SENG 365 Week 10

SPA Communication with Server

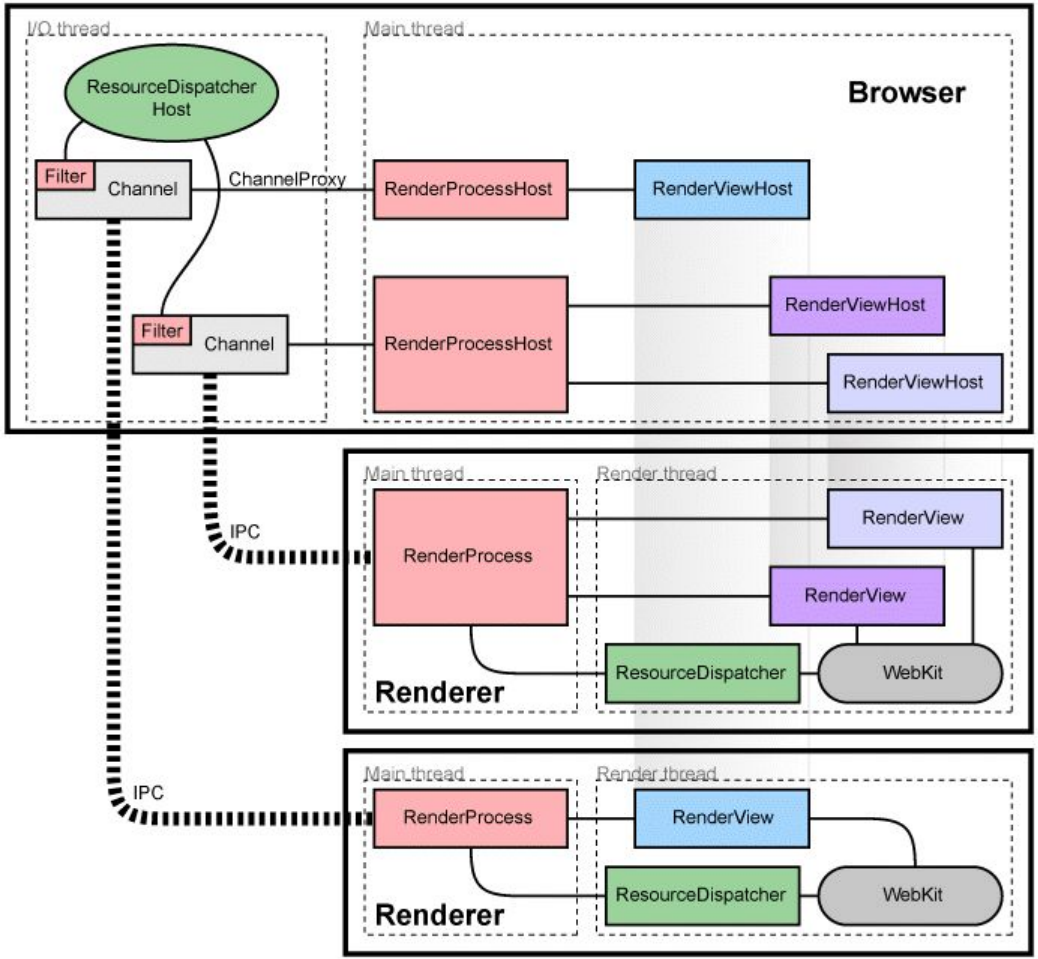
This week

- AJAX
- XHR
- CORS
- Web sockets

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?



