

# IN4MATX 133: User Interface Software

Lecture:  
AJAX, Fetch, & Promises

# Goals for this Lecture

**By the end of this lecture, you should be able to...**

- Explain how programs access web resources and common ways they respond
- Implement a fetch request to get a resource from a web API
- Use promises to make an asynchronous request

# Web APIs

- Many web services and data sources allow you to use HTTP (web) requests to access their data
- This is done by providing a web API.
- <https://developer.twitter.com/>



# Web APIs

## Application Programming Interface

- The *interface* we can use to interact with an *application* through *programming*
- An interface is just a defined set of functions

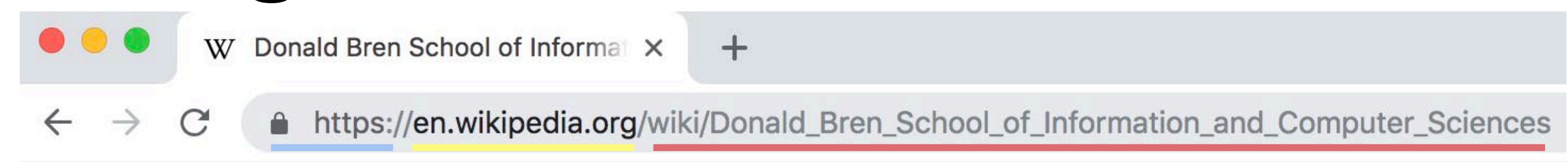
```
function doSomething(param1, param2) {  
  // ...  
}
```

↑      ↑  
An interface

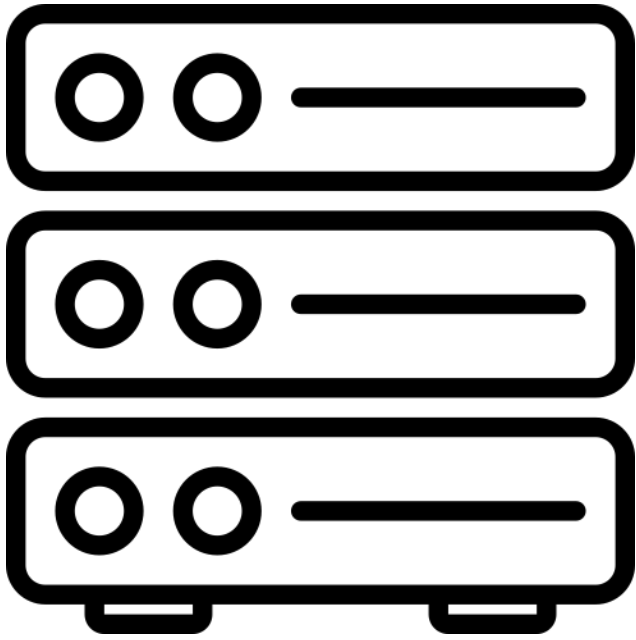
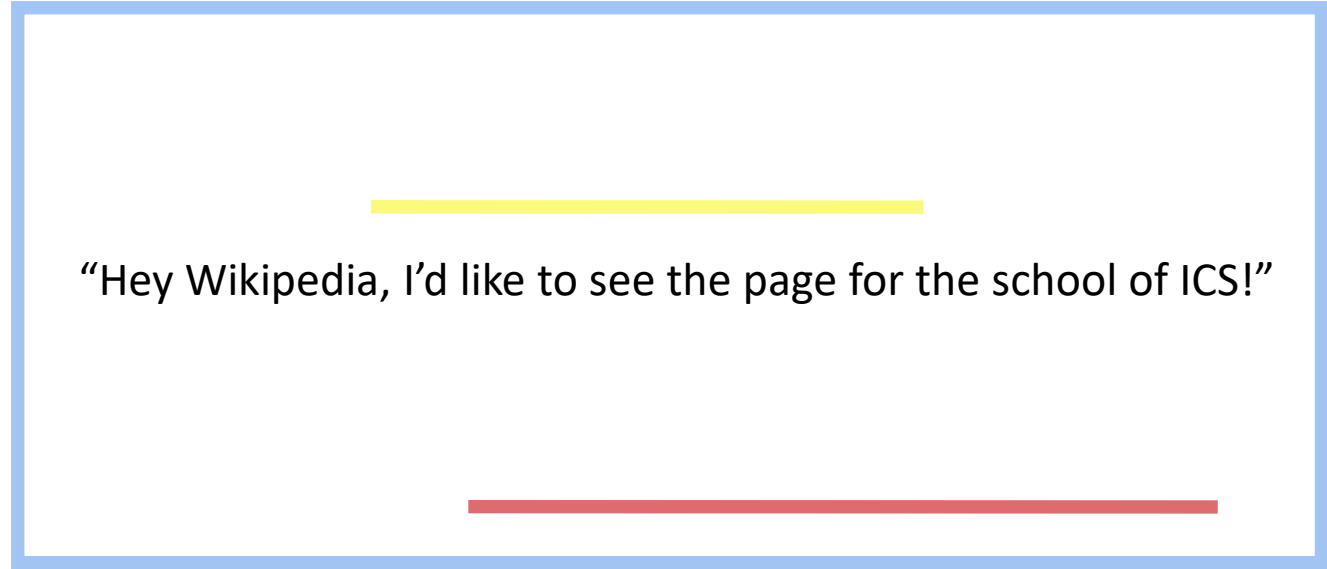
# Web APIs



