use the
console to view the
value of various
variables.



React developer tools

<https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadoplbjfkapdkoienihi>

The screenshot shows the React DevTools interface within the Chrome developer tools. The Components tab is highlighted with a yellow circle. The left sidebar displays the component hierarchy: App > Header > TodoTextInput (selected), MainSection, TodoList, Footer, Link, Link, Link. The right panel provides detailed information about the selected component: props (newTodo: checked, onSave: onSave(), placeholder: What needs to be done?), hooks (State: Try React DevTools), and rendered by (Header, App).

Adds a Components tab to the developer console

Can view component state and props, and hooks in order of definition



React developer tools

<https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadoplbjfkapdkoienihi>

The screenshot shows the React DevTools interface within the Chrome developer tools. The 'Components' tab is active, indicated by a yellow circle around its tab bar. On the left, the component tree shows 'App' with 'Header', 'TodoTextInput' (selected), 'MainSection', 'TodoList', and 'Footer'. 'TodoList' has a child 'TodoItem key="0"'. The right panel shows 'props' for 'TodoTextInput' including 'newTodo' (checkbox checked), 'onSave' (function), and 'placeholder' (text). It also shows 'state' (checkbox checked) and 'hooks' (empty array). Below that, it says 'rendered by' 'Header' and 'App'.

Adds a Components and Profiler tabs to the developer console

Can view component state and props, and hooks in order of definition

Profiler allows you to test performance of components

SENG 365 Week 10

SPA Communication with Server





This week

- AJAX
- XHR
- CORS
- Web sockets



AJAX



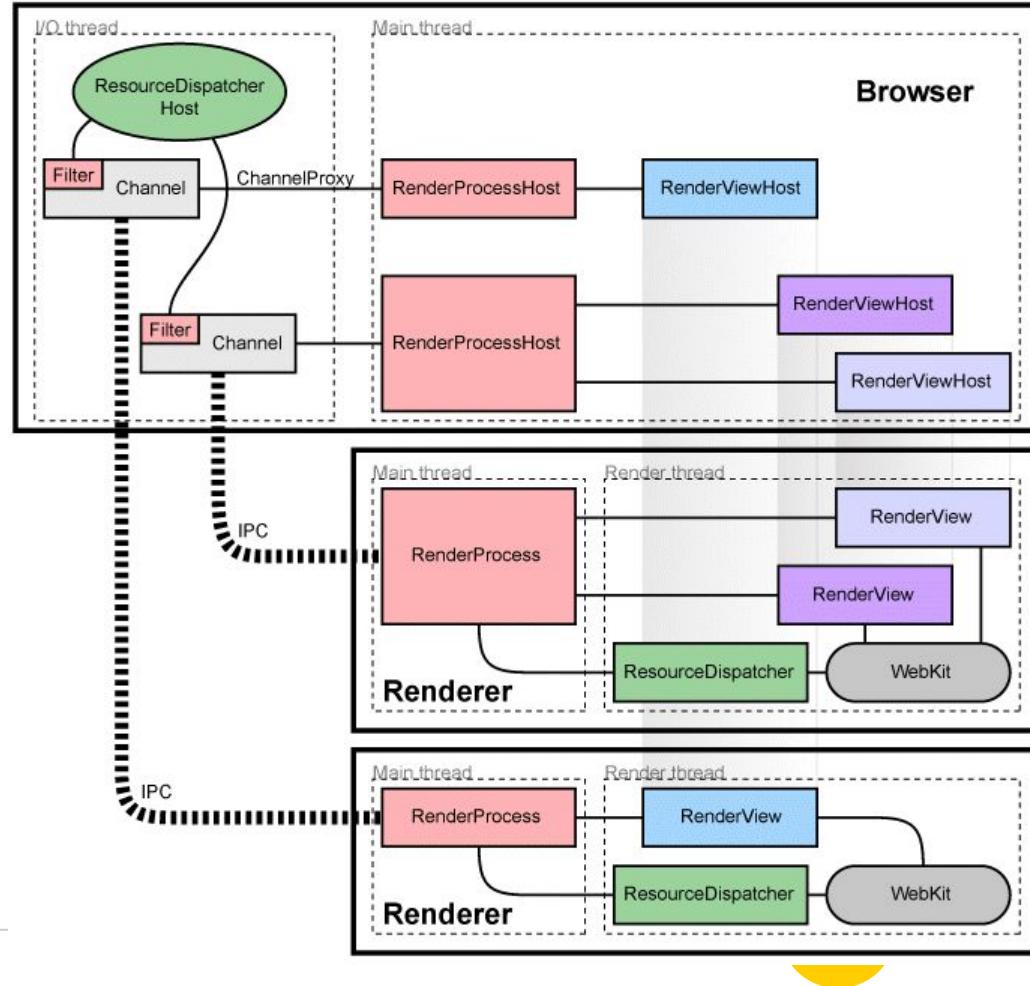
<https://xhr.spec.whatwg.org/>

AJAX: Asynchronous JavaScript and XML

XHR: XMLHttpRequest

How can you retrieve data from the server, or send data to the server, whilst the user is interacting with your webpage on the browser?







Getting data from server into SPA

- SPAs separate data from content and presentation.
- How do you get data from the server without refreshing the page?
 - How do you get data from the server without an HTTP GET of HTML?
 - How do you get data from the server without requiring a user to reload the page?
- Problem/requirement:
 - Need some way of performing HTTP requests concurrently to, and independently of, the user interacting with the application on the browser
 - Balance security and usability



Summary of XHR / AJAX

- ◉ Execute HTTP methods programmatically, and in the ‘background’
 - But remember that JavaScript is single threaded.
- ◉ Use JavaScript to issue HTTP requests e.g. HTTP GET, HTTP POST etc.
- ◉ Use XMLHttpRequest (XHR) JavaScript API
 - Raw JavaScript XHR; or
 - Abstraction of XHR e.g. jQuery’s `$.ajax()` method, axios, fetch, etc.
- ◉ Setup XHR request, in which you specify things like:
 - HTTP request
 - Method e.g. GET, etc. including ? query parameters
 - Body e.g. what’s in the HTTP body you’re sending (of anything)
 - Expected HTTP responses: what data you want back from the server e.g. JSON, XML etc
 - Callbacks for handling the range of HTTP response/s
 - e.g. successful | unsuccessful response
- ◉ Execute the request

```
<div>
  <p><strong>Example data returned.</strong></p>
</div>
<div>
  <div id="pid">Nothing.</div>
  <button type="button" onclick="go()">Click me.</button>
</div>
<script>
  function go(){
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function() {
      if (this.readyState == 4 && this.status == 200) {
        document.getElementById("pid").innerHTML =
          this.responseText;
      }
    };
    xhttp.open("GET", "https://www.canterbury.ac.nz", true);
    xhttp.send();
  }
</script>
```

Example of raw XHR

1. Copy and paste into HTML file.
2. Add the DOCTYPE, <html>, <head>, and <body> elements
3. Open in a browser window.
4. Enjoy responsibly...

NOTE

1. The code does not handle unsuccessful responses...



Events (handlers) for XHR

onloadstart

onprogress

onabort

onerror

onload

ontimeout

onloadend

onreadystatechange



XMLHttpRequest ++

- ◉ The term **XMLHttpRequest** (XHR) is (now) misleading:
 - Can retrieve data other than XML e.g. JSON
 - Works with other protocols, not just HTTP
 - Doesn't have to be asynchronous
 - ... but should be asynchronous and should NOT synchronous
- ◉ Browser differences (in older browsers)
 - Internet Explorer variants do things differently...
 - XDomainRequest in Internet Explorer 8 and 9
 - Best to use an abstraction rather than raw XHR e.g. a library such as **axios** or **fetch API**

fetch API

- ◉ Native Javascript API introduced in 2017
- ◉ Implemented in most modern browsers now
 - https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API#browser_compatibility

```
fetch('examples/example.json')
  .then(function(response) {
    // Do stuff with the response
  })
  .catch(function(error) {
    console.log('Looks like there was a problem: \n', error);
});
```

fetch API

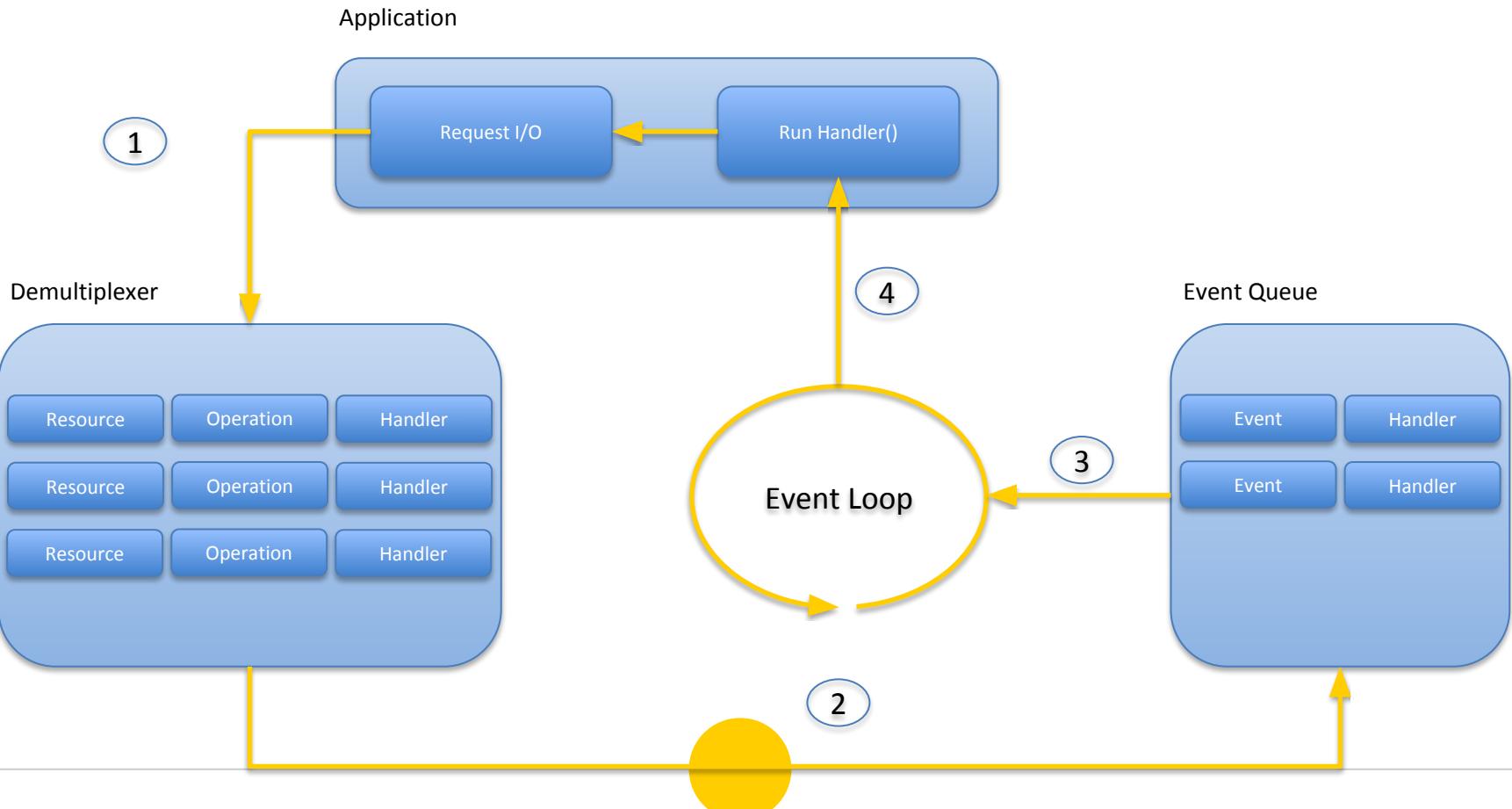
- Native Javascript API introduced in 2017
- Implemented in most modern browsers now
 - [https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API#
browser_compatibility](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API#browser_compatibility)

```
async function doAjax() {  
    try {  
        const res = await fetch('send-ajax-data.php');  
        const data = await res.text();  
        console.log(data);  
    } catch (error) {  
        console.log('Error:' + error);  
    }  
}  
  
doAjax();
```

Works with async functions

*How does an application respond
to multiple overlapping requests?*

A yellow circle containing two black double quotes ("") is positioned above a vertical line.