https://www.chromium.org/developers/design-documents/multi-process-architecture

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

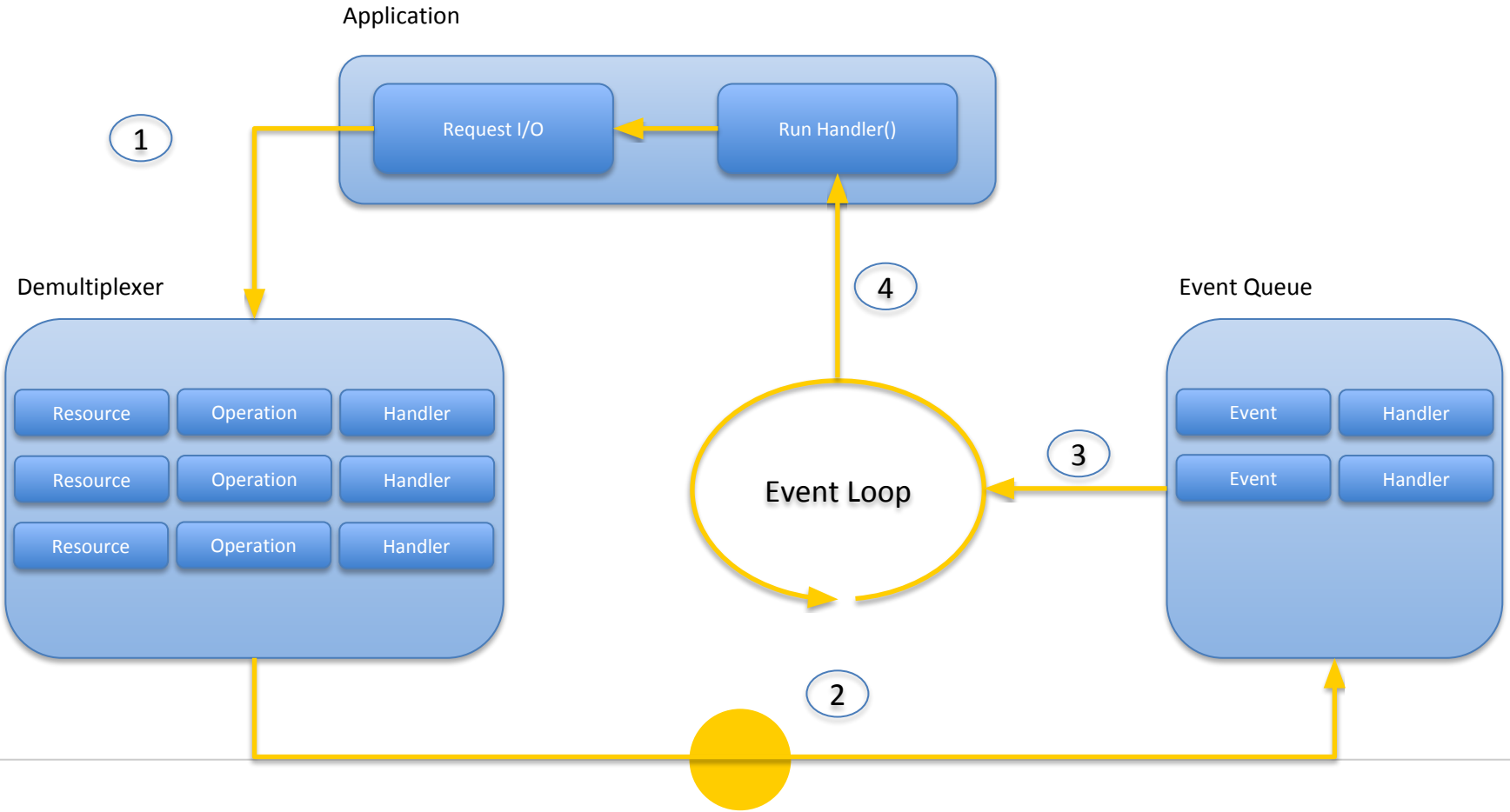
- Native Javascript API introduced in 2017
- Implemented in most modern browsers now