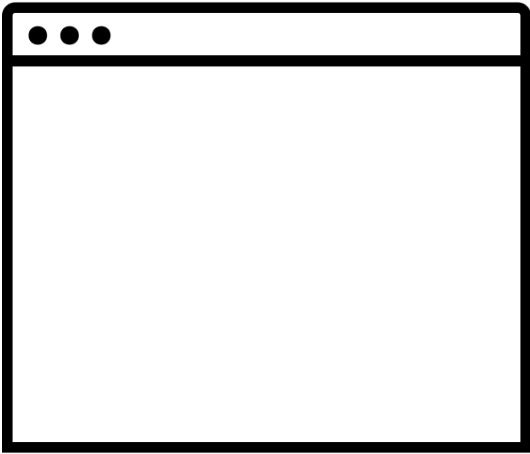
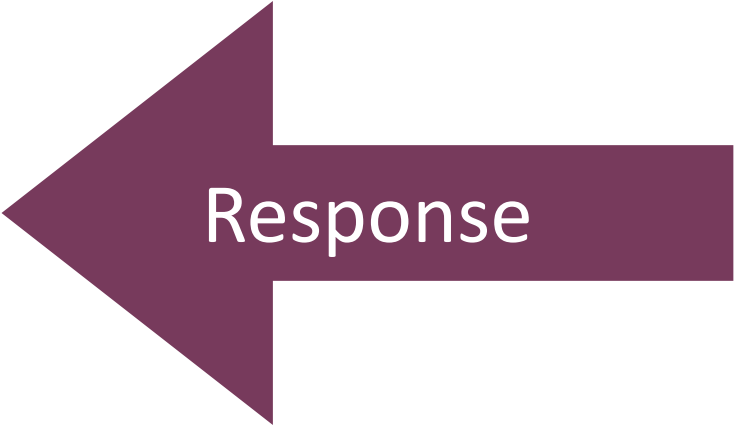
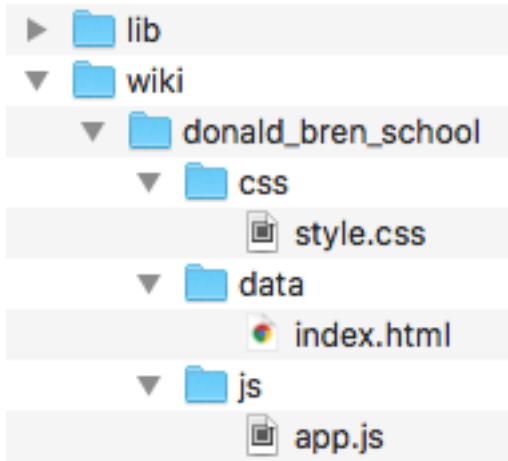
# Using the internet



Protocol    Host    Resource  
(how to handle info)    (who has info)    (what info you want)



Web server



Your browser

# URI

## Uniform Resource Indicator

- All URLs are URIs, but URLs also specify “access mechanism”
  - `http://`, `file://`
- URIs will return a resource
  - Could be a webpage, image file etc.
  - Could also just be data

# URI

## Uniform Resource Indicator

- `http://www.domain.com/users` => returns a list of users
  - The list of users is the *resource*
- Can have sub-resources
- `http://www.domain.com/users/shawna`
  - Returns a specific user



