Week 9 Day 1 Study Guide

Use this outline to guide you through today's readings and practices. This will focus your attention on the most important points to prepare you for the weekly assessment.

Browser Basics

Learning Objectives:

- Run JavaScript on the browser by importing scripts into HTML files
- Import JavaScript from one file into another
- Diagram the process in which the browser loads HTML, images, CSS stylesheets, scripts, fonts, and other assets
- Compare and contrast running JavaScript in Node in a console vs. the runtime environment on the browser
- Execute specified JavaScript after all of the elements in a page have loaded
- Compare and contrast DOM and BOM
- Manipulate key elements of the BOM, including the `window` and the `document`, using JavaScript

Chrome Developer Tools

- Elements tab
    - test and manipulate your HTML and CSS
- Console tab
    - test JavaScript and debug JavaScript code on the frontend
- Sources tab
    - inspect an application's file structure and create/edit files to the application
- Network tab
    - see HTTP requests the page is making

- Application tab
    - view and manipulate the application's data (Web Storage and Cookies)

Importing JavaScript files into HTML

- Using the <script> HTML tag with the src attribute, you can load JavaScript files into an application that will be run when loaded
- The JavaScript files will be loaded in the order that they are placed in the HTML document (from top-down)
- Placement of the <script> tags determine when the JavaScript file that it imports gets loaded
- Best practices for where to place the <script> tags:
    1. As children in the <head> HTML element
        - JavaScript files that are imported this way may be loaded and run BEFORE all the HTML DOM elements are constructed in the <body> HTML element
    2. As the last child elements of the <body> HTML element
        - This will guarantee the JavaScript files that are imported to be loaded and run AFTER all the HTML DOM elements are constructed before them

Example of imported JavaScript files inside of the <head> HTML element:

```
<!DOCTYPE html>
<html>
 <head>
   <title>Importing JavaScript to HTML</title>
   <script src="your-script-here.js"></script>
   <script src="../your-other-script-here.js"></script>
   <script src="/another-script-here.js"></script>
 <head>
 <body>
   ...
 </body>
</html>
```

Example of imported JavaScript files at the end of the <body> HTML element:

```
<!DOCTYPE html>
<html>
 <head>
   <title>Importing JavaScript to HTML</title>
   ...
 <head>
```

```
 <body>
   ...
   <script src="your-script-here.js"></script>
   <script src="../your-other-script-here.js"></script>
   <script src="/another-script-here.js"></script>
 </body>
</html>
```

DOM vs. BOM

- **BOM** - Browser Object Model, the chief browser object is the `window`
    - useful properties on the `window` object:
        - `navigator` - information about the browser and device
            - ex: `window.navigator.cookieEnables` or `navigator.cookieEnables`
        - `screen` - information about the dimensions and how the HTML is rendered
            - ex: `window.screen.height` or `screen.pixelDepth`
        - `history` - interface for reading and manipulating the browser history
            - ex: `window.history.back()` or `history.go(-1)`
        - `location` - interface for reading and manipulating the URL
            - ex: `window.location.host` or `location.pathname`
        - `document` - interface for reading and manipulating the HTML elements
            - ex: `window.document.title` or `document.body`
            - `document.body.children` results in an array-like object whose elements are the child HTML elements of the <body> HTML element
- **DOM** - Document Object Model, or `window.document` or just `document`
    - contains a collection of HTML elements
    - the web page and the object hierarchy of the HTML document

ES6 Modules

- used to import/export JavaScript files into other JavaScript files

- `export default ...` - statement to export one unnamed item per file
    - the item will be named when imported
    - can only have one `default` or unnamed export per file
- `export ...` - keyword to export multiple named items per file
    - the items will be named when exported
    - can have as many named exports per file
- `import ... from ...` statement to import items from one file to another
    - define the items you want to import between `import` and `from`
    - define the file path of the file you want to import from after `from`
        - takes in a relative file path
        - must have the file extension name in the file path
        - ex: `./exported-items.js`
    - ex: `import printHelloWorld from './export.js'` where `printHelloWorld` is the `default` exported item from the `export.js` file in the same folder as the file that you are importing into
- `as` keyword (in an `import ... from ...` statement) to alias and namespace all of a file's exported items