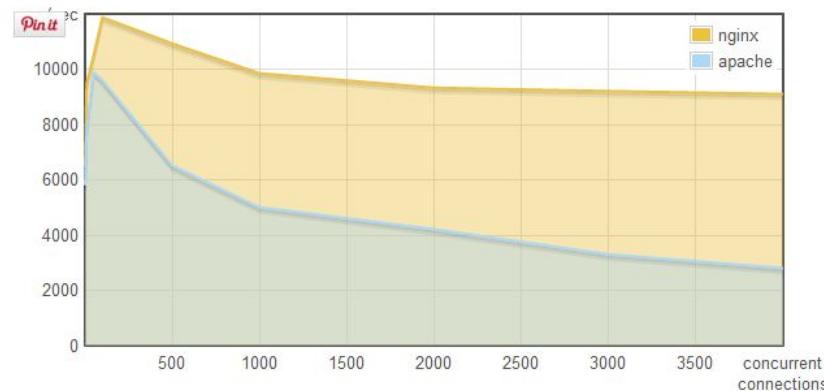
A little holiday present: 10,000 reqs/sec with Nginx!

Posted in [Server setup](#) December 18, 2008 by Remi D

Updated Dec 19 at 05:15 CDT (first posted Dec 18 at 06:01 CDT) by Remi

A few weeks ago we quietly started to configure our new machines with Nginx as the front web server instead of Apache (we still run Apache behind Nginx for people who need all the features from Apache).

Here is a little benchmark that I did to compare Nginx versus Apache (with the worker-MPM) for serving a small static file:



This benchmark is not representative of a real-world application because in my benchmark the web servers were only serving a small static file from localhost (in real life your files would get served to



Concurrency (vs Parallelism)

"In programming, concurrency is the composition of independently executing processes, while parallelism is the simultaneous execution of (possibly related) computations. **Concurrency** is about **dealing** with lots of things at once. **Parallelism** is about **doing** lots of things at once."

– Concurrency is not Parallelism, the GoLang blog



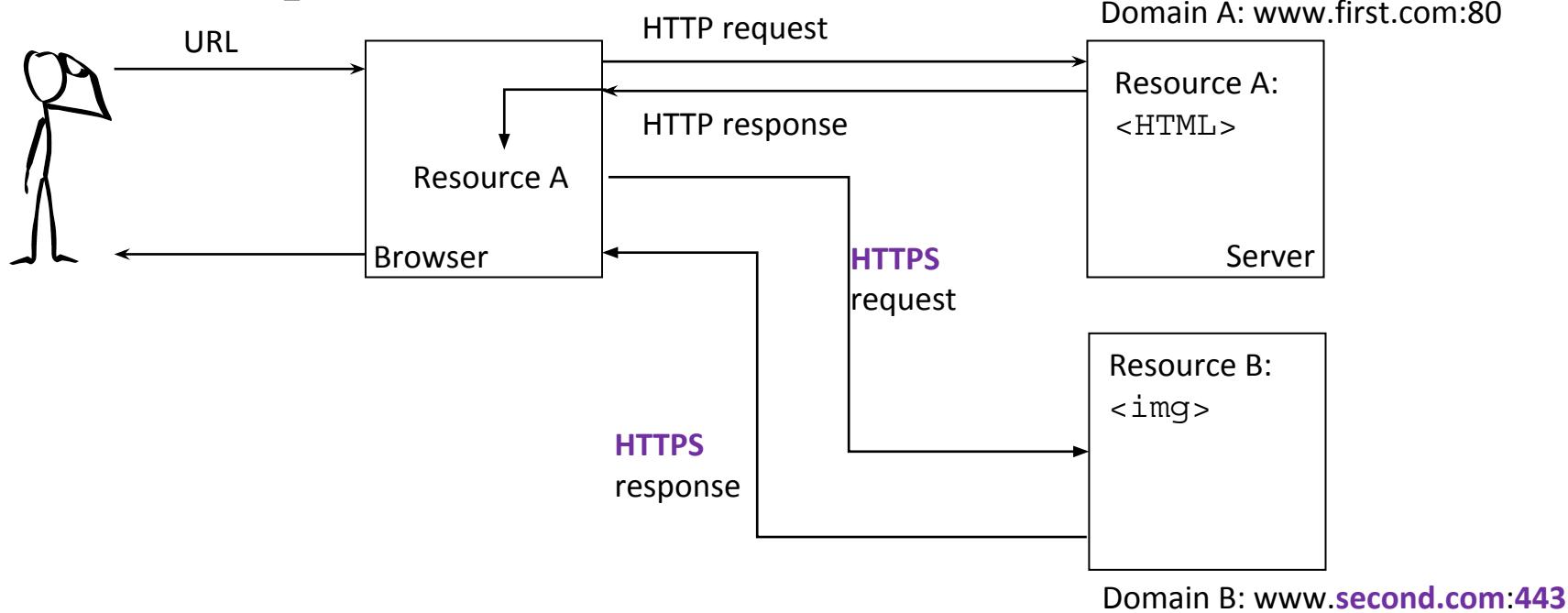
CORS

Cross Origin Resource Sharing

```
app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*");
  res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With,
Content-Type, Accept");
  res.header("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE");
  next();
});
```



Everyday & legitimate cross-origin resource request

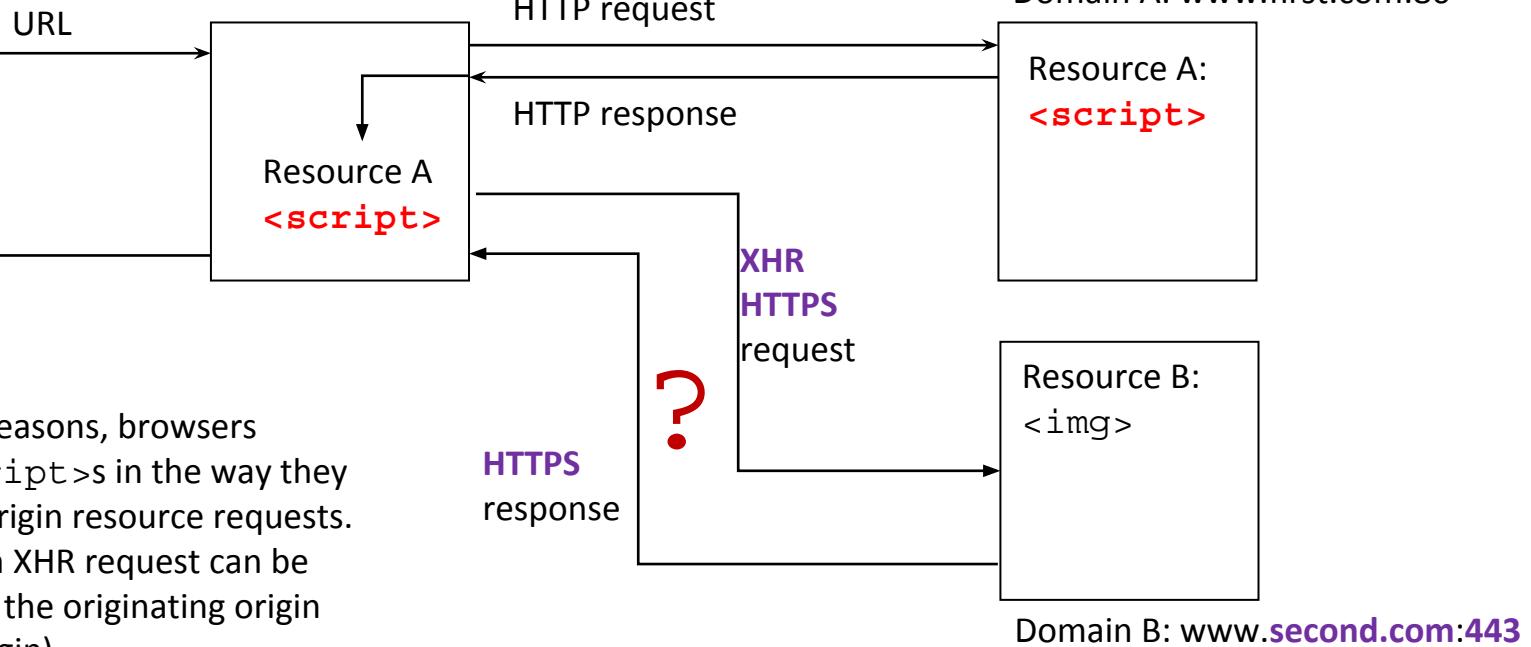




Cross-origin requests with `<script>`



For security reasons, browsers restrict `<script>`s in the way they make cross-origin resource requests. By default, an XHR request can be made only to the originating origin (the same origin).



Sending and retrieving resources

- Many web pages (web applications) load resources from *separate* domains
 - CSS stylesheets, images, frames, video
- Certain "cross-domain" requests, notably **AJAX / XHR** requests, are **forbidden** by the same-origin security policy.
 - AJAX requests are JavaScript and <script>s can't by default make cross-origin requests

CORS to the rescue

- CORS defines a way in which **a browser** and **a server** can **work together** to determine whether or not it is safe to allow a **client-side app** to make a cross-origin request
- The CORS standard describes HTTP headers which provide browsers and servers a way to request remote URLs only when they have permission.
 - The app can't access (some of) these headers.
- Some validation and authorization is performed by the server
 - The server specifies the acceptable origins of the HTTP requests

It is generally the browser's responsibility to support these headers and honour the restrictions they impose.

Note: **not** the application's responsibility; the browser's responsibility e.g. to prevent the application doing something

What defines 'origin'?

