Week 8: REST APIs, Fetch & JSON

INFSCI 2560
Web Standards & Technologies

Housekeeping

- Project 2
  - “Rubric” guidelines posted.
  - Come to class next week, there will be a participation grade.
- Exam 2 is expected to be returned next class
- Looking forward:
  - 3 lectures remaining, so it is important to attend and participate.
  - 1 more project, 1 final (group) project
  - Exam 2 is 11/18
  - Project Presentations will be 12/2 and 12/9
- Tips for succeeding in this class
  - Understand, don’t memorize.
  - Take notes. The lecture slide deck is not the textbook.
  - Read the readings and linked articles
  - Do the examples, plus some (challenge yourself).

Today's Agenda

1. Web APIs & REST
2. JSON
3. Fetch API
4. Activity 7

Web Application Programming Interfaces (APIs)

- **API**: A set of subroutine definitions, protocols and tools for building software and applications
- **Web API:** an API which is accessed using the Internet using HTTP protocols
- Defines a set of endpoints, request messages and response structures.
- You can build Web APIs using different technologies such as Java, JavaScript, .NET, etc.
- Sometimes called *web services*.
- Example APIS: Twitter, Reddit, AccuWeather

Let's look at a simple a web API

- Imagine having an API that takes a time zone string and returns the current time for that time zone
- This API would take a string like "America/Los_Angeles" and return "2019-02-28T20:09:45-07:00"
- One design of the API might look like this:
- This API has an *endpoint*, `/timezone` that expects a *query parameter*, `tz={Timezone location}`

```
http://api.example.com/timezone?tz=America+Los_Angeles

    {

      "time": "2019-02-28T20:09:45-07:00",

      "zone": "America/Los_Angeles"

    }
```

Why do we use web APIs?

- Helps to accelerate development by providing common functionality out-of-the-box.
- You don't need to build multiple interfaces to interact with the same data set(s).
- Tool to safely expose data or functionality to other developers
- Easy to consume the same web API using different programming languages.

RESTful APIs

Representational State Transfer (REST)

- REST is a software architectural style (*not* a technology)
- Provides a set of principles for designing RESTful web services
- A way of designing Application Programming Interfaces (APIs) for the web
- Not a formal standard, but rather an approach to designing APIs
- This means there are varying levels of restfulness
- And many opinions

RESTful APIs

Composed of:

- Uniform Resource Identifiers (URIs) or URL
  - An identifier of a resource. The path to get the resource.
- Resources
  - Anything that can be identified by a URI/URL
  - This can be anything - an image, a file, an HTML page or any other data
- Representations
  - The resource that the server returns is a representation of a resource
  - An encapsulation of the information (state, data, or markup) of the resource, encoding using a format such as XML, JSON or HTML.

RESTful APIs

Composed of:

- Uniform Interface
  - The (uniform) interface defines the communication between the client and the server.
  - Consists of the OPTIONS, GET, HEAD, POST, PUT, DELETE and TRACE methods.
  - Use of a uniform (or consistent) interface makes the request and responses self-describing and visible.

REST Architecture

- REST has a set of theoretical architectural constraints that inform how to design APIs
- Resourced Based
  - Nouns or things vs. verbs or actions
  - Everything is identified via their URI
  - Things are different from the representation of a thing
- Representations
  - How resources are manipulated in REST Systems
  - Representations are the things transferred between the client and the server
  - You can have more than one representation of a resource (JSON & XML format)
  - The representation contains enough information to customize or delete the resource on the server (e.g. ID)

6 REST Constraints

Uniform Interface

All resources should be accessible through a common approach such as HTTP GET and similarly modified using a consistent approach.

Stateless

Server will not store anything about latest HTTP request client made. It will treat each and every request as new. No session, no history. State is maintained by the client.

6 REST Constraints

Cacheable

To enable large scale services, caching shall be applied to resources when applicable and then these resources MUST declare themselves cacheable.

Client-Server

A client application and a server application MUST be able to evolve separately without any dependency on each other. A client should know only resource URIs and that’s all.