  - https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API#browser_compatibility

```
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?



Node.js Design Patterns, 2nd edition, Casciaro and Mammino, 2016, Packt Publishing

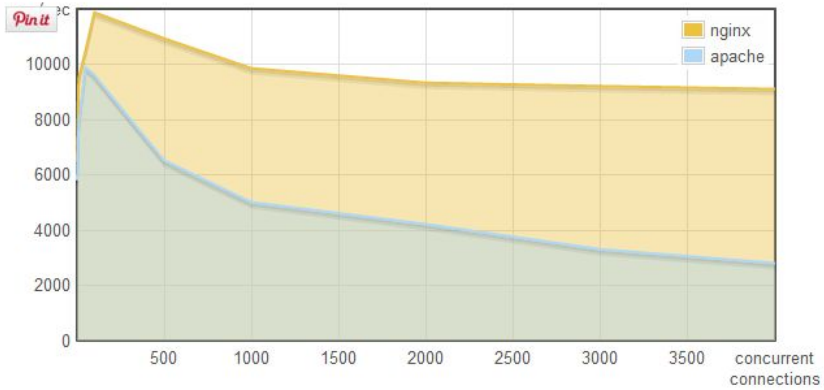
A little holiday present: 10,000 reqs/sec with Nginx!

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."

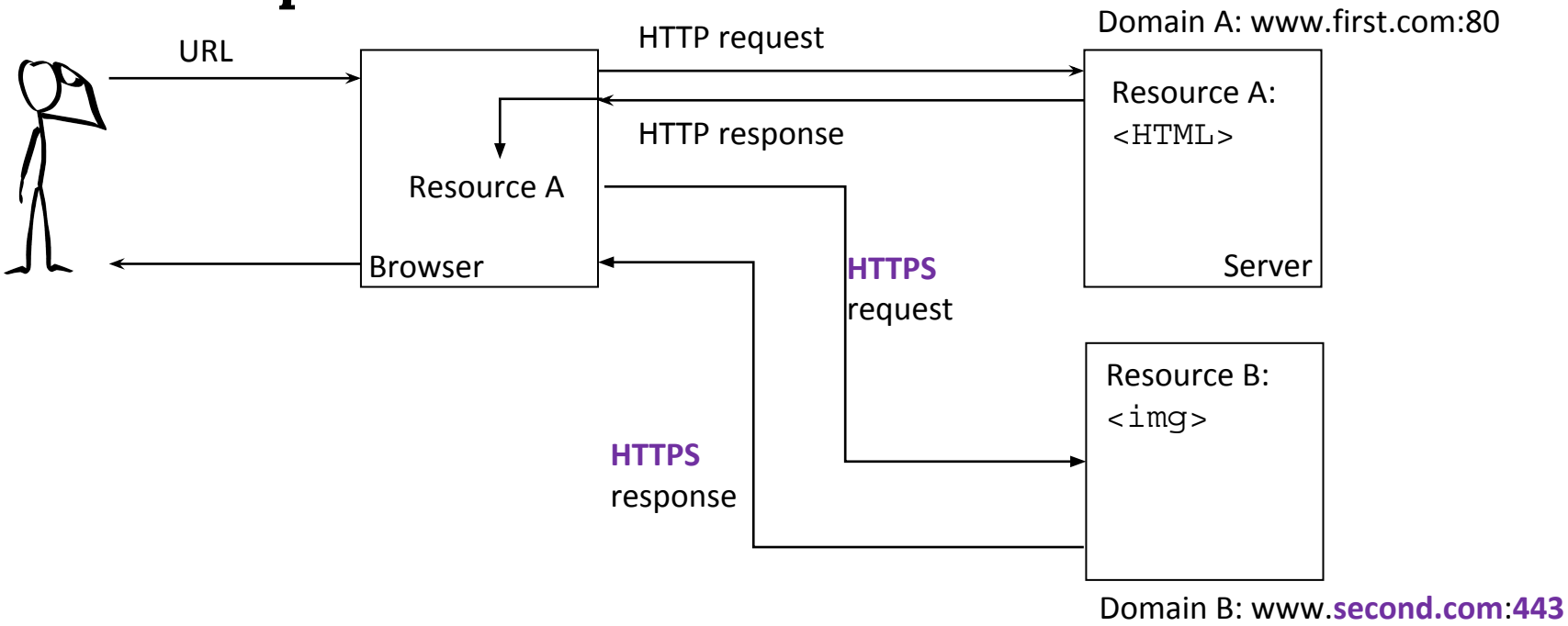
– Concurrency is not Parallelism, the GoLang blog

CORS

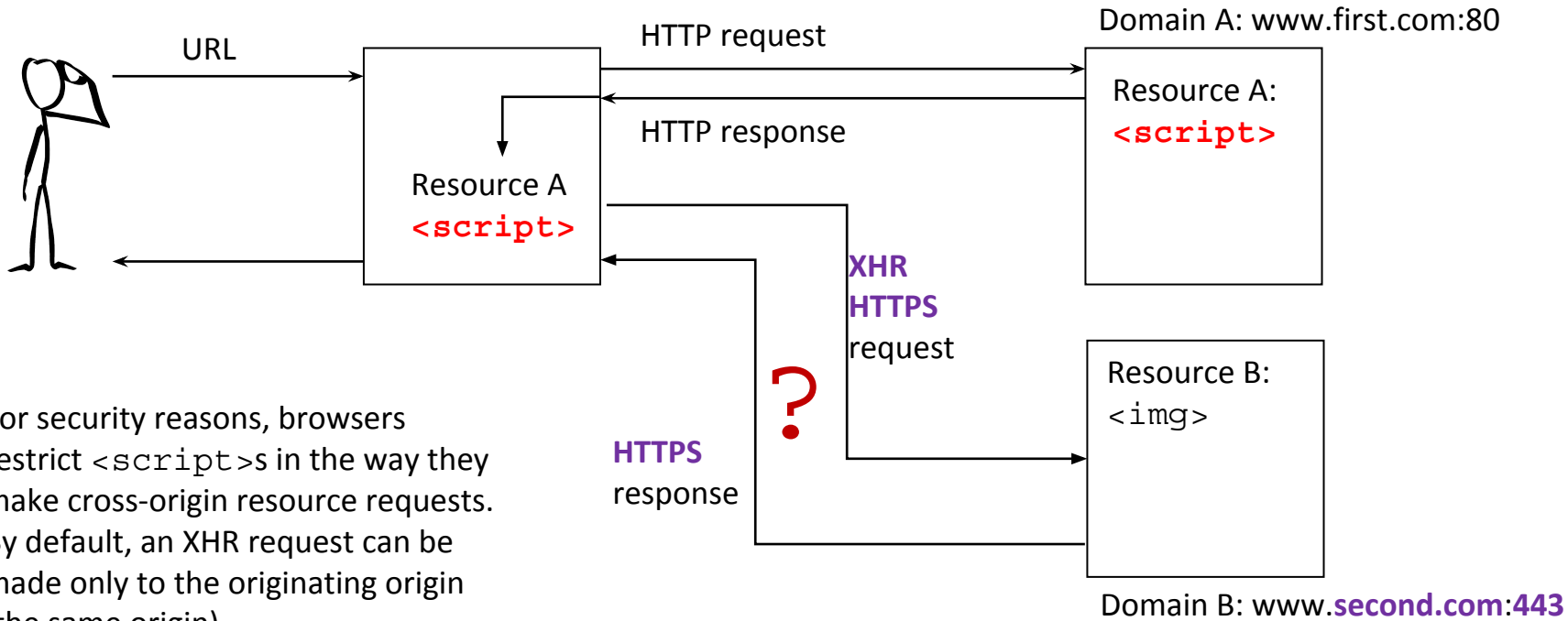
Cross Origin Resource Sharing

