JSON

CSCI 1720

JS Objects

JS Objects

In order to understand JavaScript Object Notation, it's best to have at least a basic understanding of JS objects

JSON is a subset of JS objects used to transfer data in a structured, standardized format

So, what about objects?

JS Objects

An ‘object’ is one of JavaScript’s supported data types

It is an associative array of key:value pairs

Keys are developer-defined values that describe parts of an object

Values represent information associated with the object that may be unique to a given instantiation of that object

JS Objects

For example:

- We have an object named ‘Cars’
- One of the properties (key:value pairs) is ‘make’
- We have two ‘Cars’ objects
- The ‘make’ for car1 might be ‘Ford’; for car2 ‘Porsche’
- Each of these objects are independent
- Each contains the same ‘kind’ of data, but with different values for that data

https://csci1720.net/lecture/JSON-examples/json1.html

JS Objects

For example:

- Each contains the same ‘kind’ of data, but with different values for that data
- We can change the ‘make’ of car1 (and probably should) to ‘Chevrolet’
- Action is specific to car1; car2 is not affected
- We could delete car2, but car1 would still persist

JS Objects

Data Types

Values assigned to keys can have several data types

- Primitives
- Arrays
- Objects
- Arrays of objects
- Functions

JS Objects

Data Types

Primitives

```
{
    firstName:      'Joe',
    lastName:       'Dokes',
    age:            '34',
    vet:            true
}
```

JS Objects

Data Types

Arrays

```
var person = {
    firstName:      'Joe',
    lastName:       'Dokes',
    pets:           [
                        'Fluffy',
                        'Spike',
                        'Renaldo',
                        'Shirl'
                    ]
}
```

JS Objects

Data Types

Objects

```
{
    person: {
                firstName:  'Joe',
                lastName:   'Dokes',
                age:        '34'
            }
}
```

JS Objects

Data Types

Array of Objects

```
{
    [
        person:  {
                    firstName:  'Joe',
                    lastName:   'Dokes',
                    age:        '34'
                 },
        address: {
                    number:     201,
                    street:     'South Sycamore St.',
                    city:       'Elizabethton',
                    state:      'TN',
                    zip:        '37643'
                 }
    ]
}
```

JS Objects

Data Types

Functions

In JS, a function can be assigned to a variable, in general

When it is used in a JS object, it becomes the equivalent to a Java object method

JS Objects

Data Types

Functions

Functions (we'll just call them methods from now on) serve to access and manipulate object data

We could, for example, create a method that will return a "person's" full name, concatenated from the 'firstName' and 'lastName' values

https://csci1720.net/lecture/JSON-examples/json4.html

JS Objects

Data Types

Functions

```
var person = {
    firstName:      'Joe',
    lastName:       'Dokes',
    getName:        function() {
                        return this.firstName + ' '
                        + this.lastName;
                    }
}

console.log(person.firstName);
console.log(person.getName());
```

JS Objects

Data Types

Functions

```
> var person = {
      firstName:        'Joe',
      lastName:         'Dokes',
      getName:          function() {
                            return this.firstName + ' '
                            + this.lastName;
                        }
  }
  console.log(typeof(person));
  console.log(person.firstName);
  console.log(person.getName());
```

```
object
Joe
Joe Dokes
```

JS Objects

That's the basics of JS objects

There are a number of built-in methods associated with all objects

As we've seen, we can define custom methods as well

Other examples of object use:

https://csci1720.net/lecture/JSON-examples/json8.html
https://csci1720.net/lecture/JSON-examples/json9.html

So how does this relate to JSON?

Let's look first at an older technology

Old School: XML

eXtensible Markup Language

XML is an older way of structuring data for sharing

MUCH more verbose than JSON

Makes use of ‘custom’ HTML tags and elements

Compliant with HTML

Platform independent

Still widely used (e.g., RSS feeds)

XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<root>
   <person1>
      <age>35</age>
      <firstName>Joe</firstName>
      <lastName>Dokes</lastName>
      <midName>Alan</midName>
   </person1>
   <person2>
      <age>24</age>
      <firstName>Sally</firstName>
      <lastName>Linkhouse</lastName>
      <midName>Mae</midName>
   </person2>
   <person3>
      <age>48</age>
      <firstName>Bobby</firstName>
      <lastName>Sturmgard</lastName>
      <midName>Jessie</midName>
   </person3>
</root>
```

https://csci1720.net/lecture/JSON-examples/json7.html

eXtensible Markup Language

If you looked at the last slide and immediately thought, “Ugh.”, welcome to the club

It is structured and can be parsed as needed

But … Wow

JSON, as we’ll see, is a lot easier

JSON is replacing XML as the language of choice for data structuring and sharing

JSON

JavaScript Object Notation

JSON stands for JavaScript Object Notation

It was designed for human-readable data interchange

It has been extended from the JavaScript scripting language

The filename extension is .json

JavaScript Object Notation

It is used with JavaScript based applications that include browser extensions and websites

JSON format is used for serializing and transmitting structured data over a network connection

It is primarily used to transmit data between a server and web applications

Web services and APIs use JSON format to provide public data

It can be used with modern programming languages, not just JS

JavaScript Object Notation

Essentially, JSON is a JS object without methods

The only difference, otherwise, is that keys are enclosed with quotes

Unlike many other times we deal with strings, double quotes are required for well-formed JSON

Values that are numeric and/or Booleans do not have to be quoted

JSON does not (with one obscure exception) allow comments

JavaScript Object Notation

Syntax

- Considered a subset of JavaScript syntax
- Data is represented in name/value pairs
- Braces ({ }) hold objects
- Each name is followed by : (colon), the name/value pairs are separated by , (comma)
- Square brackets hold arrays and objects within the array are separated by , (commas)

JavaScript Object Notation

JSON is easy to read and write

Lightweight text-based (string) interchange format

Language/platform independent

https://csci1720.net/lecture/JSON-examples/json5.html

Example

Note that 'age' is defined as a string instead of an integer

All JSON data is string-based

It's easy enough to convert strings to integers or floats as needed

```
var age = parseInt(person.age);
```

```
{
    person1: {
        "firstName":  "Joe",
        "midName":    "Alan",
        "lastName":   "Dokes",
        "age":        "35"
    },
    person2: {
        "firstName":  "Sally",
        "midName":    "Mae",
        "lastName":   "Linkhouse",
        "age":        "24"
    },
    person3: {
        "firstName":  "Bobby",
        "midName":    "Jessie",
        "lastName":   "Sturmgard",
        "age":        "48"
    }
}
```

JavaScript Object Notation

Data Types	
Number	Double-precision floating-point format in JavaScript
String	Quoted Unicode with backslash escaping
Boolean	true or false
Array	Ordered sequence of values
Value	String, number, Boolean, null
Object	Unordered collection of key:value pairs

JavaScript Object Notation

stringify & parse

Useful methods with JSON -

- JSON.stringify() converts a JS object into JSON format
- JSON.parse() converts well-formed text strings into an array of JSON objects

JavaScript Object Notation

```
var log = { firstName: "Joe",
            lastName: "Dokes",
            age: 34
          }
var log1 = JSON.stringify(log);
console.log('JSON format ', log1);
console.log('Back to object (log) ', JSON.parse(log1));
```

JavaScript Object Notation

```
> var log = { firstName: "Joe",
              lastName: "Dokes",
              age: 34
            }
  var log1 = JSON.stringify(log);
  console.log('JSON format ', log1);
  console.log('Back to object (log) ', JSON.parse(log1));
```