6 REST Constraints

Layered System

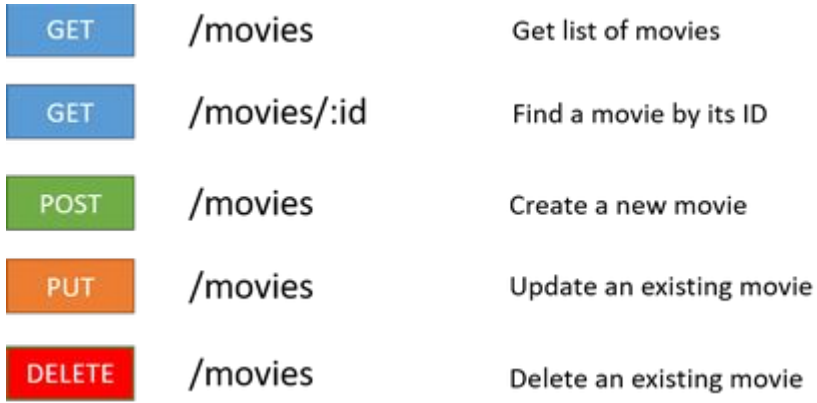
A RESTful architecture can be composed of several hierarchical layers that the client does not need to be aware of. These layers (e.g. security, caching, load-balancing) should not affect the request or the response.

Code on Demand (Optional)

REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented.

REST in Practice

- Identify the nouns in your system.
- You define your API via URLs
- Use resource methods to transform the state of your resources.
- This is often HTTP verbs (GET, POST, PUT, DELETE) but REST does not explicitly specify what resource methods, only that they have a uniform interface



REST API Endpoints. Image from HackerNoon

REST Resources

- What is REST? A simple explanation (Medium article, definitely read!)
- RESTful Webservices Cookbook (if you want to dig deeper)

Let’s look at some example REST calls.

HTTP GET Request Example

```
GET https://developer.mozilla.org/en-US/search?q=client+server+overview HTTP/1.1
Host: developer.mozilla.org
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 ...
Accept: text/html,application/xhtml+xml, ...
Referer: https://developer.mozilla.org/en-US/
Accept-Encoding: gzip, deflate, sdch, br
Accept-Charset: ISO-8859-1,UTF-8;q=0.7,*;q=0.7
Accept-Language: en-US,en;q=0.8,es;q=0.6
Cookie: sessionid=6ynxs23n521lu21b1t136rhbv7ezngie;
```

- Type of request - `GET`
- Target Resource `/en-us/search`
- URL Parameters - `q=client%2Bserver%2Boverview&`
- Host website - `developer.mozilla.org`
- Then there is a bunch of information in the HTTP Headers
  - `User-Agent` - What browser or client made the request
  - `Accept-Encoding` - Can the response be compressed to save bandwidth and what formats are supported
  - `Accept-Charset` and `Accept-Language` - What character encoding and languages are acceptable
  - `Referer` - What is the address

HTTP GET Response Example

```
HTTP/1.1 200 OK
Server: Apache
X-Backend-Server: developer1.webapp.scl3.mozilla.com
Vary: Accept,Cookie, Accept-Encoding
Content-Type: text/html; charset=utf-8
Date: Wed, 07 Sep 2016 00:11:31 GMT
Keep-Alive: timeout=5, max=999
Connection: Keep-Alive
X-Frame-Options: DENY
Allow: GET
X-Cache-Info: caching
Content-Length: 41823
```

```
<!DOCTYPE html>
<html lang="en-US" >
<head prefix="og: http://ogp.me/ns#">
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=Edge">
  <script>(function(d) {
    d.className = d.className.replace(/\bno-js/, '');
  })(document.documentElement);</script>
  ...
```

Some of the relevant HTTP headers:

- First is the *response code* - 200 ok which means everything is good
- Content-Type - tells us what the MIME type of the content is. Here we have text/html
- charset - tells us the character encoding of the response
- Content-Length - tells us how big the body content is in bytes
- Finally there is the actual content, in this case some HTML that goes on for 41832 bytes (41kilobytes)