Examples

Given HTML:

```
<!DOCTYPE html>
<html>
 <head>
   <title>Importing JavaScript to HTML</title>
   <script type="module" src="import.js"></script>
 <head>
 <body>
   ...
 </body>
</html>
```

How to import an exported unnamed item:

```
// export.js
export default function() {
  console.log('Hello World!');
}
```

```
// import.js
import printHelloWorld from './export.js';

printHelloWorld(); // "Hello World!" will be printed to the
console
```

How to import multiple exported named items:

```
// export.js
export const hello = "Hello World!";
export const oneStep = "One Step at a Time...";
export function printPhrase(phrase) {
  console.log(phrase);
}
// import.js
import { hello, printPhrase } from './export.js';

printPhrase(hello); // "Hello World!" will be printed to the
console
printPhrase(oneStep); // Error: oneStep is not defined
```

OR using an alias:

```
// import.js
import * as allNamedItems from './export.js';

console.log(allNamedItems); // { hello, oneStep, printPhrase }

allNamedItems.printPhrase(allNamedItems.hello); // "Hello World!"
will be printed to the console
allNamedItems.printPhrase(allNamedItems.oneStep); // "One Step at a
Time..." will be printed to the console
```

How to import multiple exported named items AND an unnamed item:

```
// export.js
export const hello = "Hello World!";
export const oneStep = "One Step at a Time...";
export default (phrase) => {
  console.log(phrase);
}
// default export above could have been written as:
// export default function nameDoesntMatterHere(phrase) {
//   console.log(phrase);
// }
```

```
// import.js
import printPhrase, { hello, oneStep } from './export.js';

printPhrase(hello); // "Hello World!" will be printed to the
console
printPhrase(oneStep); // "One Step at a Time..." will be printed
to the console
```

Did you find this lesson helpful?

No

Yes

✓ Mark As Complete

Finished with this task? Click **Mark as Complete** to continue to the next page!

Week 8 Day 2 Study Guide

Use this outline to guide you through today's readings and practices. This will focus your attention on the most important points to prepare you for the weekly assessment.

Element Selection and Manipulation

Learning Objectives:

- Compare and contrast `NodeList` and `HTMLCollection`
- Write a JavaScript statement that selects one or more elements by their attributes or tags
- Write JavaScript to add/remove attributes to an HTML element(s)
- Write JavaScript to get the children elements of a given parent element
- Write JavaScript to create/remove an HTML element(s) from the DOM
- Append a child HTML element to a parent HTML element using JavaScript
- Use a string to construct HTML elements with `.innerHTML`
- Compare and contrast `.innerHTML` and `.innerText` methods on an HTML element
- Write JavaScript to add/remove CSS inline-styling to an HTML element(s)
- Manipulate DOM elements using the response of a `fetch` request

NodeList vs. HTMLCollection

See 00-NodeLists-vs-HTMLCollections for definitions and examples of Nodes, HTML Elements, NodeLists, and HTMLCollections.

Element Selectors

- **element selectors** in JavaScript - methods on the DOM or `document` (`window.document`) that will help you easily find HTML elements or Nodes on the DOM
    - different kinds of element selectors:
        - `document.getElementById(id)` - takes in an `id` string and returns the first HTML element with an `id` attribute that matches the given `id` string
            - ex: `document.getElementById('title')` will return an HTML DOM element with an `id` attribute with a value of "title"
        - `document.getElementsByTagName(tag)` - takes in a `tag` string and returns all the HTML elements in an `HTMLCollection` with the tagname matching the given `tag` string
            - ex: `document.getElementsByTagName('span')` will return an `HTMLCollection` containing all the HTML elements
        - `document.getElementsByClassName(class)` - takes in a `class` string and returns all the HTML elements in an `HTMLCollection` with the `class` attribute containing the value of the given `class` string
            - ex: `document.getElementsByClassName('yellow')` will return an `HTMLCollection` containing all the HTML elements with a class of "yellow"
        - `document.querySelector(selector)` - takes in a `selector` string and returns the first Node or HTML Element that matches the given CSS `selector`