```
JSON format  {"firstName":"Joe","lastName":"Dokes","age":34}
Back to object (log)  ▶ {firstName: "Joe", lastName: "Dokes", age: 34}
```

JSON Conclusion

Much of our profession involves efficiently handling data

‘data’ can be in any format

JSON provides a standardized way in which we can structure and share data

‘standardized’ means that, as long as you conform to the established standard, anyone can make use of a given data set

You’ll see JSON in other classes at ETSU

Asynchronous JS

Synchronous JS

In order to understand asynchronous JS, let's first look at what JS, by default, is

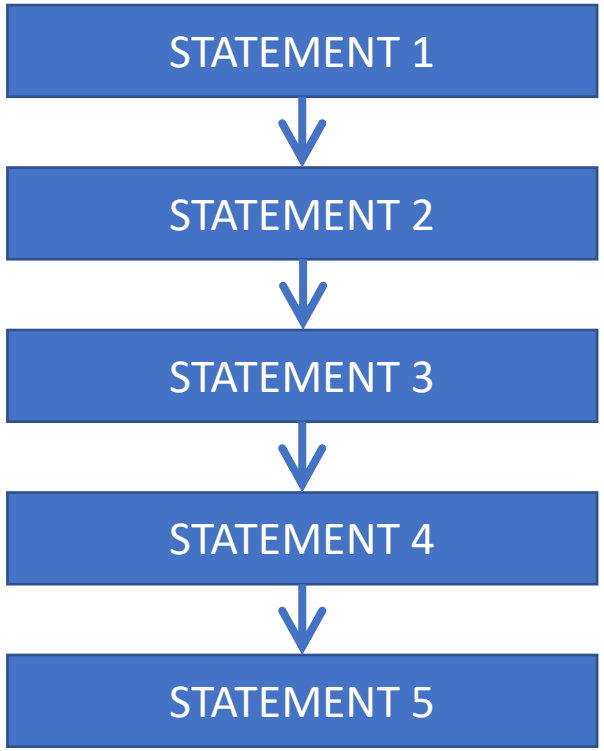
JS is a synchronous (or single-threaded) environment

thread === 'ordered sequence of statements'

thus,

Runs one (1) statement at a time

Synchronous JS



Synchronous JS

In the RW, we often need to execute functions that take a little time to complete

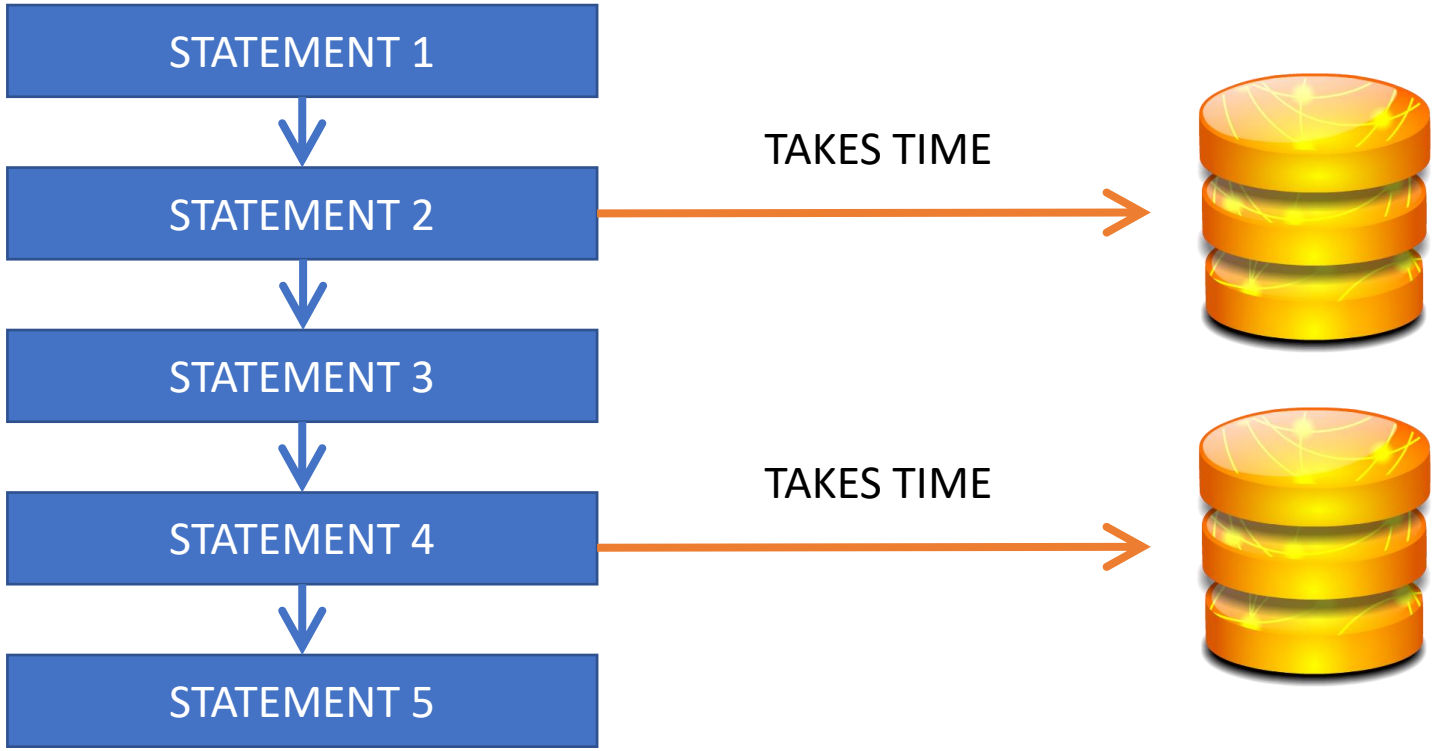
- These are referred to as ‘blocking statements’
- JS doesn’t like to wait

When JS encounters such a function, again by default, it will execute it and charge on to the next statement without waiting to see if it finished

This can lead to some wonky behavior/results