HTTP POST Request Example

```
POST https://developer.mozilla.org/en-US/profiles/hamishwillee/edit HTTP/1.1
Host: developer.mozilla.org
Connection: keep-alive
Content-Length: 432
Pragma: no-cache
Cache-Control: no-cache
Origin: https://developer.mozilla.org
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64)
  AppleWebKit/537.36 (KHTML, like Gecko) Chrome/52.0.2743.116 Safari/537.36
Content-Type: application/x-www-form-urlencoded
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Referer: https://developer.mozilla.org/en-US/profiles/hamishwillee/edit
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.8,es;q=0.6
Cookie: sessionid=6ynxs23n521lu21b1t136rhbv7ezngie;
  _gat=1; csrftoken=zIPUJsAZv6pcgCBJSCj1zU6pQZbfMUAT;
  dwf_section_edit=False; dwf_sg_task_completion=False;
  _ga=GA1.2.1688886003.1471911953; ffo=true

csrfmiddlewaretoken=zIPUJsAZv6pcgCBJSCj1zU6pQZbfMUAT
&user-username=hamishwillee
&user-fullname=Hamish+Willee
&user-title=
&user-organization=
&user-location=Australia
&user-locale=en-US
&user-timezone=Australia%2FMelbourne&user-irc_nickname=
```

- This is a POST Request
- Notice there are no URL parameters
- Instead the information is contained in the *body*
- Looks like an HTML Form submission
- Newlines have been added for readability (usually one long line)

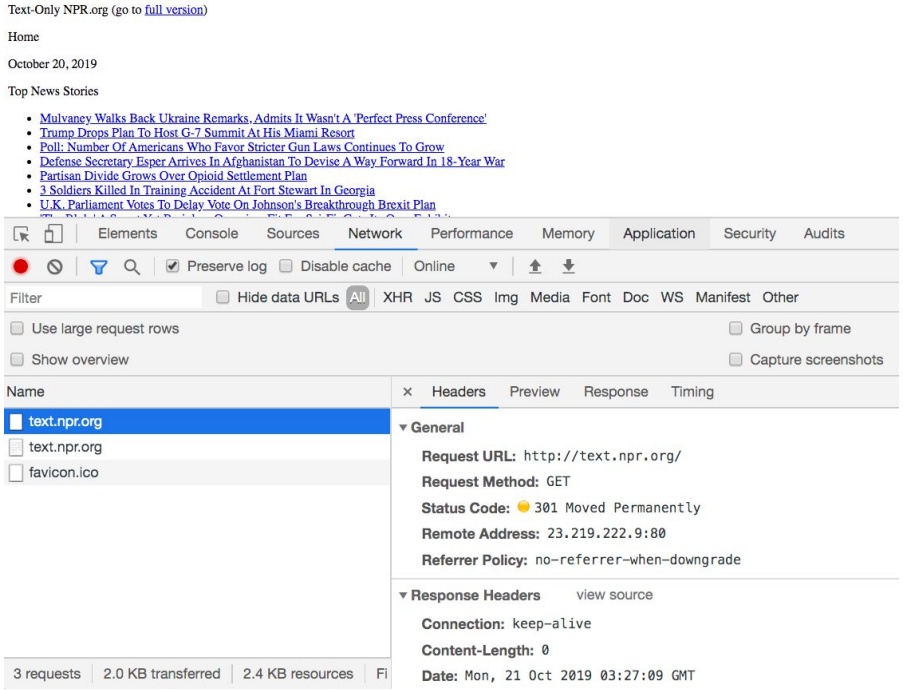
HTTP POST Response Example

```
HTTP/1.1 302 FOUND
Server: Apache
X-Backend-Server: developer3.webapp.scl3.mozilla.com
Vary: Cookie
Vary: Accept-Encoding
Content-Type: text/html; charset=utf-8
Date: Wed, 07 Sep 2016 00:38:13 GMT
Location: https://developer.mozilla.org/en-US/profiles/hamishwillee
Keep-Alive: timeout=5, max=1000
Connection: Keep-Alive
X-Frame-Options: DENY
X-Cache-Info: not cacheable; request wasn't a GET or HEAD
Content-Length: 0
```

- This is a POST Response to an edit url
- The HTTP code 302 FOUND means the submission was a success
- The body is empty (Content-Length: 0). This is a redirect
- The browser now needs to make an additional request to the page specified by the Location: header
- Sends us to an updated user page