Origin is defined in terms of

- Protocol
- Domain
- Port

Identical {protocol, domain, port}
= same origin

Different {protocol, domain, port}
= cross-origin

So:

`http://example.com:80`

is different from:

`https://example.com:80`

`http://exemple.com:80`

`http://example.com:463`

HTTP forbidden headers

- Some headers are managed by the browser and/or the server, and...
- ... these headers can't be manipulated by the client-side application.
- "A forbidden header name is an HTTP header name that cannot be modified programmatically; specifically, an HTTP request header name."

Examples of forbidden headers

The browser should prevent your application from modifying these:

Access-Control-*

Access-Control-Origin

Access-Control-Headers

Origin

There are many others

https://developer.mozilla.org/en-US/docs/Glossary/Forbidden_header_name

Some of the headers returned by the server

Access-Control-Allow-Origin: *

Allow the request from any origin

Access-Control-Allow-Origin: <https://foo.bar>

Allow the request only from resources originating from [HTTPS://foo.bar](https://foo.bar)
(remember what defines an origin)

Access-Control-Allow-Methods: POST, GET, OPTIONS

Allow only these HTTP methods

Access-Control-Max-Age: 86400

Access is allowed for up to 86400 milliseconds (86.4s)

Access-Control-Allow-Credentials: true

Required if you want to send cookies etc
(and can't use

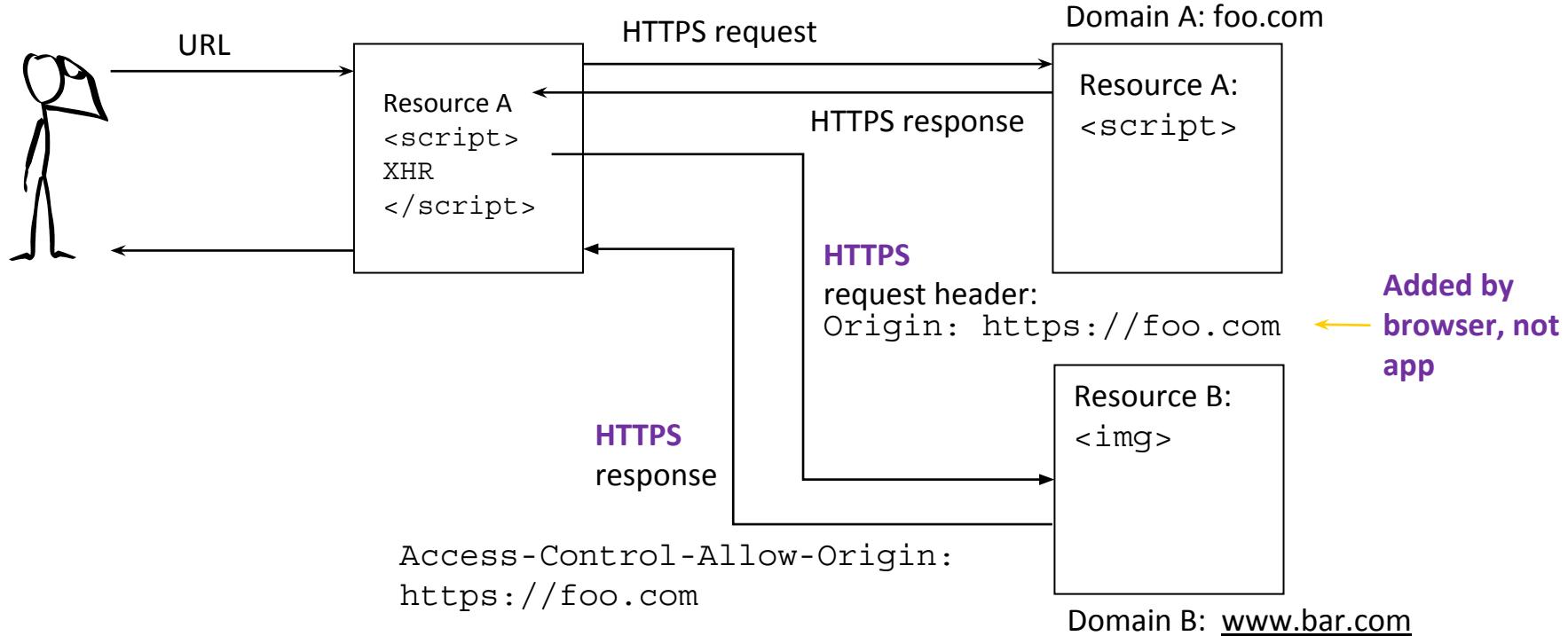
Access-Control-Allow-Origin: *
)

Adding CORS headers to server response in node.js

```
app.use(function(req, res, next) {
    res.header("Access-Control-Allow-Origin", "*");
    res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With,
Content-Type, Accept");
    res.header("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE");
    next();
});
```

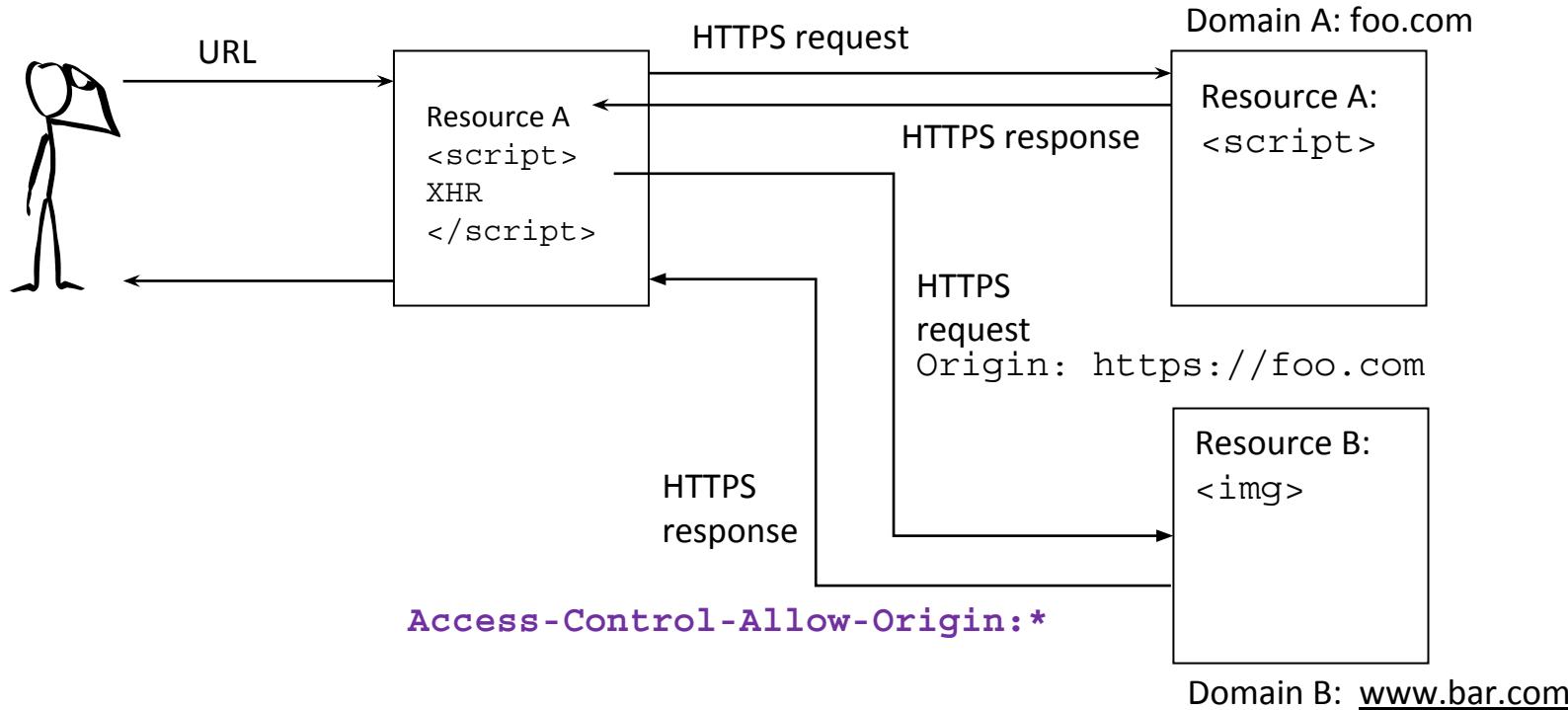


Example 1: accept request from foo.com only



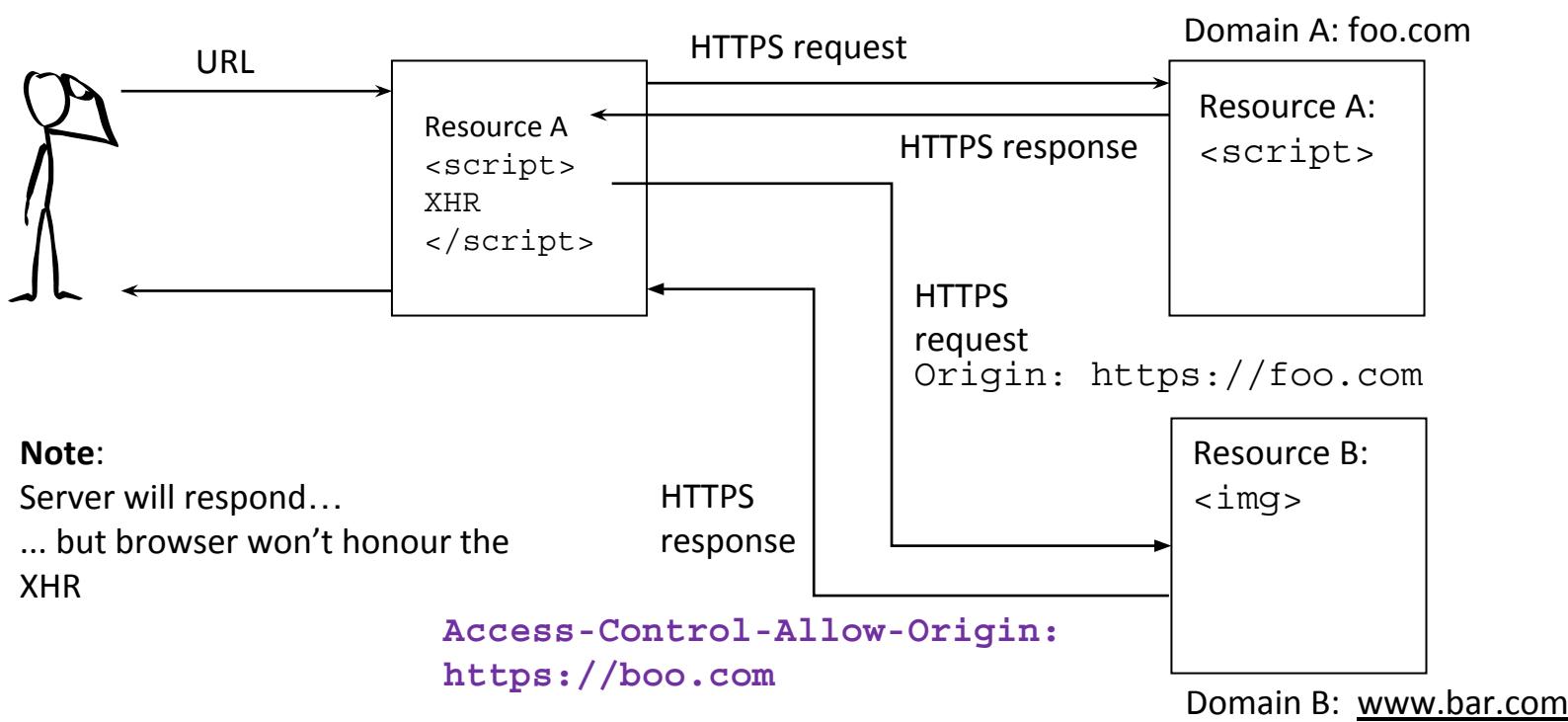


Example 2: accept request from anywhere (*)