```
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

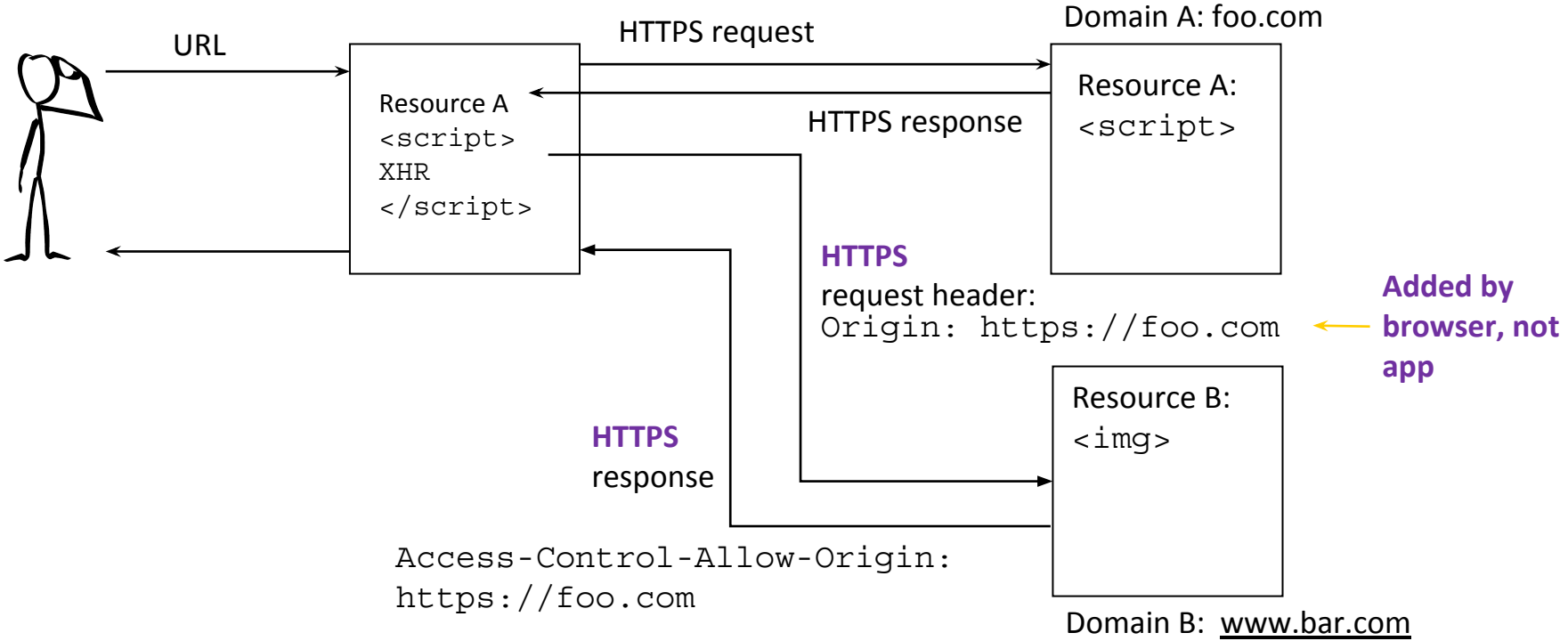
Some of the headers returned by the server

`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 (remember what defines and 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: *`)

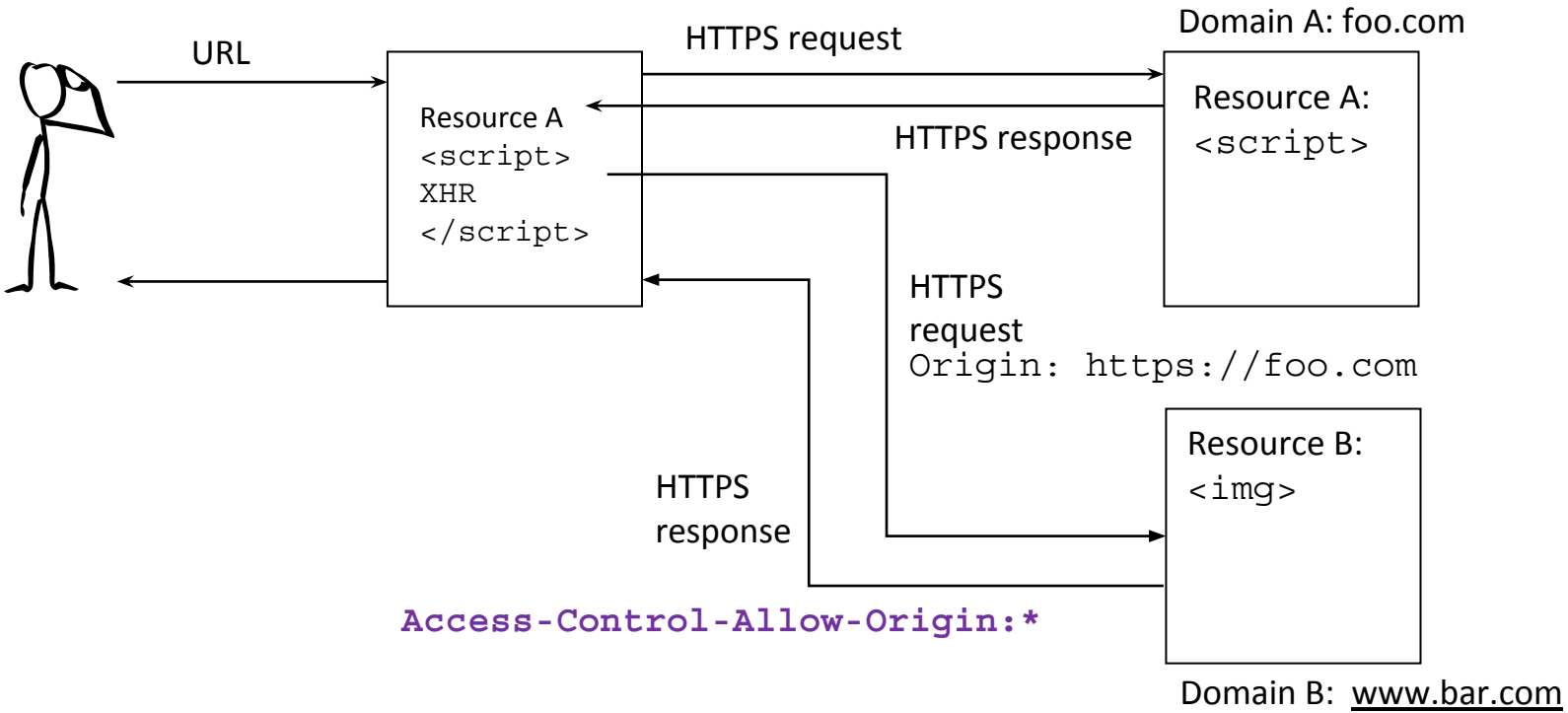
Adding CORS headers to server response in node.js

```
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();
});
```