HTTP Request/Response Sniffing

- Open the Web Inspector
  - Right-click "Inspect" on Chrome
  - Right-click "Inspect Element" on Safari and Firefox
- Select "Network"
- Type in a URL (http://text.npr.org/ is a simple)



JavaScript Object Notation (JSON)

- A lightweight format for storing and transferring data
- Is a text-based format that is valid JavaScript code
- It is "self-describing" and easy to read and understand
- While it uses JavaScript syntax, it is supported by nearly every programming language
- Supports strings, numbers, objects, arrays, booleans and null values.

```
{ "name":"John", "age":30, "car":null }
```

JavaScript Object Notation (JSON)

- JSON objects are surrounded by curly braces {}.
- JSON objects are written in key/value pairs.
- Keys must be strings, and values must be a valid JSON data type (string, number, object, array, boolean or null).
- Keys and values are separated by a colon.
- Each key/value pair is separated by a comma.

```
{ "name":"John", "age":30, "car":null }
```

JSON Data Types

- **Strings** - Strings must be double quotes. `{"name":"Bob"}`
- **Numbers** - Numbers must be integer or floating point numbers.

```
5, 5.6
```

- **Objects** - Values can be JSON objects.

```
{"employee":{ "name":"John", "age":30, "city":"New York" }}
```

- **Arrays** - Arrays must be an order list of any values.

```
{"employees":[ "John", "Anna", "Peter" ]}
```

- **Boolean** - Must be true or false value. `{"sale":true}`
- **Null** - Values can also be null. `{"middlename":null}`

JSON Objects

An object is an unordered set of name/value pairs.

An object begins with { (left brace) and ends with } (right brace).

Each name is followed by : (colon) and the name/value pairs are separated by , (comma).

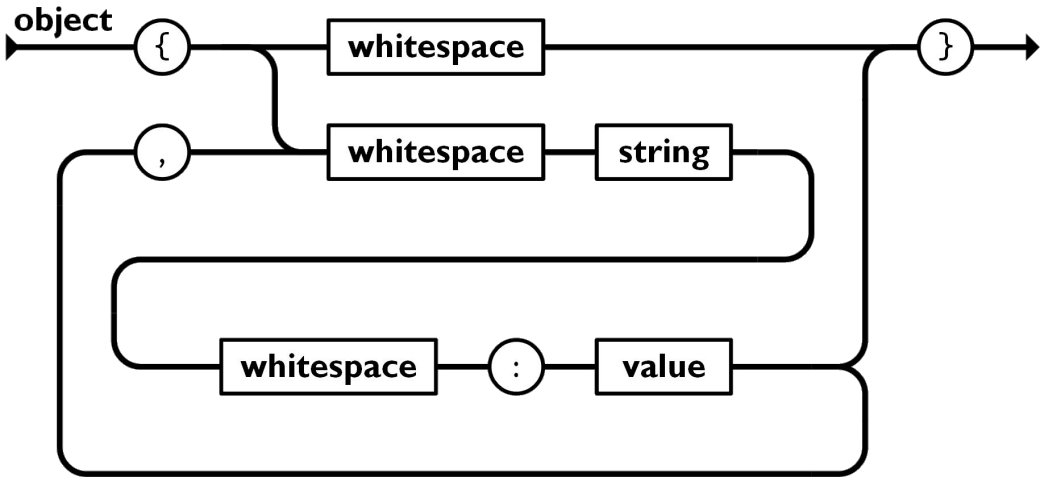


Image from json.org

JSON Arrays

An array is an ordered collection of values.

An array begins with [(left bracket) and ends with] (right bracket). Values are separated by , (comma).