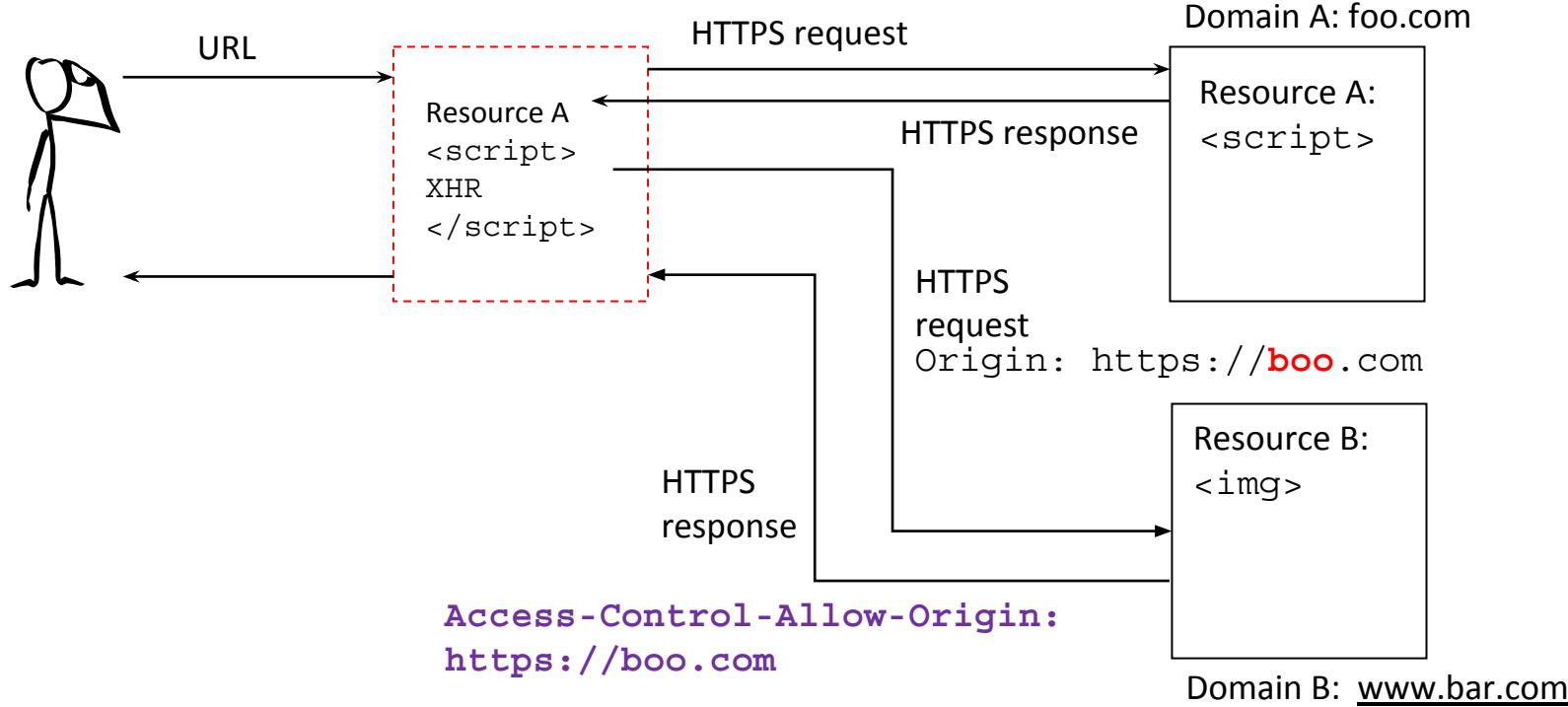


Example 3: reject request





Example 4: fooling the server?



Worked example using Flickr

- Follow-up reading
- Code example from *CORS in Action* (Manning Publications)

Chapter 1. The Core of CORS - CORS in Action: Creating and consuming cross-origin APIs

WebSockets



WebSockets vs HTTP AJAX

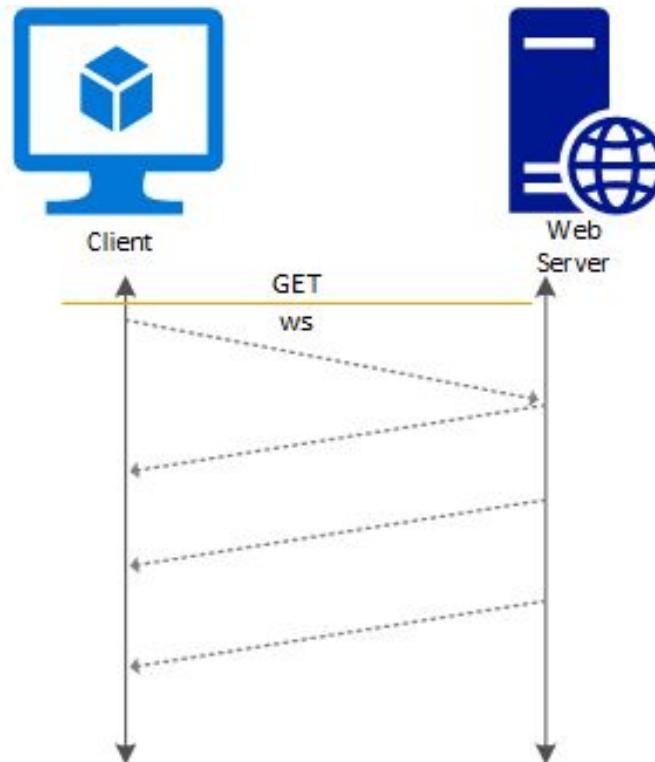
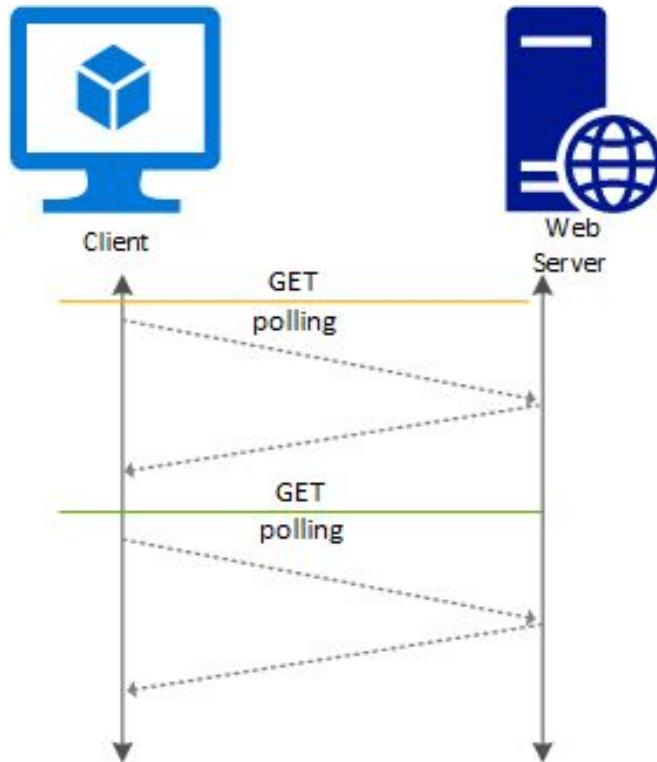
HTTP, AJAX

- ◉ Uni-directional communication: client makes request, server makes response
- ◉ Stateless
- ◉ Relatively less efficient
- ◉ Good for retrieving resources
- ◉ Must implement polling to get updates from the server

Web socket

- ◉ Persistent 2-way communication
- ◉ Maintains state
- ◉ Fast, lightweight and can maintain much higher load of connections
- ◉ Good for real-time communication: chats, games, etc.
- ◉ Server can push data to client

Polling vs pushing



How are **web socket connections** made?

Handshake mechanism

1. Client send initial HTTP GET request to the server.
2. Server responds with information on how to connect to socket server.
3. Client sends HTTP GET with header “Connection: Upgrade” and connection is upgraded to a socket connection.

Handshake messages

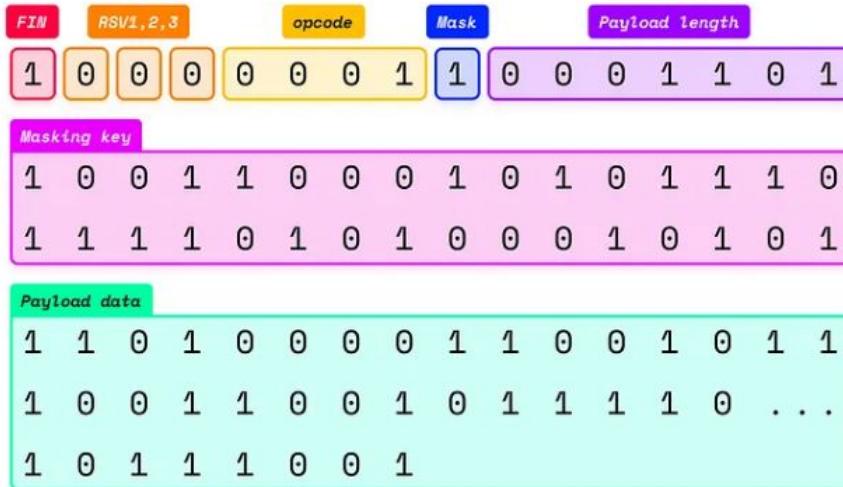
Request

```
GET /chat HTTP/1.1
Host: server.example.com
Connection: upgrade
Upgrade: websocket
Origin: <http://example.com>
Sec-WebSocket-Key: NnRlZW4gYnl0ZXMgbG9uZw==
Sec-WebSocket-Protocol: html-chat, text-chat
Sec-WebSocket-Version: 13
```

Response

```
HTTP/1.1 101 Switching Protocols
Connection: upgrade
Upgrade: websocket
Sec-WebSocket-Accept: 5TJpHv9RoA17w8ytsXcWxT0Z9Q==
Sec-WebSocket-Protocol: new-chat
```

Open connection sends **data frames** between server and client



- **FIN** – last frame of message boolean bit
- **RSV** – extension bits
- **opcode** – how message is interpreted (data frame or control frame)
- **mask** – data encrypted boolean bit
- **masking key** – key used to encrypt data