# URI format



- Base URI:
  - How every API request for that API starts
  - `https://api.twitter.com/`
- Endpoint
  - Specific resources which can be accessed via that api
  - `1.1/search/tweets.json`
  - `1.1/status/filter.json`



Endpoints often contain an API version number

<https://developer.twitter.com/en/products/tweets.html>

# URI queries

- Key/value pairs which follow the URI
  - Parameters for the resource, may specify exactly what to return or what format it should be in
  - `?key=value&key=value`
- `https://api.twitter.com/1.1/search/tweets.json?q=UCI&language=en`
  -  “query”, in Twitter this means what text or hashtag to search for
  -  `language=english`

# HTTP verbs

- HTTP requests include a target resource and a verb (method) specifying what to do with it
  - GET: return a representation of the current state of the resource
  - POST: add a new resource (e.g., a record, an entry)
  - PUT: update an existing resource to a new state
  - PATCH: update a portion of the resource's state
  - DELETE: remove the resource
  - OPTIONS: return a set of methods that can be performed on the resource

# HTTP responses

- Responses will include a *status code* (whether it worked as expected) and a *body* (the actual response)
  - 200: OK
  - 201: Created (for POST)
  - 400: Bad request (something is wrong with your URI)
  - 403: Forbidden (some access or authentication issue)
  - 404: Not found (resource does not exist)
  - 500: Internal server error (generic server-side error)

# Putting it all together

- HTTP GET  
`https://api.twitter.com/1.1/search/tweets.json?q=UCI&lang=en`
  - Use the “get” verb to access English-language tweets which mention UCI
  - We expect/hope for status code 200 (OK)
  - Then we access the *body*

# Escaping characters

- Some characters, like the hash (#) are reserved in URLs
  - Linking to IDs within pages
- We need to *encode* the character to search for a hashtag on Twitter
- HTTP GET  
`https://api.twitter.com/1.1/search/tweets.json?q=%23UCI&lang=en`

Character	From Windows-1252	From UTF-8
space	%20	%20
!	%21	%21
"	%22	%22
#	%23	%23
\$	%24	%24
%	%25	%25

[https://www.w3schools.com/tags/ref\\_urlencode.asp](https://www.w3schools.com/tags/ref_urlencode.asp)

So how do we make a web request?



Asynchronous JavaScript and XML



# XML

## Extensible Markup Language

- A generalized syntax for semantically defining structured content
- HTML is XML with defined tags

```
<person>  
  <firstName>Alice</firstName>  
  <lastName>Smith</lastName>  
  <favorites>  
    <music>jazz</music>  
    <food>pizza</food>  
  </favorites>  
</person>
```

# Plain text

```
Belgian Waffles
"Two of our famous Belgian Waffles with plenty of real maple syrup"
$5.95
650 calories

Strawberry Belgian Waffles
"Light Belgian waffles covered with strawberries and whipped cream"
$7.95
900 calories

Berry-Berry Belgian Waffles
"Light Belgian waffles covered with an assortment of fresh berries and whipped cream"
$8.95
900 calories

French Toast
"Thick slices made from our homemade sourdough bread"
$4.50
600 calories

Homestyle Breakfast
"Two eggs, bacon or sausage, toast, and our ever-popular hash browns"
$6.95
950 calories
```

# XML

```
<breakfast_menu>
  <food>
    <name>Belgian Waffles</name>
    <price>$5.95</price>
    <description>
      Two of our famous Belgian Waffles with plenty of real maple syrup
    </description>
    <calories>650</calories>
  </food>
  <food>
    <name>Strawberry Belgian Waffles</name>
    <price>$7.95</price>
    <description>
      Light Belgian waffles covered with strawberries and whipped cream
    </description>
    <calories>900</calories>
  </food>
  <food>
    <name>Berry-Berry Belgian Waffles</name>
    <price>$8.95</price>
    <description>
      Light Belgian waffles covered with an assortment of fresh berries and
whipped cream
    </description>
    <calories>900</calories>
  </food>
  <food>
    <name>French Toast</name>
    <price>$4.50</price>
    <description>
      Thick slices made from our homemade sourdough bread
    </description>
    <calories>600</calories>
  </food>
  <food>
    <name>Homestyle Breakfast</name>
    <price>$6.95</price>
    <description>
      Two eggs, bacon or sausage, toast, and our ever-popular hash browns
    </description>
    <calories>950</calories>
  </food>
</breakfast_menu>
```