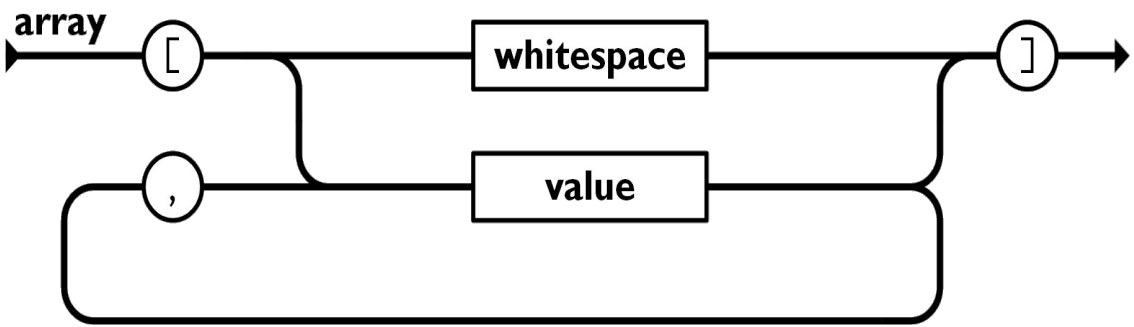


Image from json.org

JSON values

A value can be a string in double quotes, or a number, or true or false or null, or an object or an array.

These structures can be nested.

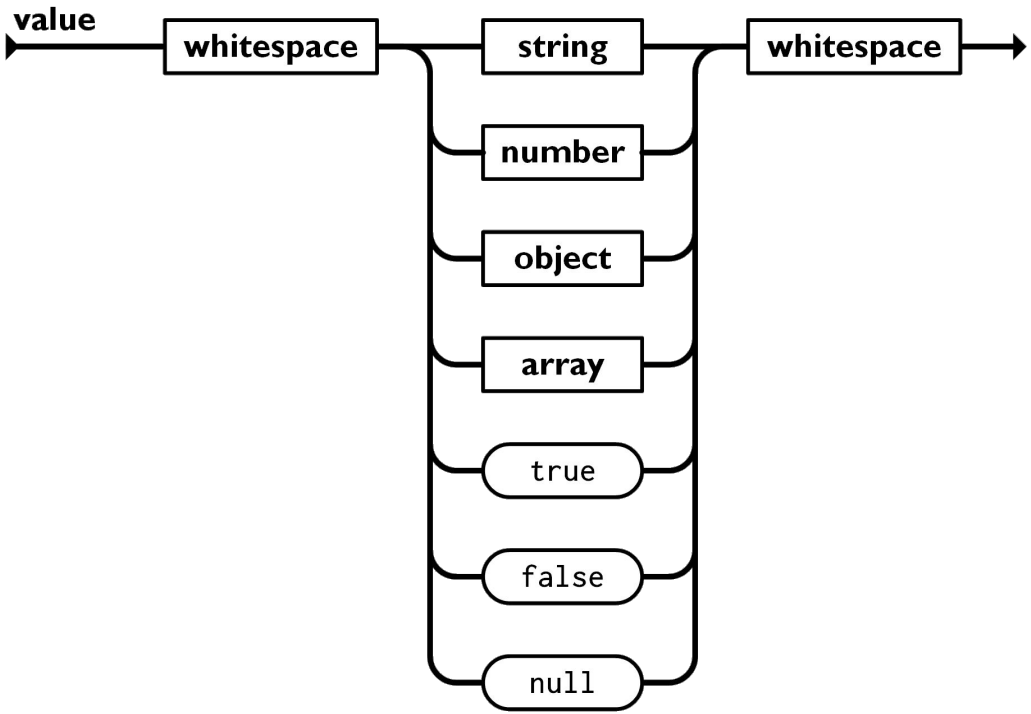


Image from json.org

JSON Strings, Characters and Numbers

- A string is a sequence of zero or more Unicode characters, wrapped in double quotes, using backslash escapes.
- A character is represented as a single character string. A string is very much like a C or Java string.
- A number is one or more positive or negative digits and decimals.

You can validate if a string is valid JSON using online JSON validators, such as JSONLint or JSON Formatter.

Parsing JSON in JavaScript

- A common use of JSON is to exchange data to/from a web server
- When receiving data from a web server, the data is always a string
- `JSON.parse()` is *one of* the methods available to convert text to a JSON object.
- Note: Dates and functions will be read in as strings. You must convert them manually with `new Data(value)` and `eval(value)`

Parsing JSON in JavaScript

Given a JSON string

```
'{ "name":"John", "age":30, "city":"New York"}'
```

We can parse it into a native Javascript Object using `JSON.parse()`

```
var obj = JSON.parse('{ "name":"John", "age":30, "city":"New York"}');
obj.name
> "John"
```

Parsing JSON in JavaScript

When working with JSON arrays you will get a JavaScript List

```
> var listy = JSON.parse('[{"first":"Iggy","last":"Peck","city":"Pittsburgh"},\
        { "first":"Ada", "last":"Marie", "city":"New York"}]');
  listy[0].city
<- "Pittsburgh"
```

Serializing JSON in JavaScript

When sending data from a web server, the data must be a string

Serialize the data with `JSON.stringify()` to transform JavaScript objects into string.