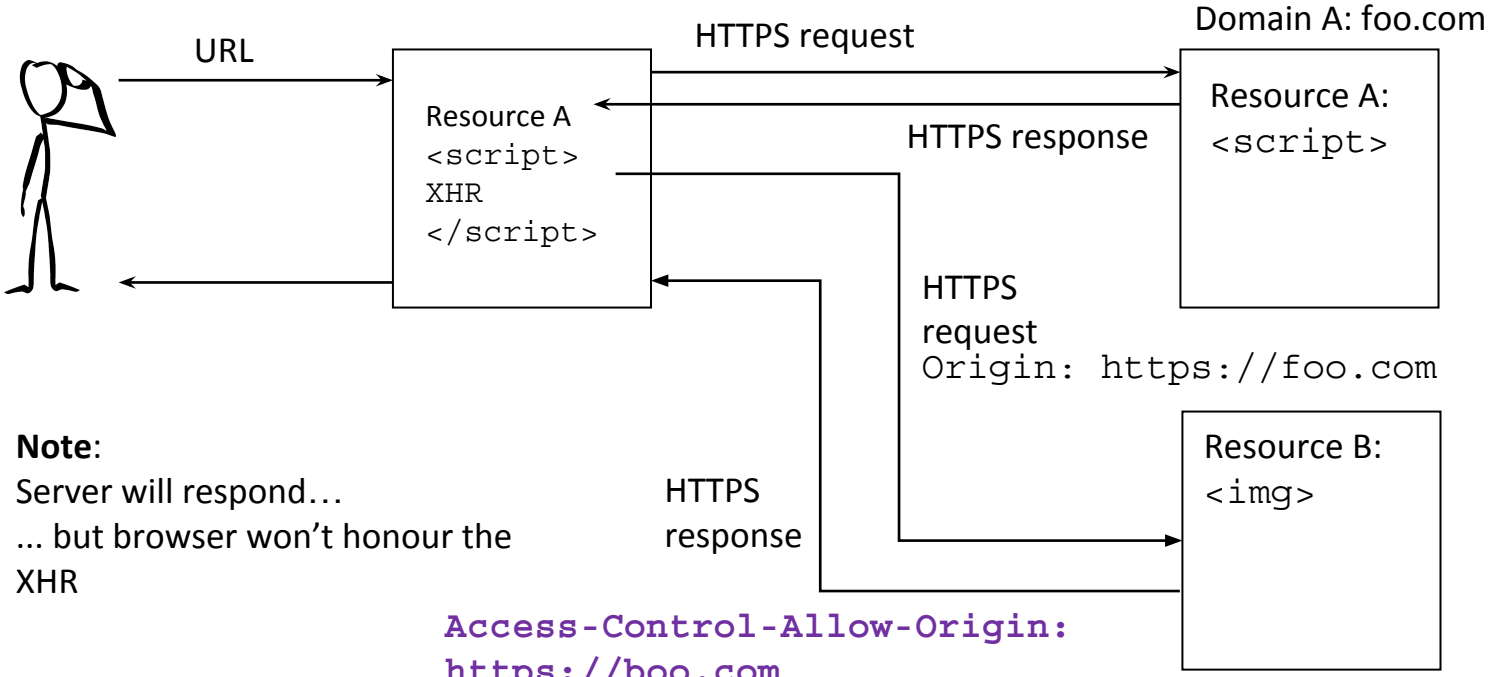
Example 1: accept request from foo.com only