# XML

```
<breakfast_menu>
  <food>
    <name>Belgian Waffles</name>
    <price>$5.95</price>
    <description>
      Two of our famous Belgian Waffles with plenty of real maple syrup
    </description>
    <calories>650</calories>
  </food>
  <food>
    <name>Strawberry Belgian Waffles</name>
    <price>$7.95</price>
    <description>
      Light Belgian waffles covered with strawberries and whipped cream
    </description>
    <calories>900</calories>
  </food>
  <food>
    <name>Berry-Berry Belgian Waffles</name>
    <price>$8.95</price>
    <description>
      Light Belgian waffles covered with an assortment of fresh berries and
      whipped cream
    </description>
    <calories>900</calories>
  </food>
  <food>
    <name>French Toast</name>
    <price>$4.50</price>
    <description>
      Thick slices made from our homemade sourdough bread
    </description>
    <calories>600</calories>
  </food>
  <food>
    <name>Homestyle Breakfast</name>
    <price>$6.95</price>
    <description>
      Two eggs, bacon or sausage, toast, and our ever-popular hash browns
    </description>
    <calories>950</calories>
  </food>
</breakfast_menu>
```

# JSON

```
{
  "breakfast_menu": {
    "food": [
      {
        "name": "Belgian Waffles",
        "price": "$5.95",
        "description": "Two of our famous Belgian Waffles with plenty of real maple
syrup",
        "calories": "650"
      },
      {
        "name": "Strawberry Belgian Waffles",
        "price": "$7.95",
        "description": "Light Belgian waffles covered with strawberries and whipped
cream",
        "calories": "900"
      },
      {
        "name": "Berry-Berry Belgian Waffles",
        "price": "$8.95",
        "description": "Light Belgian waffles covered with an assortment of fresh
berries and whipped cream",
        "calories": "900"
      },
      {
        "name": "French Toast",
        "price": "$4.50",
        "description": "Thick slices made from our homemade sourdough bread",
        "calories": "600"
      },
      {
        "name": "Homestyle Breakfast",
        "price": "$6.95",
        "description": "Two eggs, bacon or sausage, toast, and our ever-popular hash
browns",
        "calories": "950"
      }
    ]
  }
}
```

# XML vs. JSON

- XML and JSON represent the same data
- JSON is more concise
  - Less data to move around on the web
- JSON is easier to read
  - Close tags in XML are redundant
- JSON has taken over as the typical format of web requests



Asynchronous JavaScript and ~~XML~~

**JSON**

# Sending an *AJAX* request

# XMLHttpRequest

- AJAX requests are built into a browser-provided object called XMLHttpRequest

```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    if (xhttp.readyState == 4 && xhttp.status == 200) {
        // Action to be performed when the document is read;
        var xml = xhttp.responseXML;

        var movie = xml.getElementsByTagName("track");
        // ...
    }
};
xhttp.open("GET", "filename", true);
xhttp.send();
```