Note: Dates will be turned into strings and functions will be ignored

Serializing JSON in JavaScript

Given a JSON Object

```
{ name:"John", age:30, city:"New York"}
```

We can serialize it into a string using `JSON.stringify()`

```
var obj = { "name":"John", "age":30, "city":"New York"};
JSON.stringify(obj)
> '{"name":"John","age":30,"city":"New York"}'
```

eXtensible Markup Language (XML)

- Markup language like HTML
- Designed to store and transport data
- Designed to be self-descriptive

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

JSON vs XML

JSON is Unlike XML because:

- JSON is shorter
- JSON is quicker to read and write
- JSON can use arrays
- XML has to be parsed with an XML parser. JSON can be parsed by a standard JavaScript function.

JSON vs XML

JSON is like XML because:

- Both JSON and XML are "self describing" (human readable)
- Both JSON and XML are hierarchical (values within values)
- Both JSON and XML can be parsed and used by lots of programming languages
- Both JSON and XML can be fetched with AJAX

JSON vs XML

```json
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

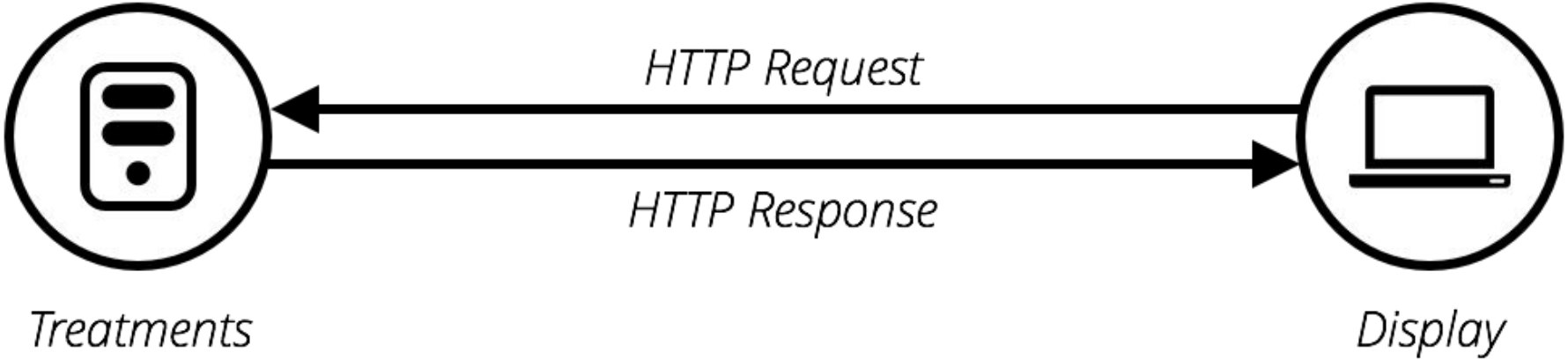
```xml
<menu id="file" value="File">
  <popup>
    <menuitem value="New" onclick="CreateNewDoc()" />
    <menuitem value="Open" onclick="OpenDoc()" />
    <menuitem value="Close" onclick="CloseDoc()" />
  </popup>
</menu>
```

Why JSON is better than XML

- XML is hard to parse, JSON is parsed into a ready-to-use JavaScript object.
- When using JS with XML:
  - Fetch an XML document
  - Use the XML DOM to loop through the document
  - Extract values and store in variables
  - Process the data
- When using JS with JSON:
  - Fetch a JSON string
  - JSON.Parse the JSON string into a JavaScript object
  - Process the data

Break (10 minutes)

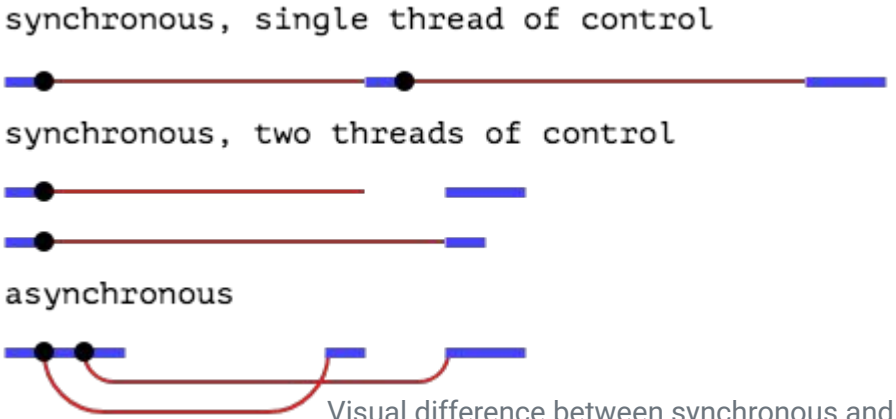
Making incremental updates to web pages



- Request made to the server, the server would send HTML and other assets (images, CSS, JavaScript) back and the browser would render the page.
- How do we make incremental updates without reloading the page?

Fetch API

- Allows us to make asynchronous requests
- Load data outside of the standard page request and loading process.
- Makes web pages faster and more responsive
- Cuts down on the amount of network traffic and loading required on the client and server.



Visual difference between synchronous and asynchronous processing. Image from Eloquent JavaScript

Fetch API

Allows us to fetch resources without reloading the page.

`fetch(url)` - GETs a resource

The then() part is the promise - event handler function with the onload event.