Web socket API

- ◉ Web sockets are now supported in most browsers
 - <https://caniuse.com/websockets>
- ◉ Implemented in npm packages, e.g. `ws`, `socket-io`, `websocket`, and `express-ws` (express integration with `ws`)
 - Some libraries implement fallback (emulates socket connection in http when not supported in browser)

Sample server

Simple server

```
const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 8080 });

wss.on('connection', function connection(ws) {
  ws.on('message', function incoming(message) {
    console.log('received: %s', message);
  });
  ws.send('something');
});
```

Sample client connection

Sending and receiving text data

```
const WebSocket = require('ws');

const ws = new WebSocket('ws://www.host.com/path');

ws.on('open', function open() {
  ws.send('something');
});

ws.on('message', function incoming(data) {
  console.log(data);
});
```

SENG 365 Week 11

Web Storage and Progressive Web Apps



Assignment 2

Submission requirements

- Zipped project
- Use username as zip filename e.g.,
`<usercode>.zip`
- Without
`node_modules` to reduce size
- No server code

During the test

- Preparation
 - Download .zip of apps
 - Download latest version of server, install it and run it
- For each application
 - Reset the database
 - First /reset and /resample database using Postman
 - Install and run the client-side app
 - npm install and run
 - Open Chrome: localhost:<port>
- Run tests



Fragment from 2017's test script

1. Can the application be run?	YES	NO	
2. Project views			
Precondition: not logged in, viewing a list of sample projects (you might need to navigate to get to this point)			
Steps	Expected	Pass	Fail
1. Page or scroll to see all the sample projects	Should be able to see at least 12 projects (perhaps after scrolling or paging)		
2. Find a search box, and search for farm	Two projects: Let's Raise the Roof (Farm)! and The Farmery should be shown in the project view		
3. Select project The Farmery for detailed view	Extra information for project should be shown including rewards, progress towards goal, backers and pledges		
Totals			
Comments			

Application Test

*Required

Section 2 - Project Views

Precondition : Not logged in

Viewing a list of sample projects (you might need to navigate to this point)

1. Action : page or scroll to see all the sample projects *

Pass Fail Can't answer

Expected result :
Should be able to see
at least 12 projects
(perhaps after scrolling
or paging)

Comments

Your answer

2. Action : find a search box, and search for "farm" *

Pass Fail Can't answer

Expected result :
Should be able to see
at least 12 projects
(perhaps after scrolling
or paging)

Expected result :
Should be able to see
at least 12 projects
(perhaps after scrolling
or paging)

Comments

Your answer

3. Action : select the project "The Farmery" for detailed view *

Pass Fail Can't answer

Expected result : Two
projects: "Let's Raise
the Roof (farm) I" and
"The Farmery" should
be shown in the project
view

Comments

Your answer

General comments for this section

Your answer

BACK NEXT

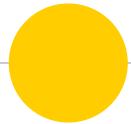
Page 2 of 6

Never submit passwords through Google Forms.



This week

- Web storage
- IndexedDB
- CacheStorage
- Progressive Web Apps
- Service Workers
- Web Assembly



Web Storage

Session Storage, Local Storage, IndexedDB



HTML5 Web Storage

- Cookies have limitations
 - Size limited to approx. 4KB
 - Send to the server with each request
- HTML5 web storage
 - Allows up to approx. 5MB (depends on browser implementation)
 - Two types:
 - Local storage
 - Session storage
- Not more secure than cookies, however



Local Storage

- Stores **permanent** data for your site (i.e., no expiration date)
- Stores data in **key, value pairs**
- Keys and values are **Strings**
- **Setter** and **getter** functions:

```
localStorage.setItem(key, value)
```

```
localStorage.getItem(key)
```



Local Storage example

```
<script>
// Check if the localStorage object exists
if(localStorage) {
    // Store data
    localStorage.setItem("first_name", "Peter");

    // Retrieve data
    alert("Hi, " + localStorage.getItem("first_name"));
} else {
    alert("Sorry, your browser do not support local storage.");
}
</script>
```



LocalStorage + Zustand

```
1 const useStore = create((set) => ([])
2   |   collection: getLocalStorage("collection") || [],
3   |   setCollection: (collection) =>
4   |     set((state) => {
5   |       |   setLocalStorage("collection", collection);
6   |       |   return { collection };
7   |     })
8   |));
9
10 const getLocalStorage = (key) => JSON.parse(window.localStorage.getItem(key));
11 const setLocalStorage = (key, value) =>
12   |   window.localStorage.setItem(key, JSON.stringify(value));
```

Read **collection** from local storage when the Zustand store is created, or default to []

When **collection** is set in Zustand, also update local storage

Wrappers for Local Storage getters and setters



Session Storage

- Stores ***temporary*** data for your site
- Deleted** when the session ends (browser or tab is closed by user)
- Same getters and setters** as Local storage

```
<script>
// Check if the sessionStorage object exists
if(sessionStorage) {
    // Store data
    sessionStorage.setItem("last_name", "Parker");

    // Retrieve data
    alert("Hi, " + localStorage.getItem("first_name") + " " +
sessionStorage.getItem("last_name"));
} else {
    alert("Sorry, your browser do not support session storage.");
}
</script>
```



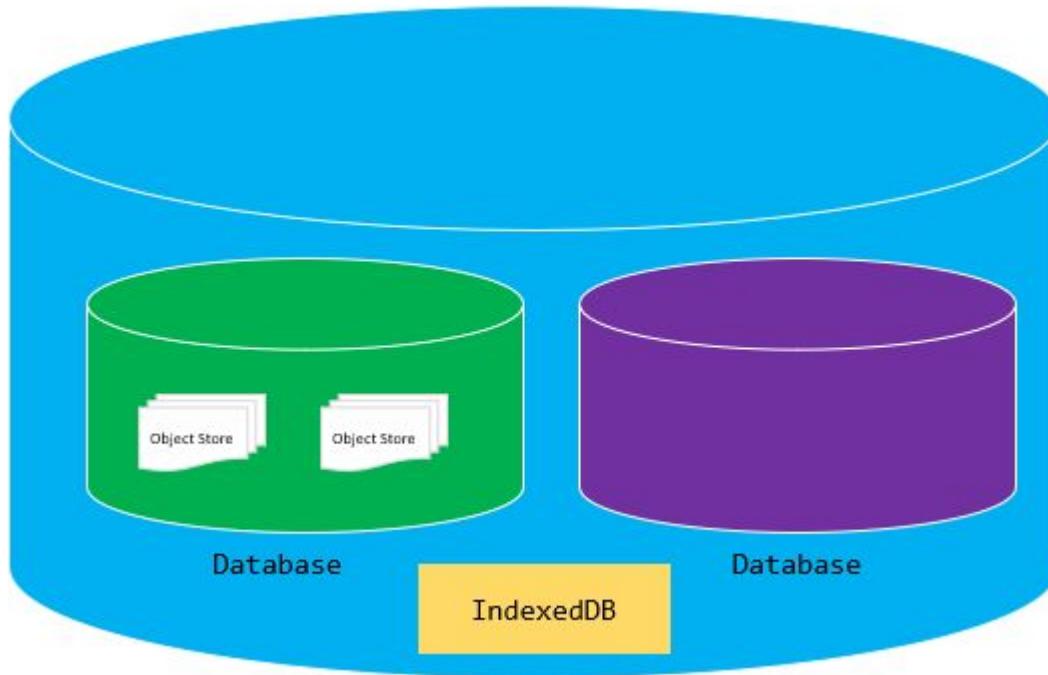
IndexedDB

- Web API for creating indexed NoSQL databases in browser for a web page
- Can create multiple object stores
- Primary keys
- Indexes
- CRUD requests are asynchronous using promises
- Follows same-origin policy (see CORS)

See: <https://developers.google.com/web/ilt/pwa/working-with-indexeddb>



IndexedDB



One or more databases

Each database made up of one or more Object Stores



IndexedDB elements

The screenshot shows the IndexedDB developer tools interface. On the left, a sidebar lists storage types: Local Storage, Session Storage, and Indexed DB. Under Indexed DB, it shows origins: developer.microsoft.com and WebAudioDemo. Under developer.microsoft.com, there are databases: configs, author, genre, and name. A context menu is open over the 'name' database entry, listing Refresh, Delete item, Copy selected items, and Select all. The main area displays an object store named 'name' with two entries: 'Jelly' (Key 1) and 'Ketchup' (Key 2). The 'Value' column shows JSON objects representing metadata for each item.

Origin

Databases

Object stores

Indices

Refresh

Object store entries list

Context menu

Key	Primary Key	Value
"Jelly"	1	{"metadata":{"name":"Jelly","author":"PB","ge...}}
"Ketchup"	2	{"metadata":{"name":"Ketchup","author":"DJ...}}



IndexedDB Object Store

- Conceptually similar to database table, but NoSQL
- Records – key, value pairs
- Create indexes on object stores
- Read/write is transactional



indexedDB.open callback functions

1. **onerror** - handles an error when opening database
2. **onsuccess** - should execute your transactions in this callback
3. **onupgradeneeded** - executed when a new database is created
 - a. Create Object Stores here
 - b. “New database” includes a new version number



IndexedDB opening a database

```
if (!window.indexedDB) {  
    console.log(`Your browser doesn't support IndexedDB`);  
    return;  
}  
  
const request = indexedDB.open('contacts-db', 1);
```

Check if the browser supports
IndexedDB

Note: `indexedDB.open` returns a Promise.
Can use `await` syntax here.



IndexedDB onerror, onsuccess

```
request.onerror = (event) => {
  console.error(`Database error: ${event.target.errorCode}`);
}

request.onsuccess = (event) => {
  const db = event.target.result;

  // do db transactions here
}
```



IndexedDB onupgradeneeded

```
request.onupgradeneeded = (event) => {
  let db = event.target.result;

  // create the Contacts object store
  // with auto-increment id
  let store = db.createObjectStore('Contacts', {
    autoIncrement: true
  });

  // create an index on the email property
  let index = store.createIndex('email', 'email', {
    unique: true
  });
}
```

Create a new object store called Contacts and give it an auto incrementing key

Create an index on the email field and make sure that each email is unique. It will throw an error if you attempt to add 2 records with the same email.