# XMLHttpRequest

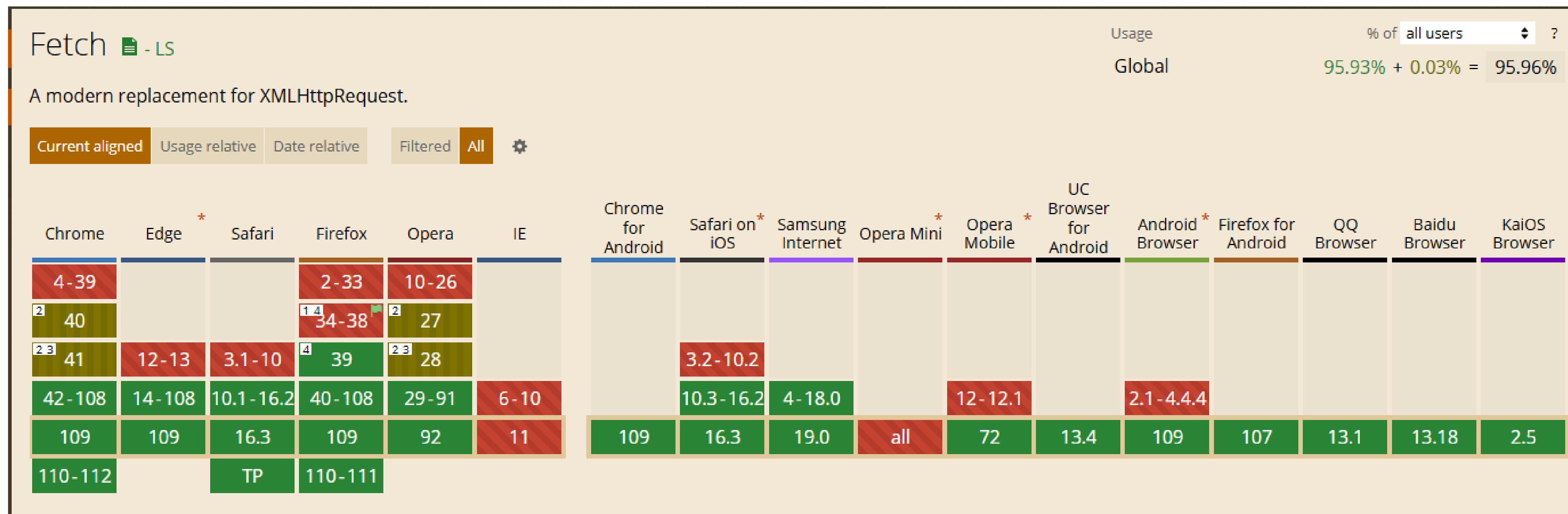
- AJAX requests are built into a browser-provided object called XMLHttpRequest

```
var xhttp = new XMLHttpRequest();  
xhttp.onreadystatechange = function() {  
    if (xhttp.readyState == 4 && xhttp.status == 200) {  
        // Action to be performed when the document is read;  
        var xml = xhttp.responseXML;  
  
        var movie = xml.getElementsByTagName("track");  
        //...  
    }  
};  
xhttp.open("GET", "filename", true);  
xhttp.send();
```



# Fetch

- A new-ish, modern method for submitting XMLHttpRequests
- Included in most browsers (but not IE)



<https://caniuse.com/fetch>

# Using fetch

- `fetch('some-url')` defaults to a GET request
- `fetch` can optionally take a second `options` argument (as a dictionary)
  - `method`: what method to use (e.g., POST, PUT, DELETE)
  - `headers`: specify content type format, etc. (more on headers in the next week)
  - `body`: what you want to send for a POST/PUT request

# Using fetch

- For a GET request

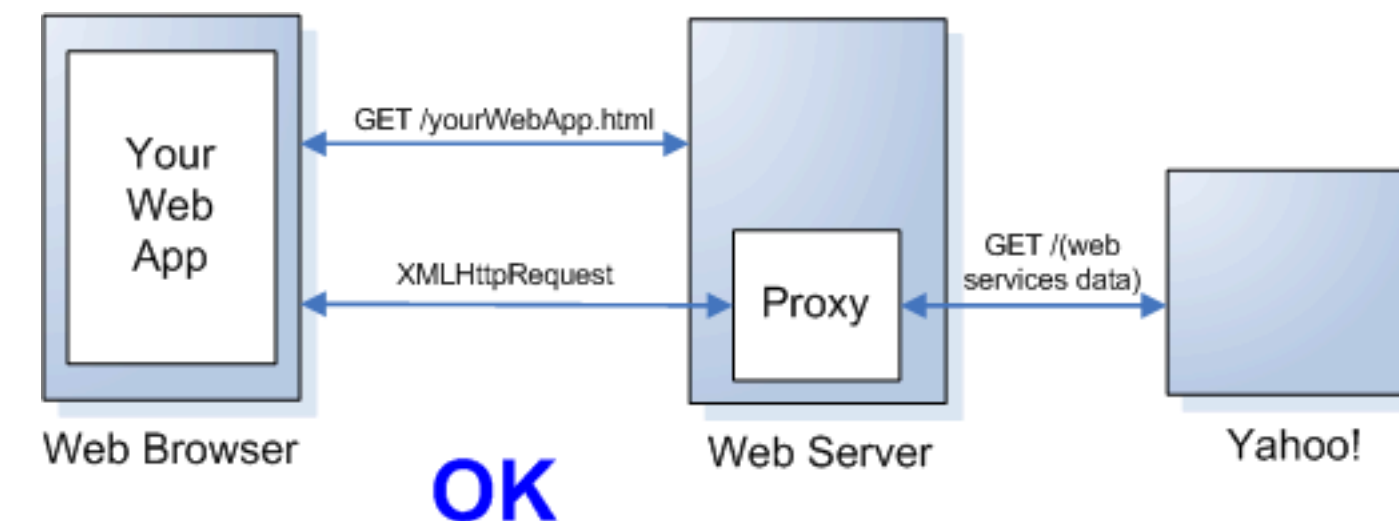
```
fetch('some-url');
```

- For a POST request

```
fetch('some-url', {  
  method: 'POST',  
  headers: { 'Content-Type': 'application/json' },  
  body: JSON.stringify(data-to-send)  
});
```