Synchronous JS

Synchronous JS



Asynchronous JS

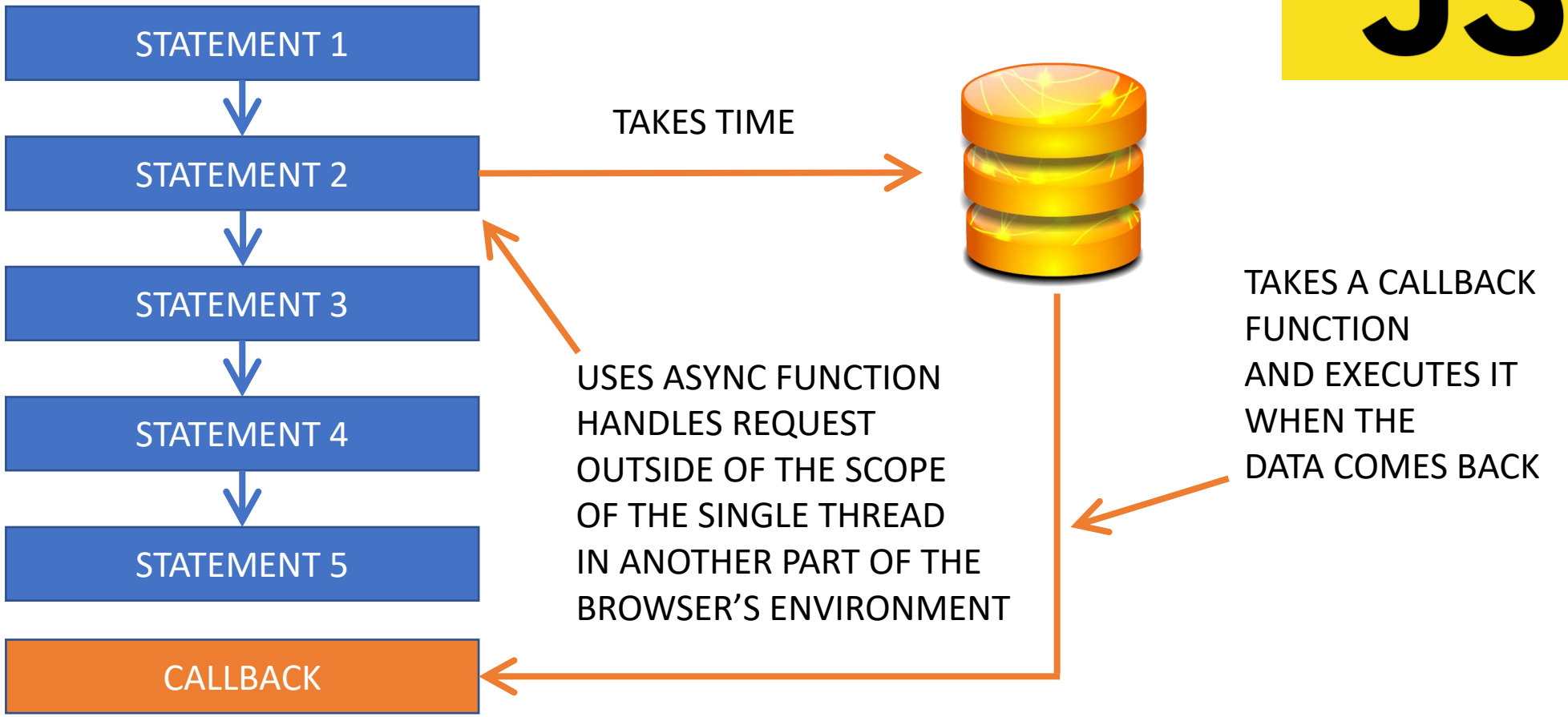
JS

Start something now & finish it later

Network requests:

- Database
- API

Synchronous JS



Synchronous JS

All Videos: https://csci1720.net/lecture/video/lecture13-json/index.php

https://csci1720.net/lecture/video/lecture13-json/003.mp4

Asynchronous JS

https://csci1720.net/lecture/video/lecture13-json/004.mp4

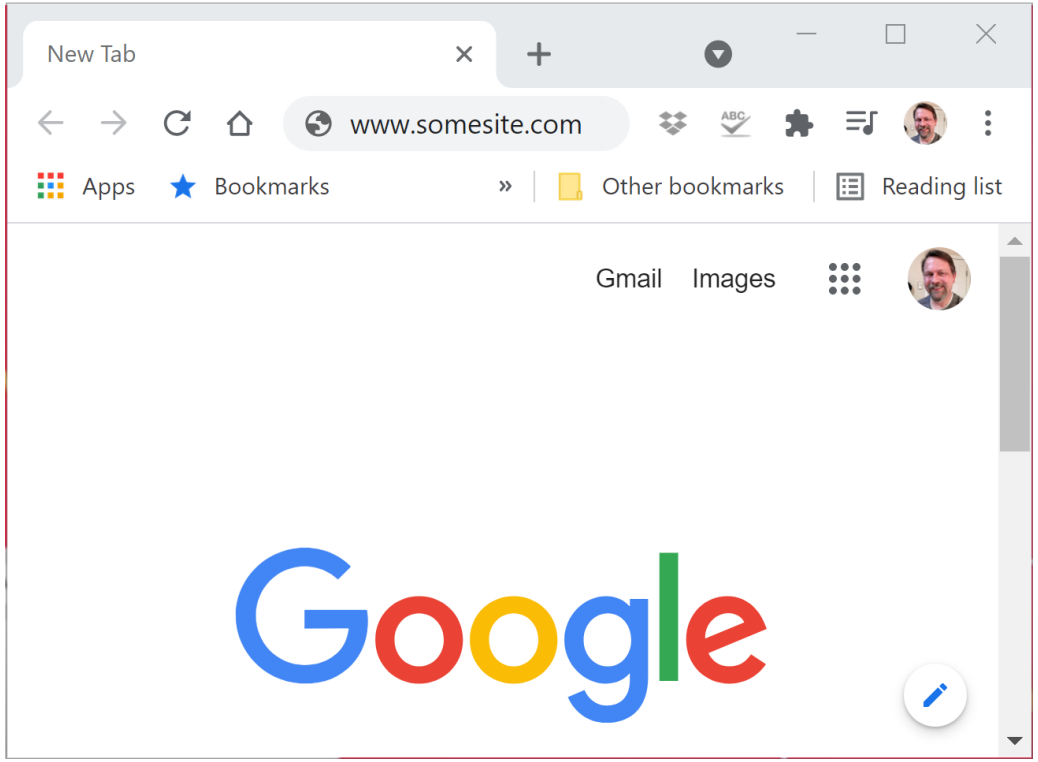
HTTP Requests

Make HTTP requests to get data from another server

We make these requests to API endpoints

URLs that an API or server exposes to us so that we can obtain the data

API Endpoints



HTTP REQUEST

HTTP RESPONSE

XMLHttpRequest

In this example, we're using an Application Programming Interface (API), typicode.com

We'll make a call to typicode, which will return dummy data in JSON format

Like a database request, an API call takes a little time to complete

https://csci1720.net/lecture/video/lecture13-json/006.mp4

XMLHttpRequest

Data is typically returned from an API in JSON format

So, to demonstrate asynchronous requests, we're going to use an API endpoint at https://jsonplaceholder.typicode.com/todos

To make the request, we have to set up an XMLHttpRequest object, open it, and send it

XMLHttpRequest

https://csci1720.net/lecture/video/lecture13-json/007.mp4

XMLHttpRequest

So, at this point, we've made and sent the request and received the response

But we still don't know when the request is complete or how to access that data

We need a little more code

XMLHttpRequest

We can track the progress of the request using an event listener

Specifically,

`readystatechange`

There are four states logged by the readystatechange event

XMLHttpRequest

https://csci1720.net/lecture/video/lecture13-json/008.mp4

XMLHttpRequest

Value	State	Description
0	UNSENT	Client has been created. `open()` not called yet.
1	OPENED	`open()` has been called.
2	HEADERS_RECEIVED	`send()` has been called, and headers and status are available.
3	LOADING	Downloading; `responseText` holds partial data.
4	DONE	The operation is complete.

XM

The
calle

rty

JS

localhost/csci1720.net/temp/asy

localhost/csci1720.net/temp/async_js/

Apps Bookmarks Alarm Facebook ETSU Google Shop Learn » Other bookmarks Reading list

Async JS

Making an HTTP request

Elements Console Sources Network »

top Filter Default levels