Example 2: accept request from anywhere (*)

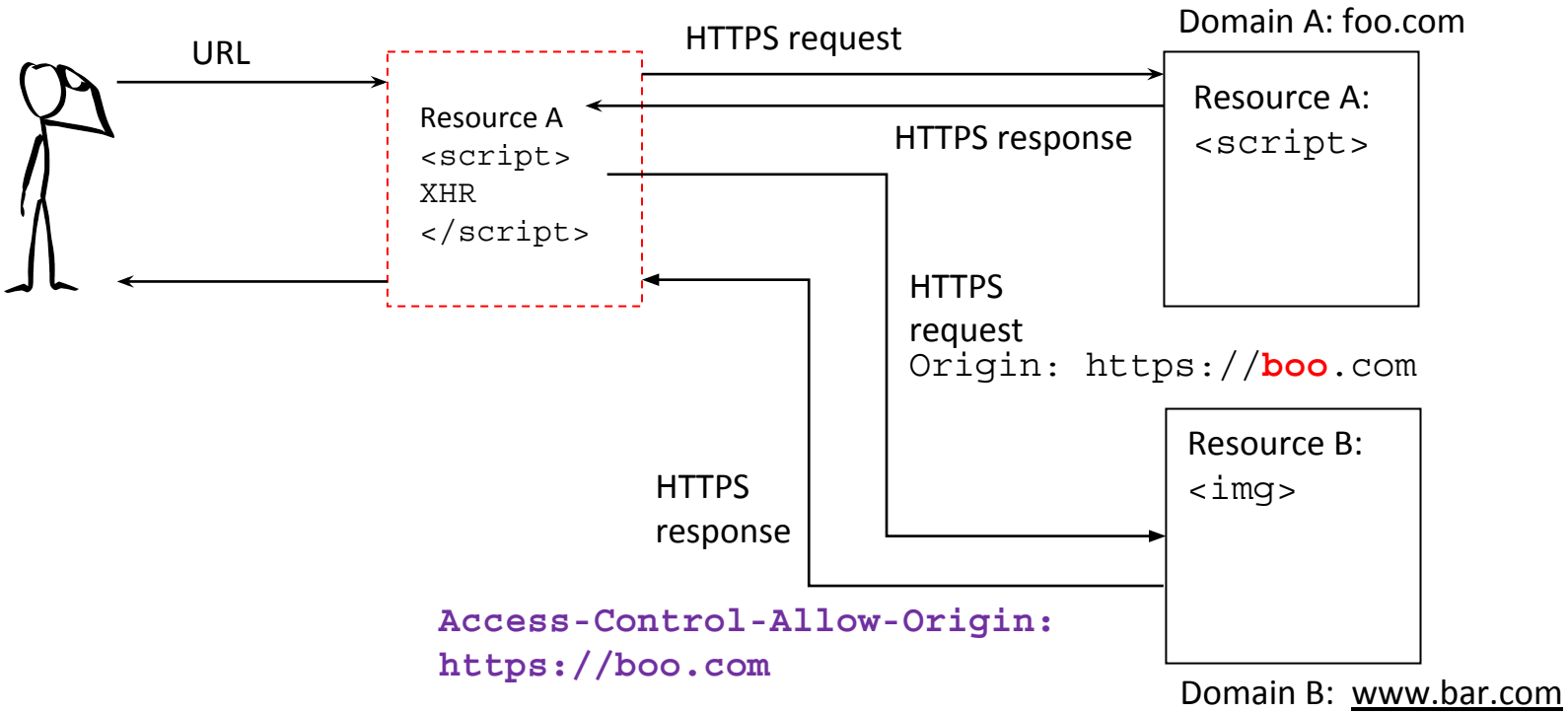


Example 3: reject request



Note:
Server will respond…
… but browser won't honour the XHR

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