# Same-origin policy

- Many browsers will not permit AJAX requests to a different server. This helps prevent malicious scripts from accessing data in the DOM
  - A non-browser proxy server running locally can communicate with a different server
  - The browser can communicate with the proxy server



# Same-origin policy

- Two browser tabs: A bank app open in one, an evil app in the other
  - Both run JavaScript scripts written by their source
- The *origin* is what HTML page opened the JavaScript file
  - So each tab is a separate origin
- *Without* the same-origin policy, the evil app could read, edit, etc. your bank information
  - Different tabs, but both running with the same JavaScript engine



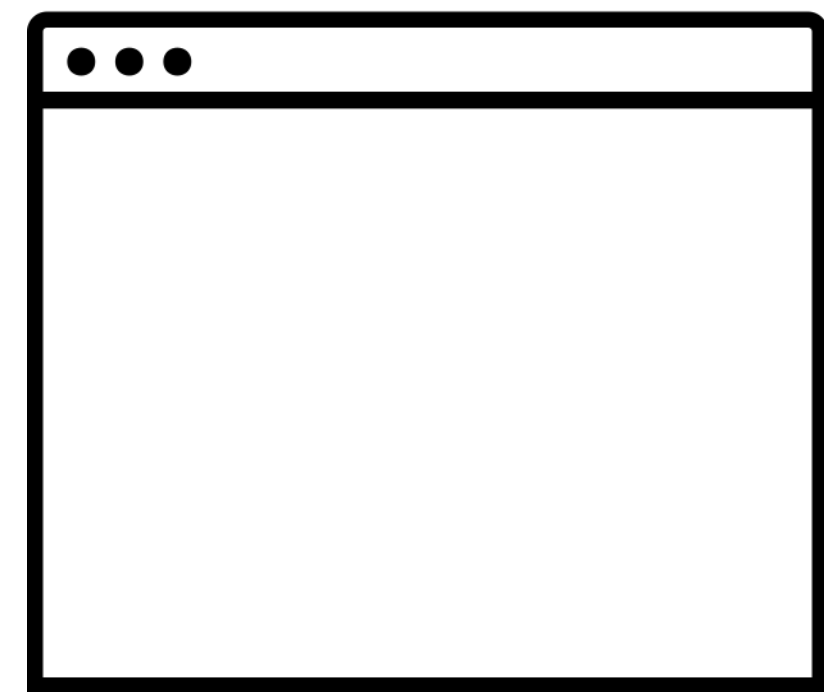
# Same-origin policy

- So instead, the bank app can only talk to the bank server, and the evil app can only talk to the evil server
- Two exceptions:
  - An app can always communicate with other apps in the same domain (e.g., localhost apps can communicate with any other localhost apps)
  - A server can designate that it will accept connections from sources with a particular origin (or any origin)
  - You *can* disable this in your browser, but probably shouldn't