IndexedDB onsuccess populating the database

```
request.onsuccess = (event) => {
  const db = event.target.result;

  insertContact(db, {
    email: 'john.doe@outlook.com',
    firstName: 'John',
    lastName: 'Doe'
  });

  insertContact(db, {
    email: 'jane.doe@gmail.com',
    firstName: 'Jane',
    lastName: 'Doe'
  })
}
```

```
function insertContact(db, contact) {
  // create a new transaction
  const txn = db.transaction('Contacts', 'readwrite');

  // get the Contacts object store
  const store = txn.objectStore('Contacts');
  let query = store.put(contact);

  //handle success case
  query.onsuccess = (event) => {
    console.log(event);
  }

  // handle the error case
  query.onerror = (event) => {
    console.log(event.target.error);
  }

  // close the database once the transaction completes
  txn.oncomplete = () => {
    db.close();
  };
}
```



Cache Storage API

- Stores pairs of Request and Response objects
- Cache can be hundreds of megabytes in size
- Access cache: `const cache = await caches.open('my-cache');`
- Adding to cache
 - add – `cache.add(new Request('/data.json'));`
 - addAll – `const urls = ['/weather/today.json', '/weather/tomorrow.json'];
cache.addAll(urls);`
 - put – `cache.put('/test.json', new Response('{"foo": "bar"}'));`
- Retrieving from cache

```
const response = await cache.match(request);  
console.log(request, response);
```



Progressive Web Apps



Progressive web apps

- Web applications designed to appear to be ‘installed’ as native applications
- Begin life in a browser tab...
- ... then can be ‘installed’ as native apps
- Rely on Service Workers
 - Notifications
 - Background sync (offline cache)

For more:

<https://developers.google.com/web/progressive-web-apps/checklist>



Service Workers

- ◉ Proxy ‘servers’ that sit between web applications, and the browser and network
- ◉ JavaScript that:
 - runs on its own thread: not blocking
 - is headless (no access to DOM)
- ◉ Rely on HTTPS, for security
- ◉ Associated with specific server/website

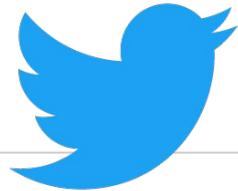
For more:

<https://developers.google.com/web/fundamentals/primers/service-workers>

Devices running SPAs

- Many form factors and connection types
- All have browsers, but also apps
- One option is to develop web sites and native apps in parallel
- Alternative is progressive web apps





Twitter PWA

- ◉ On mobile originally had separate website for in browser and an installable native app from app store.
- ◉ Migrated to progressive web app on mobile in 2020
- ◉ Advantages cited:
 - Much smaller size
 - Adaptable (no need for app store approvals when making changes)
 - Automatic updates
 - New operating systems
 - Faster, more efficient development



Criteria for Progressive

- ◉ Responsive
- ◉ Connectivity independent
- ◉ App-like interactions
- ◉ Fresh
- ◉ Safe
- ◉ Discoverable
- ◉ Re-engagable
- ◉ Installable
- ◉ Linkable



PWA “Good to haves”

- Mobile-friendly design
- Near-instant loading
 - Interactive in less than 5 sec before Service Worker installed
 - Once Service Worker installed should load in < 2 sec
- Work across devices & browsers
 - 90%+ of all users in market
- Fluid animations
 - Visual transitions



Technical definition of a PWA

- Originate from a **Secure Origin**
- **Load while offline**
- Reference a **Web App Manifest**
 - W3C spec defining a JSON-based manifest
 - [Web app manifests | MDN](#)
 - `name`
 - `short_name`
 - `start_url`
 - `display`
 - Icon – at least 144x144 px in PNG format



Example manifest

Deployed in HTML using a link tag

```
<link rel="manifest" href="/manifest.json">
```

```
{
  "short_name": "Weather",
  "name": "Weather: Do I need an umbrella?",
  "icons": [
    {
      "src": "/images/icons-vector.svg",
      "type": "image/svg+xml",
      "sizes": "512x512"
    },
    {
      "src": "/images/icons-192.png",
      "type": "image/png",
      "sizes": "192x192"
    },
    {
      "src": "/images/icons-512.png",
      "type": "image/png",
      "sizes": "512x512"
    }
  ],
  "start_url": "/?source=pwa",
  "background_color": "#3336D6",
  "display": "standalone",
  "scope": "/",
  "theme_color": "#3336D6",
  "shortcuts": [
    {
      "name": "How's weather today?",
      "short_name": "Today",
      "description": "View weather information for today",
      "url": "/today?source=pwa",
      "icons": [{ "src": "/images/today.png", "sizes": "192x192" }]
    },
    {
      "name": "How's weather tomorrow?",
      "short_name": "Tomorrow",
      "description": "View weather information for tomorrow",
      "url": "/tomorrow?source=pwa",
      "icons": [{ "src": "/images/tomorrow.png", "sizes": "192x192" }]
    }
  ],
  "description": "Weather forecast information",
  "screenshots": [
    {
      "src": "/images/screenshot1.png",
      "type": "image/png",
      "sizes": "540x720"
    },
    {
      "src": "/images/screenshot2.jpg",
      "type": "image/jpg",
      "sizes": "540x720"
    }
  ]
}
```

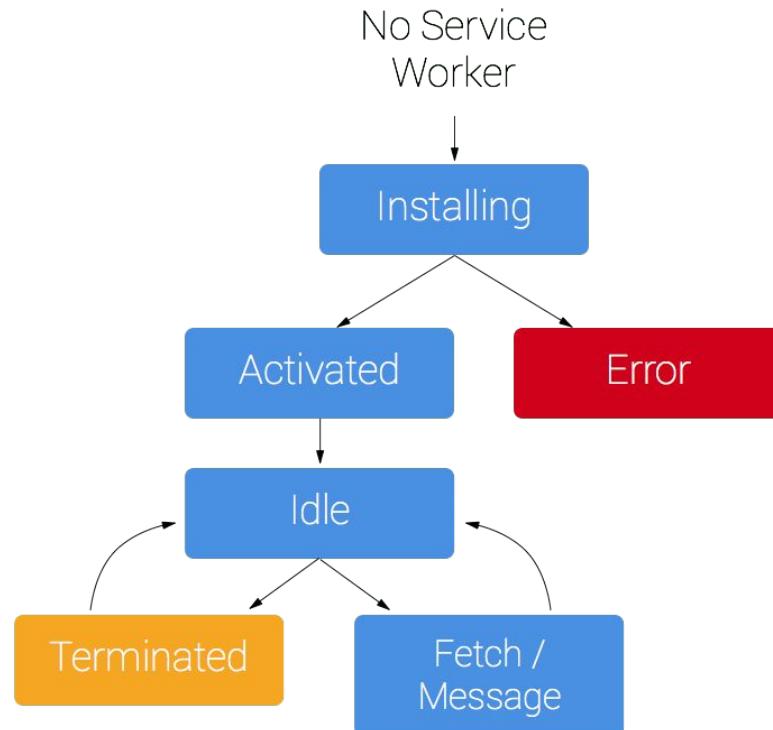


Service Worker

- ◉ Javascript that runs in the background
- ◉ Can be used to execute long-running processes
- ◉ Must be started/registered by a web page
- ◉ Allows websites to run offline by serving cached data
- ◉ Must be served over https



Service Worker lifecycle





Registering a Service Worker

- `serviceWorker.register` registers the service worker (js file) for this site
- `scope` defines subset of content that Service Worker controls
- Max scope is the location of the worker

```
const registerServiceWorker = async () => {
  if ('serviceWorker' in navigator) {
    try {
      const registration = await navigator.serviceWorker.register(
        '/sw-test/sw.js',
        {
          scope: '/sw-test/',
        }
      );
      if (registration.installing) {
        console.log('Service worker installing');
      } else if (registration.waiting) {
        console.log('Service worker installed');
      } else if (registration.active) {
        console.log('Service worker active');
      }
    } catch (error) {
      console.error(`Registration failed with ${error}`);
    }
  }
};

// ...

registerServiceWorker();
```



Worker Install event

- When `install` event is triggered, the worker can cache files in its scope
- `self` refers to the worker

```
const addResourcesToCache = async (resources) => {
  const cache = await caches.open("v1");
  await cache.addAll(resources);
}

self.addEventListener("install", (event) => {
  event.waitUntil(
    addResourcesToCache([
      "/sw-test/",
      "/sw-test/index.html",
      "/sw-test/style.css",
      "/sw-test/app.js",
      "/sw-test/image-list.js",
      "/sw-test/star-wars-logo.jpg",
      "/sw-test/gallery/bountyHunters.jpg",
      "/sw-test/gallery/myLittleVader.jpg",
      "/sw-test/gallery/snowTroopers.jpg",
    ])
  );
});
```

This code is inside the worker: /sw-test/sw.js

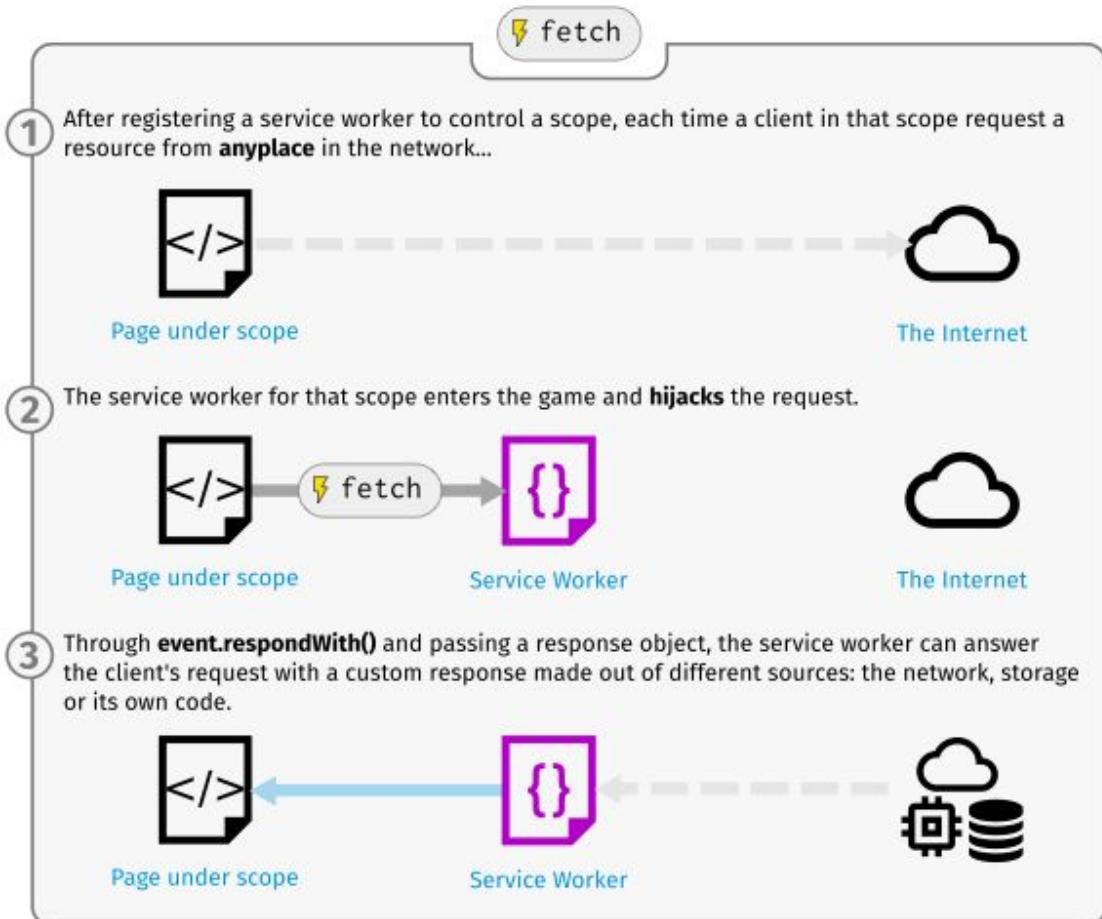


Serving data when offline

```
self.addEventListener('fetch', (event) => {  
  event.respondWith(  
    caches.match(event.request)  
  );  
});
```

This example listens for **fetch** and returns the cached data, but you can return anything you want in **event.respondWith!**

If data is not cached, it could make network request and update the cache.