```
XMLHttpRequest {onreadystatechange: null, readyState: 4, timeout:
0, withCredentials: false, upload: XMLHttpRequestUpload, …}
  onabort: null
  onerror: null
  onload: null
  onloadend: null
  onloadstart: null
  onprogress: null
  onreadystatechange: null
  ontimeout: null
  readyState: 4
  response: "[↵  {↵    "userId": 1,↵    "id": 1,↵    "title": "…
  responseText: "[↵  {↵    "userId": 1,↵    "id": 1,↵    "title": …
  responseType: ""
  responseURL: "https://jsonplaceholder.typicode.com/todos"
  responseXML: null
  status: 200
  statusText: ""
  timeout: 0
  upload: XMLHttpRequestUpload {onloadstart: null, onprogress: nul…
  withCredentials: false
  __proto__: XMLHttpRequest
4
>
```

XMLHttpRequest

So we want to add a check of the readyState

```
if(request.readyState === 4) {

}
```

XMLHttpRequest

https://csci1720.net/lecture/video/lecture13-json/009.mp4

XMLHttpRequest

```
if(request.readyState === 4) {

}
```

isn't quite enough

We need to also verify that the response was successful

XMLHttpRequest

```
if(request.readyState === 4 &&
   request.status == 200) {

} else if(request.readyState === 4) {
   console.log('Could not fetch data');
}
```

So if the request finishes with an error, we'll log an error message to the console

XMLHttpRequest

https://csci1720.net/lecture/video/lecture13-json/010.mp4

THIS ACTUALLY SHOULD BE 200, NOT '200'. OOPS

XMLHttpRequest

Now we're making the request, waiting for the response, and making sure that the response is error-free

Now we need to add a callback function that'll do something with the data once it has been received

First, let's declare a callback function and put all the code we've got so far into it

XMLHttpRequest

```
const getTodos = (callback) => {
    // declare request object
    const request = new XMLHttpRequest();

    // add readystatechange eventlistener
    request.addEventListener('readystatechange', () => {
        if(request.readyState === 4 && request.status === 200) {
            console.log(request.responseText);
        } else if (request.readyState === 4) {
            console.log('Could not fetch data');
        }
    });

    // open the request; needs type and endpoint
    request.open('GET', 'https://jsonplaceholder.typicode.com/todos');

    // send the request
    request.send();
}
```

XMLHttpRequest

Now we can call the function with a separate function

```
getTodos();
```

But we need to do a little more to tie it in to the request

XMLHttpRequest

Turn it into a callback

```
getTodos(() => {

});
```

XMLHttpRequest

Add a call to the callback to our code

We'll call it 'callback,' but the name could be anything

XMLHttpRequest

```
const getTodos = (callback) => {
    // declare request object
    const request = new XMLHttpRequest();

    // add readystatechange eventlistener
    request.addEventListener('readystatechange', () => {
        if(request.readyState === 4 && request.status === 200) {
            callback(undefined, request.responseText);
        } else if (request.readyState === 4) {
            callback('Could not fetch data', undefined);
        }
    });

    // open the request; needs type and endpoint
    request.open('GET', 'https://jsonplaceholder.typicode.com/todos');

    // send the request
    request.send();
}
```

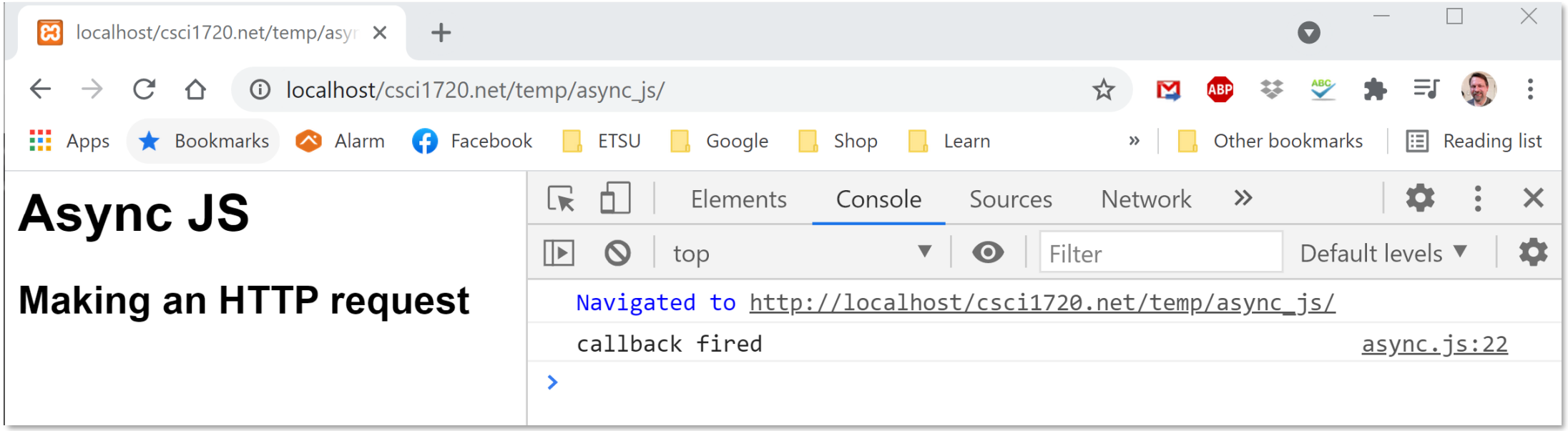
XMLHttpRequest

Add some test code