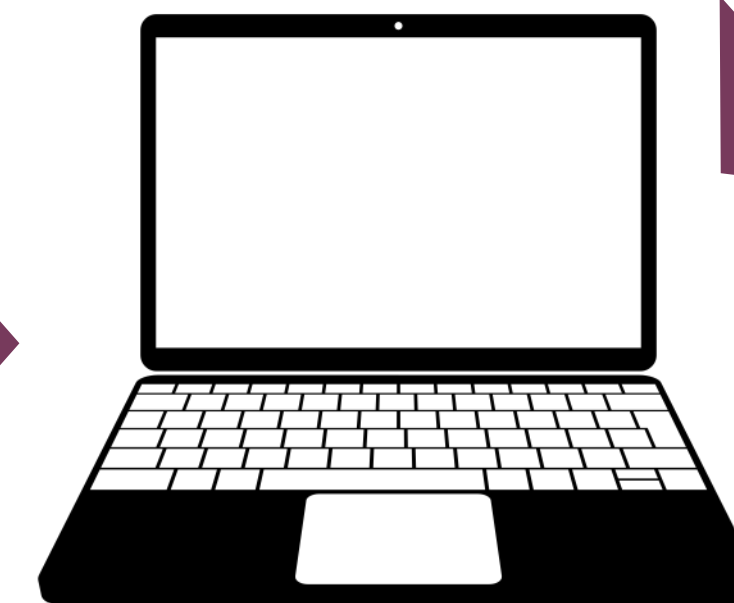
# Servers on localhost

- Localhost: “this computer”



Live server: localhost:8080

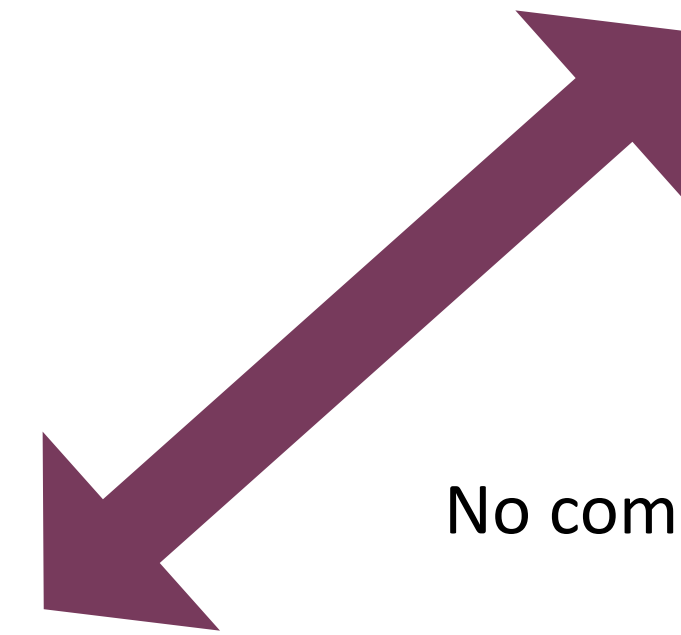
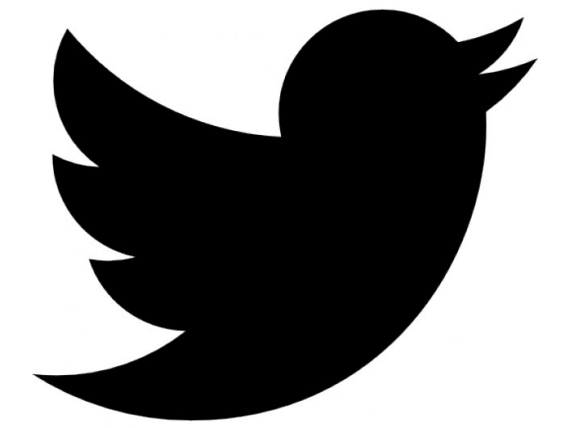
Same domain (localhost), so they can communicate



Twitter proxy: localhost:7890

*Browser* implements same-origin policy to protect the other data you have open in the browser

No same-origin policy restrictions, can communicate with Twitter



No communication restrictions

# A local web server

- Install live-server package globally
  - `npm install -g live-server`
- Running it
  - `cd path/to/project`
  - `live-server .`
- Will open up your webpage at <http://localhost:8080>





Asynchronous JavaScript and ~~XML~~

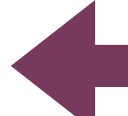
**JSON**

# Asynchronous requests

- Ajax requests are asynchronous, so they happen simultaneously with the rest of the code
- After the request is sent, the next line of code is executed **without waiting for the request to finish**

(1) `console.log('About to send request');`

`//send request for data to the url`

(2) `fetch(url);`  Does **NOT** return the data

(3) `console.log('Sent request');`

(4) Data is actually received sometime later!