Data storage in a PWA

- Web Storage (Local storage)
- IndexedDB
- Service Workers and cached resources (using CacheStorage API)



WebAssembly



- ◉ Binary code that is pre-compiled to WebAssembly (wasm) from other languages, incl.:
 - Emscripten (C, C++)
 - Rust
 - AssemblyScript (TypeScript)
 - TinyGo (Go)
- ◉ Ahead-of-time or Just-in-time compilation in browser
- ◉ 2019 W3C recommendation
- ◉ Can be a supporting technology for PWA, but designed for any high-performance web page



WASM applications

Support for languages and toolkits

Image / video editing.

Games (e.g. with heavy assets)

Peer-to-peer applications

Music applications

Image recognition

VR and augmented reality

CAD applications

Scientific visualization and simulation

Interactive educational software, and news articles

Platform simulation / emulation

Language interpreters and VMs

POSIX user-space environment

Developer tooling

Remote desktop

VPN

Encryption

Local web server

Fat client for enterprise applications (e.g. databases)



WebAssembly bytecode

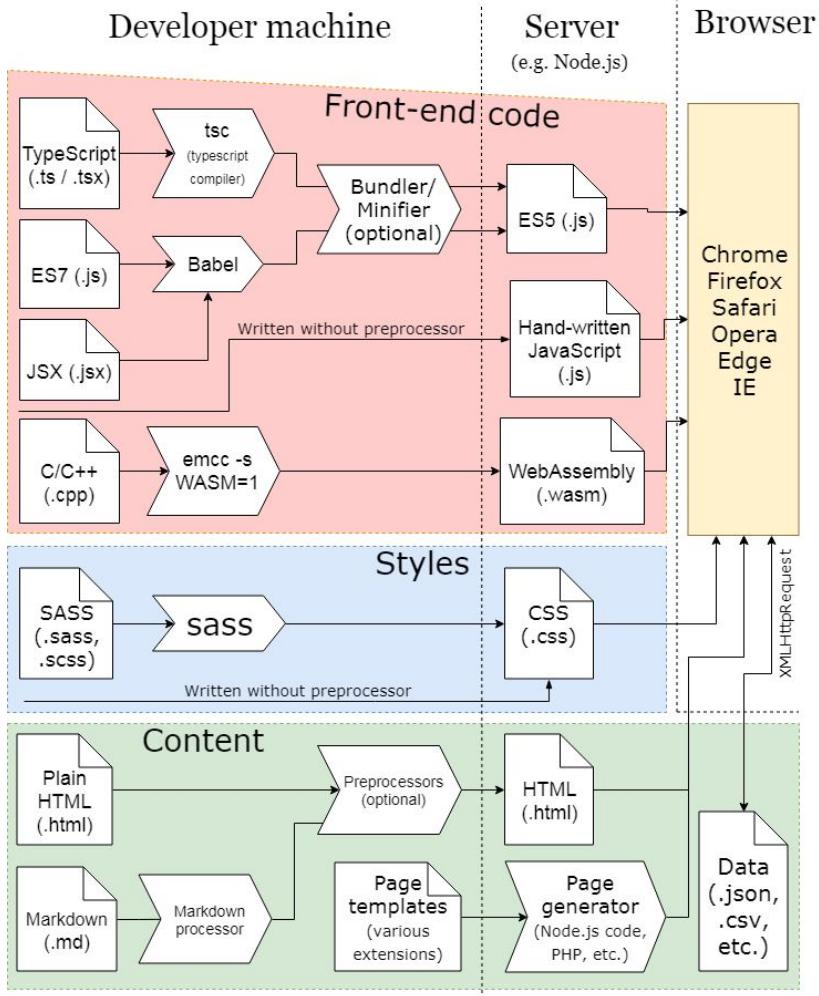
C source code	WebAssembly .wat text format	WebAssembly .wasm binary format
<pre>int factorial(int n) { if (n == 0) return 1; else return n * factorial(n-1); }</pre>	<pre>(func (param i64) (result i64) local.get 0 i64.eqz if (result i64) i64.const 1 else local.get 0 local.get 0 i64.const 1 i64.sub call 0 i64.mul end)</pre>	<pre>00 61 73 6D 01 00 00 00 01 00 01 60 01 73 01 73 06 03 00 01 00 02 0A 00 01 00 00 20 00 50 04 7E 42 01 05 20 00 20 00 42 01 7D 10 00 7E 0B 0B 15 17</pre>

More examples: <https://wasmbyexample.dev/home.en-us.html>



Web Technologies

- Client-side tech is now much more than HTML/JS/CSS



SENG 365 Week 12

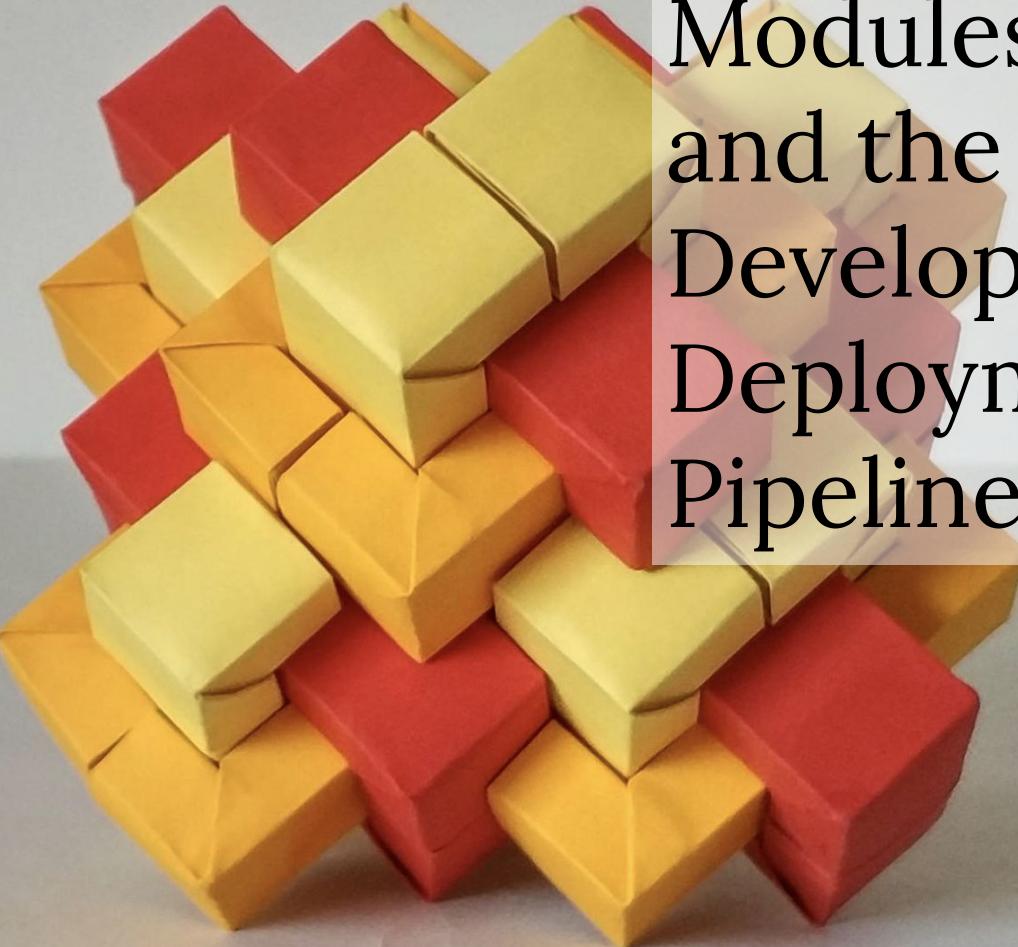
Front-end testing





Topics this week

- Modules and dev/deployment pipeline
- Automated testing
- Webdriver
- Client-side JavaScript testing
- Final Exam



Modules and the Development / Deployment Pipeline

Downloading Single Page Apps (SPA)

- A SPA usually consists mostly of JS and HTML templates. The CSS load is typically higher due to custom styling requirements
- Critically, the ‘preferred form’ of working with SPAs is not a good format for downloading:
 - Many small files,
 - Data that is not needed at runtime (ie, good function names),
 - Often the preferred coding language is not JavaScript (e.g. TypeScript)

Bundling

- Bundling is the process of taking the preferred form of Web code/content and putting it into the preferred form for a browser to consume
 - Analogous to compiling a C or Java program
- What does a browser/user prefer?
 - Fewer network requests (fewer files)
 - Smaller network requests
 - Parallel network requests
 - No network requests at all (caching)

Simple Bundling (fewer files)

- Concatenate all required *.js files to one big file
- Do the same for *.css files
- Tools like Gulp and Grunt follow this basic pattern:
 - Sweep up a bunch of files without knowing about file content
 - Perform some transform on them to generate (fewer) output files

Simple Bundling (smaller files)

- Browsers don't care about JS function names (much)
- We do care about data sent over the network though
- UglifyJS shortens var and function names

```
1  function globalFunctionName(input) {
2    function nestedFunction() {
3      input = input + 1;
4      console.log(input);
5    }
6    nestedFunction();
7  }
8  -----
9  $ uglifyjs foo.js --mangle --compress --source-map=map.out
10 function globalFunctionName(n){!function(){n+=1,console.log(n)}()}
11
```

UglifyJS (terser, etc)

- Code is name mangled, some optimizations are applied
 - Mangling can only be applied when we are **sure** that all instances of the name are renamed
- Result is difficult to read, but very easy to reverse engineer – so this offers no IP protection
 - There is no real way to protect IP that runs in a browser
 - Anything you send is public knowledge
- To aid debugging, **sourcemaps** can be created:
 - Browser loads mangled code + sourcemaps and the dev tools presents the code as it originally was
 - Learn more:
<https://github.com/ryanseddon/source-map/wiki/Source-maps%3A-languages,-tools-and-other-info>

Webpack (rollup/parcel)

- Webpack and other bundlers are the current best practice
- Webpack understands the input language (JavaScript/TypeScript)
- Configure Webpack with
 - Where to find input files
 - Where to start from (app entry point)
- Webpack will recursively follow imports
 - Import -> `import foo from "module_name"`
 - Export -> `export foo = 42;`
 - Only the files that are used get pulled into the final output
 - Unused (dead) code can be removed
- Final output will be something like `main.9789bf1740765e50e459.js`
 - E.g. ~10Mb of JS

Lazy Loading

- Break your SPA into logical chunks, eg
 - Login (likely the main chunk)
 - Settings
- Only download the chunks you need
- Typically loading is triggered on router changes
 - /login
 - /settings
- The framework and the bundler need to agree on how to split the import graph, usually done with bundler plugins.