```
getTodos((err, data) => {
    console.log('callback fired');
});
```

XMLHttpRequest

Test the code



XMLHttpRequest

This is fine, as far as it goes

But notice that we get the same feedback whether the call was a success or not

We need to add a couple of parameters to be able to distinguish between the two

XMLHttpRequest

```
getTodos((err, data) => {
    console.log('callback fired');
});
```

So we're going to pass two parameters -> err (error) and data (data)

By convention, we call them 'err' and 'data'

Now, we can modify the code above to differentiate between success and failure

XMLHttpRequest

```
// add readystatechange eventlistener
request.addEventListener('readystatechange', () => {
    if(request.readyState === 4 && request.status === '200') {
        callback(undefined, request.responseText);
    } else if (request.readyState === 4) {
        callback('Could not fetch data', undefined);
    }
});
```

If the call is successful, the 'err' parameter will be undefined and we can pass the data back

If it is not successful, we can pass an error message back while the data parameter is undefined

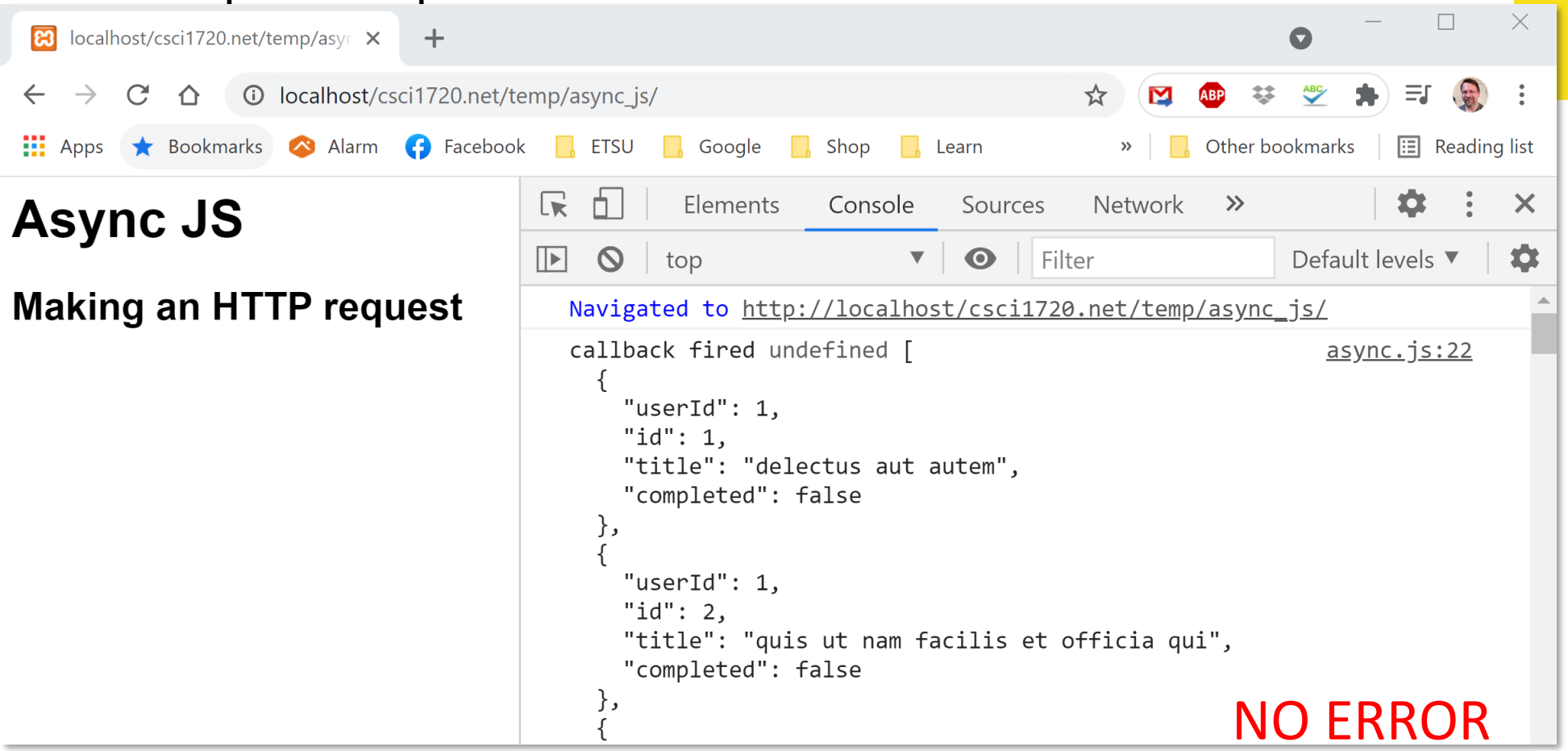
XMLHttpRequest

Now, we can modify the callback to handle either condition

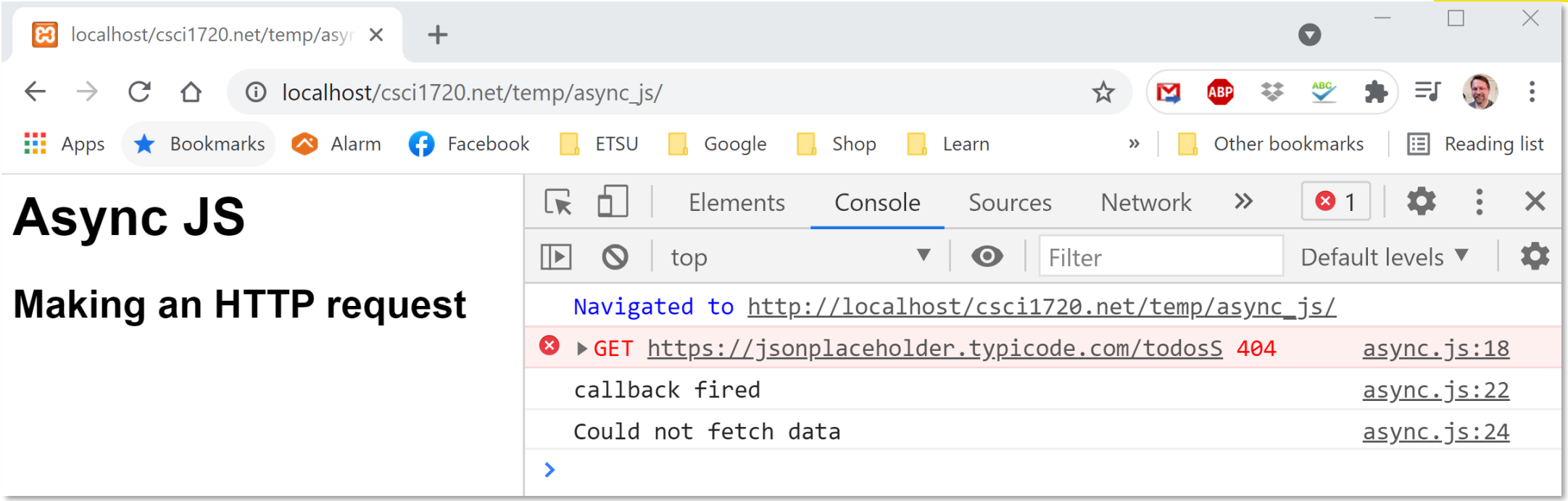
```
getTodos((err, data) => {
    console.log('callback fired');
    if(err) {
        console.log(err);
    } else {
        console.log(data);
    }
});
```

If we load this into the browser (without the 'Intentional error'):