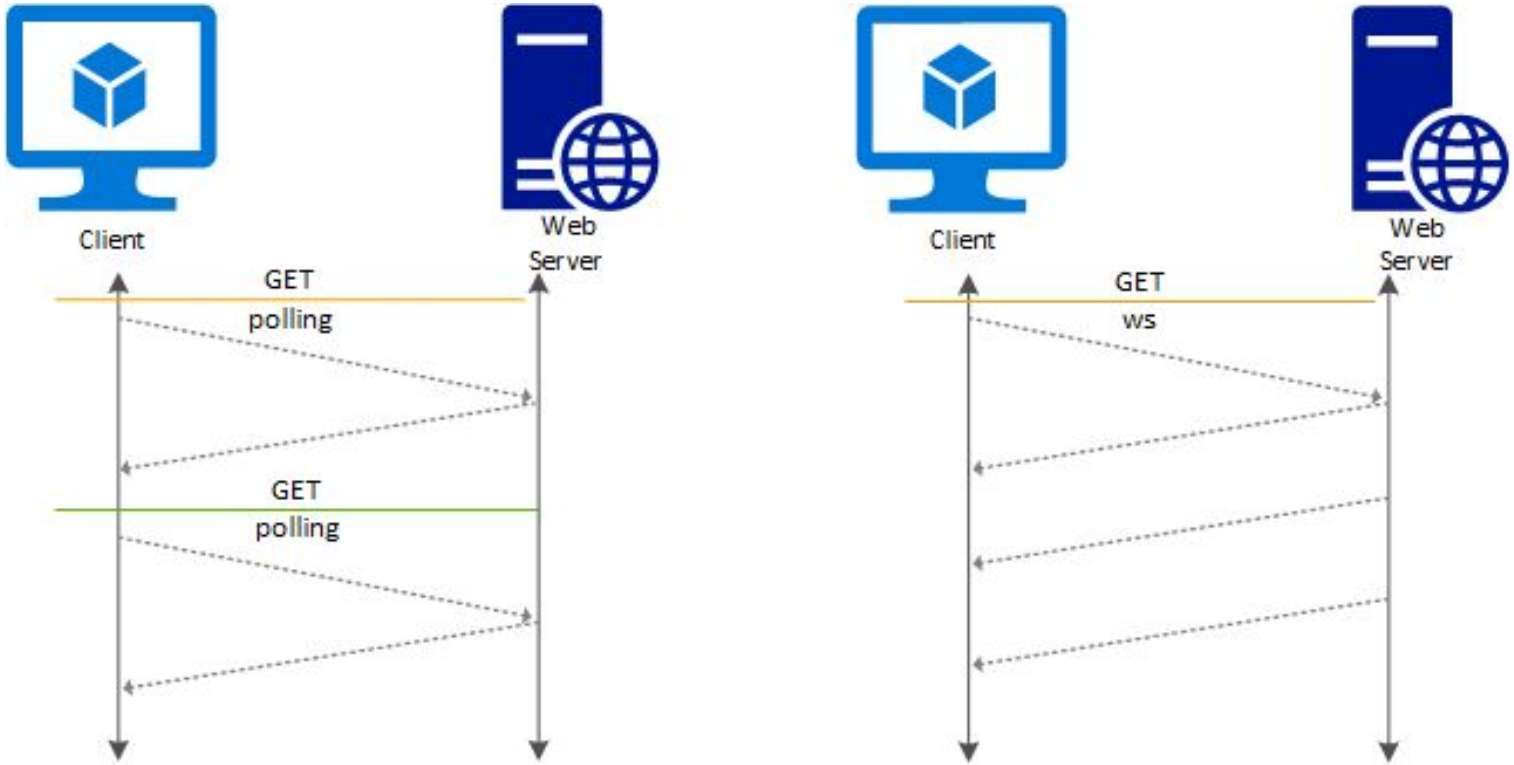
HTTP, AJAX

- 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: 5TJpHv9RoAl7w8ytsXcWxTOZ9Q==
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);
});
```