Lazy Loading

- Webpack transforms first line at compile time

```
// Instead of a usual import  
import MyComponent from "~/components/MyComponent.js";  
  
// do this  
const MyComponent = () => import("~/components/MyComponent.js");
```

js

See <https://webpack.js.org/guides/lazy-loading/>



Automated testing

Web Applications

Brief overview to testing

- Many different types of tests
 - Unit tests, behavioural tests, acceptance tests, regression tests
- Many different kinds of system to test
 - Safety-critical, embedded systems, real-time systems etc
- Particular challenges for testing web applications:
 - Different browsers (FireFox, Chrome)
 - Different versions of browser (IE6 vs IE8)
 - Differences in versions of HTML, ECMAScript (JavaScript)
 - Differences in APIs e.g. how XHRs are handled by different browsers
 - Differences in the handling of the DOM, JavaScript etc.
 - Differences in libraries and versions

Advantages of automating testing

- Automation means you can offload testing to machines (rather than rely on humans)
- (More) frequent testing e.g. regression testing
- Quick and regular feedback to developers
- Virtually unlimited iterations of test case execution
- Support for Agile and extreme development methodologies
- Disciplined documentation of test cases
- Customized defect reporting
- Finding defects missed by manual testing
- Supports continuous integration and continuous deployment

When not to automate (and manually test instead)

- On the client side:
 - If the user interface is rapidly changing
 - e.g. HTML elements are changing
 - Will need to keep changing the tests to match the interface
- On the server side
 - If the API is rapidly changing...
 - Tight timescales
 - Don't have time to develop the tests



Automated testing of clients: WebDriver API

<https://www.w3.org/TR/webdriver/>



WebDriver API

- ◉ Intended to enable web authors to write tests that automate a user agent using a separate controlling process
 - May also be used to allow in-browser scripts to control a (possibly separate) browser
- ◉ Provides a set of interfaces to:
 - discover and manipulate DOM elements in web documents; and
 - to control the behavior of a user agent (a browser).
- ◉ Provides a platform-neutral and language-neutral wire protocol as a way for out-of-process programs to remotely instruct the behavior of web browsers.
- ◉ Forms part of the W3C Web Testing Activity



WebDriver API

- Intended to enable web authors to write tests that automate a user agent using a separate controlling process...
- Provides a set of interfaces to discover and manipulate DOM elements in web documents; and
- Provides a platform-neutral and language-neutral wire protocol as a way for out-of-process programs to remotely instruct the behavior of web browsers.

Questions

- Any implications relating to the DOM here?
- Any concerns relating to out-of-process programme controlling web browsers?



Selenium

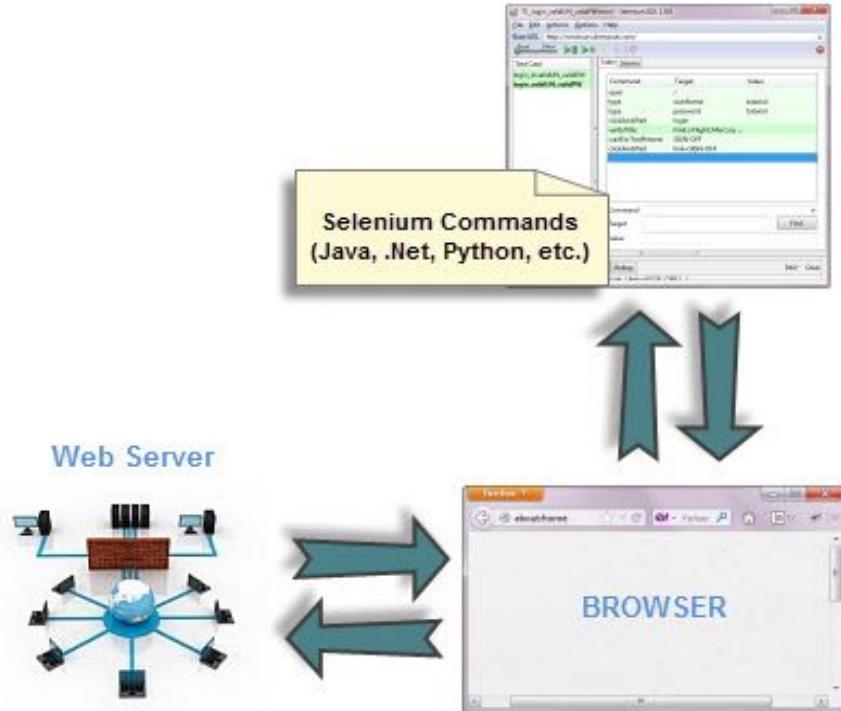
Selenium-WebDriver | Selenium 2.0

- Automates a browser
 - Use to automate testing
 - Use to automate routine web tasks e.g. some admin task
- Selenium WebDriver: drive the browser (automatically) the way a user would
 - Automate what you want the user to do
 - Automate what you want the user not to do
 - Automate unintentional behaviour, accidental behaviour, stupid behaviour, risky behaviour (e.g. security attacks?)

Selenium-WebDriver

- Developed to better support automation of **dynamic web pages**,
 - Dynamic pages: elements of a page may change without the page itself being reloaded. (What is a webpage?)
 - Single Page Applications generate dynamic web pages
- WebDriver relies on the browser's built-in (native) support for automation
 - You'll need an update-to-date browser
 - Harder to automatically test older browsers

Selenium Webdriver



Cross-browser testing

“When we say “JavaScript” we actually mean “JavaScript and the DOM”. Although the DOM is defined by the W3C each browser has its own quirks and differences in their implementation of the DOM and in how JavaScript interacts with it. HtmlUnit has an impressively complete implementation of the DOM and has good support for using JavaScript, but it is no different from any other browser: it has its own quirks and differences from both the W3C standard and the DOM implementations of the major browsers, despite its ability to mimic other browsers.”

http://www.seleniumhq.org/docs/03_webdriver.jsp#htmlunit-driver

Implication: automate your testing on **different** browsers



Example automated testing

```
const webdriver = require('selenium-webdriver'),
  By = webdriver.By,
  Key = webdriver.Key,
  until = webdriver.until;

const driver = new webdriver.Builder().forBrowser('firefox').build();

driver.get('http://google.co.nz')
  .then(_ =>
    driver.findElement(By.name('q')).sendKeys('university of canterbury',
      Key.RETURN))
  .then(_ => driver.wait(until.titleIs('university of canterbury - Google Search'),
    1000))
  .then(_ => driver.quit());
```



Example automated testing

```
const webdriver = require('selenium-webdriver'),  
  By = webdriver.By,  
  Key = webdriver.Key,  
  until = webdriver.until;  
  
const driver = new webdriver.Builder().forBrowser('firefox').build();  
  
driver.get('http://google.co.nz')  
  .then(_ =>  
    driver.findElement(By.name('q')).sendKeys('university of canterbury',  
      Key.RETURN))  
  .then(_ => driver.wait(until.titleIs('university of canterbury - Google Search'),  
    1000))  
  .then(_ => driver.quit());
```

Summary of what you can do

- Fetch a page:
- Locate a UI (DOM) element
- Get text values
- User input
- Move between windows and frames
- Popup dialogs
- Navigation and history (may be harder in SPA)
- Cookies
- Drag and drop
- Check out the details:

<https://www.selenium.dev/documentation/en/>

Example of what you can do

Fetch a page:

```
driver.get('http://www.google.com');
```

Locate a UI (DOM) element By ID:

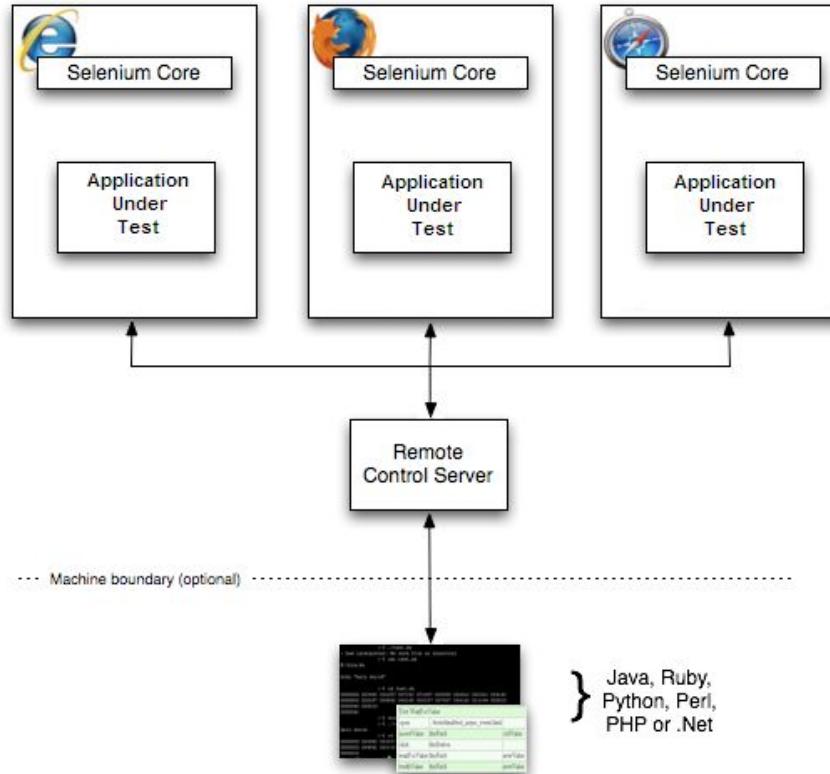
- `let element =
 driver.findElement(By.id('someID'));`
- Compare with JavaScript `getElementById()`;

WebDriver vs the Selenium-Server

- You can use WebDriver without Selenium Server
 - Browser and tests will all run on the same machine
- However, there are reasons to use the Selenium-Server, e.g.
- You are using Selenium-Grid to distribute your tests over multiple machines or virtual machines (VMs).
- You want to connect to a remote machine that has a particular browser version that is not on your current machine.
- You are not using the Java bindings (i.e. Python, C#, or Ruby) and would like to use HtmlUnit Driver
 - HtmlUnit is a "GUI-Less browser for Java programs"

Selenium-Server

Windows, Linux, or Mac (as appropriate)...



Test Utilities for JavaScript frameworks

- Vue, React, Angular, etc. all have **test utilities**
- **JavaScript testing**, *not browser testing* (Selenium-Webdriver)
- Allows you to run **unit and integration tests**
- JavaScript **test runner**:
 - Agnostic about the framework
 - Jest, <https://jestjs.io>
 - JavaScript test runner that lets you run tests of DOM interaction
- **React Testing Library**
 - <https://testing-library.com/docs/react-testing-library/intro>

React Testing Library

```
test('loads and displays greeting', async () => {
  render(<Fetch url="/greeting" />

  fireEvent.click(screen.getByText('Load Greeting'))

  await waitFor(() => screen.getByRole('heading'))

  expect(screen.getByRole('heading')).toHaveTextContent('hello there')
  expect(screen.getByRole('button')).toBeDisabled()
})
```



Final Exam

When: June 13th, 9:30 am

Time: 2 hours

Format: Mix of short answer (incl. code), multiple choice

Where: In CSSE Computer Labs.

Exam password will be shared on Learn before exam starts.

Closed book