XMLHttpRequest



XMLHttpRequest



ERROR OCCURRED

XMLHttpRequest

https://csci1720.net/lecture/video/lecture13-json/011.mp4

XMLHttpRequest

So now we've got the data back from the request without blocking the code

So, what do we do with it?

How can we manipulate the data once it has been returned?

The data is returned as text, but we need to convert it to something we can use

Enter JSON

XMLHttpRequest

https://csci1720.net/lecture/video/lecture13-json/012.mp4

XMLHttpRequest

JS

JSON.parse() will convert a properly formatted text string into an array of JSON objects

We can then access that array and process the data however our application requires

We could use a local JSON file for our data:

XMLHttpRequest

https://csci1720.net/lecture/video/lecture13-json/013.mp4

XMLHttpRequest

JS

Naturally, there are more issues with asynchronous requests…

What if we want to do multiple requests against (possibly) multiple endpoints

A typical use-case is getting data from one endpoint, doing something with it, then another endpoint, doing something, and so on

But for now, that's the basics of performing an asynchronous call in JavaScript

XMLHttpRequest

JS

You may be asking yourself at this point, "So what?"

We requested the data

We did some basic error checking

We got the data back

We logged it to the console

But, all "oohs" and "ahhs" aside, what next?

Well, here's one example

XMLHttpRequest

https://csci1720.net/lecture/video/lecture13-json/014.mp4

Copyrights