# Asynchronous requests

- It's uncertain how long it'll take the request to complete
- Handling requests asynchronously allows a person to continue interacting with your page
  - The request is not blocking their interface interactions
  - It's a bad experience when a person tries to navigate your webpage, but can't

# Promises

- Because `fetch()` is asynchronous, the method returns a **Promise**
- Promises act as a “placeholder” for the data that will eventually be received from the AJAX request

`//fetch() returns a Promise`

`var thePromise = fetch(url);`

# Promises

- We use the `.then()` method to specify a **callback** function to be executed when the promise is *fulfilled* (when the asynchronous request is finished)

```
//what to do when we get the response
function successCallback(response) {
  console.log(response);
}
```



Callback will be passed the request response

```
//when fulfilled, execute the callback function
//(which will be passed the fetched data)
var promise = fetch(url);
promise.then(successCallback, rejectCallback);
```



Optional parameter

```
//more common to use anonymous variables/callbacks:
fetch(url).then(function(response) {
  console.log(response);
});
```

# fetch() responses

- The parameter passed to the `.then()` callback is the **response**, not the data we're looking for
- The `fetch()` API provides a method `.json()` that we can use to extract the data from the response
  - But this method is *also* asynchronous and returns a promise!

```
fetch(url).then(function(response) {  
    var newPromise = response.json();
```



Another promise



Not the data

```
// ... what now?  
});
```

# Chaining promises

- The `.then()` method itself returns a Promise containing the value (data) returned by the callback method
- This allows you to **chain** callback functions together, doing one after another (but *after* the Promise is fulfilled)

```
function makeString(data) {  
    return data.join(", "); //a value to put in Promise  
}
```

```
function makeUpper(string) {  
    return string.toUpperCase(); //a value to put in Promise  
}
```

```
var promiseA = getData(); // When completed, promiseA => json data  
var promiseB = promiseA.then(makeString); // promiseB => comma-separated string  
var promiseC = promiseB.then(makeUpper); // promiseC => uppercase string  
promiseC.then(function(data) {  
    console.log(data);  
});  
  
// Data is an uppercase,  
// comma-separated string
```

# Chaining promises

- The `.then()` method itself returns a Promise containing the value (data) returned by the callback method
- This allows you to **chain** callback functions together, doing one after another (but *after* the Promise is fulfilled)

```
function makeString(data) {  
    return data.join(", "); //a value to put in Promise  
}
```

```
function makeUpper(string) {  
    return string.toUpperCase(); //a value to put in Promise  
}
```

```
//more common to use anonymous variables and chain functions  
getData()  
    .then(makeString)  
    .then(makeUpper)  
    .then(function(d) { console.log(d); });
```



# Multiple promises (sequential)

- The .then() function will also handle promises *returned by previous callbacks*, allowing for sequential async calls

```
getData(fooSrc)
  .then(function(fooData) {
    var modifiedFoo = modify(fooData)
    return modifiedFoo;
  })
  .then(function(modifiedFoo) {
    //do something with modifiedFoo
    var barPromise = getData(barSrc);
    return barPromise;
  })
  .then(function(barData) {
    //do something with barData
  })
```

# Extracting fetch() data

- To actually download JSON data...

```
fetch(url)
  .then(function(response) {
    var dataPromise = response.json();
    return dataPromise;
  })
  .then(function(data) {
    //do something with data
  });
```

# Catching errors

- We can use the `.catch()` function to specify a callback that will occur if the promise is **rejected** (an error occurs).
- This method will “catch” errors from all previous `.then()`s

```
getData(fooSrc)
  .then(firstCallback)
  .then(secondCallback)
  .catch(function(error) {
    //called if EITHER previous callback
    //has an error

    //param is object representing the error itself
    console.log(error.message);
  })
  .then(thirdCallback) //will only do this if
                       //no previous errors
```

# Multiple promises (concurrent)

- Because Promises are just commands to do something, we can wait for all of them to be done

```
var foo = fetch(fooUrl);
```

```
var bar = fetch(barUrl);
```

```
//a promise for when all commands ready
```

```
Promise.all(foo, bar)
```

```
  .then(function(fooRes, barRes) {
```

```
    //do something both both responses, e.g.,
```

```
    return Promise.all(fooRes.json(), barRes.json());
```

```
  })
```

```
  .then(function(fooData, barData) {
```

```
    //now have both data sets!
```

```
  })
```

# Socrative Quiz!

Enter your UCI Email when prompted  
name!!!

e.g.,

[xxxxx@uci.edu](mailto:xxxxx@uci.edu)

<https://api.socrative.com/rc/CvereT>



# Goals for this Lecture

**By the end of this lecture, you should be able to...**

- Explain how programs access web resources and common ways they respond
- Implement a fetch request to get a resource from a web API
- Use promises to make an asynchronous request