```
//a very basic Fetch example
fetch("http://example.com/api")
     .then(res => res.json())
     .then(json => console.log(json));
```

Understanding Promises

Promises can be a bit confusing, but they are the underpinning of many modern JavaScript APIs (like Fetch)

It represents a placeholder for an object that will eventually have a value.

It provides a generic interface, then(), for executing a callback function, passed as an argument to then(), when the object has value.

This means you can do method chaining and write cleaner code.

This, this and this are good resources for learning more about promises.

Understanding Promises

You can make an expression wait or pause for a Promise to be fulfilled using the `await` expression.

You will see examples online that use this notation.

Can only be used inside an `async` function.

```
async function f1(){

    var x = await waitNseconds(10);

}
```

Explaining a Fetch API example

1. The `fetch()` function returns a promise with the `Response` of the operation.
2. The `json()` method takes the `Response` and returns a promise that will trigger once the JSON response has arrived.
3. The JSON object contains an array of orders. Each order has an `id` attribute. Use the map function to retrieve the id from each order.
4. Print each orderId to the console

```
// make a simple GET request
fetch("http://example.com/orders")

      //Once the fetch completes, do this
      .then(rJSON => rJSON.json())

      //Once the JSON arrives, do this
      .then(res => res.map(order => order.id))

      //Now process the array of orderIds
      .then(orderIds => console.log(orderIds));
```

```
["92998-3874", "90566-7771", "59590-4157", "53919-4257", "33263", "23505-1337", "58804-1099", "45169", "76495-3109", "31428-2261"]
```

A note about security

Fetch follows the same-origin policy

This means they can only make requests to URLs with the same protocol, port, and host

Unless you specify a Cross-Origin Resource Sharing (CORS) header to grant permission to access resources at other origins

So if you load a page from the following URL you will only be able to load some resources with Fetch.

A note about security

URL	Outcome	Reason
`http://store.company.com/dir2/other.html`	Same origin	Only the path differs
`http://store.company.com/dir/inner/another.html`	Same origin	Only the path differs
`https://store.company.com/page.html`	Failure	Different protocol
`http://store.company.com:81/dir/page.html`	Failure	Different port (`http://` is port 80 by default)
`http://news.company.com/dir/page.html`	Failure	Different host

Activity 7

Due Wednesday by 11:59

The purpose of this activity is to allow you to play around with promises and the Fetch API in JavaScript. Your task is to create a simple GET request using the Random User API and then display the results.

https://glitch.com/~2560-activity7