        - ex: `document.querySelector('div.blue')` will return the first `div` HTML element with a class of "blue"
    - `document.querySelectorAll(selector)` - takes in a `selector` string and returns all the Nodes or HTML Elements in a `NodeList` that matches the given CSS `selector`
        - ex: `document.querySelectorAll('div.blue')` will return a `NodeList` containing all the `div` HTML elements with a class of "blue"

HTML Element

- **HTML element** - is a HTML DOM element with a tag
    - ex: if you have an HTML DOM element of `<h1>Hello World!</h1>`, `h1` is an HTML element.
- `document.createElement(tag)` - a method on the DOM or the `document` used to create an HTML element
    - returns a new HTML element with the given `tag` name
    - ex: `const h1 = document.createElement('h1')` creates an HTML element that looks like this: `<h1></h1>`
    - important note: it does not add it to the DOM or the HTML on the page
        - to add it to the DOM, you need to add it as a child element to any of the existing HTML elements on the DOM using the `appendChild()` method (see below for details about how to use this method)
- useful properties of an HTML element
    - `children` - returns all the HTML elements as an `HTMLCollection` that are the direct children of the given element
        - ex:

        - HTML: `<div>Hello <span>World!</span></div>`
        - element: `const div = document.querySelector('div')`
        - `div.children` would be an `HTMLCollection` that includes just the span element (`[span]`)
        - `Array.from(div.children)` allows you to turn the `HTMLCollection` into an array
- `childNodes` - returns all the Nodes as an `NodeList` that are the direct children of the given element
    - ex:
        - HTML: `<div>Hello <span>World!</span></div>`
        - element: `const div = document.querySelector('div')`
        - `div.childNodes` would be an `NodeList` that includes the text "Hello " and the span element (`[text, span]`)
        - `Array.from(div.childNodes)` allows you to turn the `NodeList` into an array
- `innerText` - used to read and set the text content of an element
    - ex:
        - HTML: `<p>Hello World!</p>`
        - element: `const p = document.querySelector('p')`
        - `p.innerText` would be "Hello World!"
        - to change the text inside of the p element to "One Step", set the property of `innerText` on the p element like so: `p.innerText = "OneStep"`
- `innerHTML` - used to read and set the HTML of an element
    - important note: best practice not to use it as it can create HTML elements that you may not want when showing data from a server or a source created from an application that isn't your own

        - ex:
            - HTML: `<p>Hello</p>`
            - element: `const p = document.querySelector('p')`
            - `p.innerHTML` would be "Hello"
            - to change the HTML inside of the `p` element and create a `span` element with the text of "World!" after "Hello" (`Hello <span>World!</span>`), set the property of `innerHTML` on the `p` element like so: `p.innerHTML = "Hello <span>World!</span>"`
    - `style` - an object that represents the inline-styling for the element
        - important note: inline-styling is the MOST-SPECIFIC CSS rule (will override even `id` rules)
        - each `key` on the object is by default an empty string "" and represents a CSS property
        - ex:
            - HTML: `<p>Hello World!</p>`
            - element: `const p = document.querySelector('p')`
            - `p.style.backgroundColor` would be ""
            - to change the background-color CSS property of the `p` element set the property of `backgroundColor` to "green" on the `p` element's `style` object like so: `p.style.backgroundColor = "green"`. This will result in changing the actual HTML for the `p` element to be: `<p style="background-color: green;">Hello World!<p>`
- useful methods of an HTML element:
    - `getAttribute(name)` - reads the attribute value of the given attribute `name` on the given element
        - ex:

        - HTML: `<div class="yellow">Hello World!</div>`
        - element: `const div = document.querySelector('div')`
        - `div.getAttribute('class')` would return "yellow"
    - `setAttribute(name, value)` - sets the attribute `value` of the given attribute `name` on the given element
        - ex:
            - HTML: `<div>Hello World!</div>`
            - element: `const div = document.querySelector('div')`
            - `div.setAttribute('class', 'yellow')` would apply a class name of "yellow" to the `div` element
    - `appendChild(node)` - adds the given `node` to the given element as the last child of the element
        - ex:
            - HTML: `<ul></ul>`
            - element: `const ul = document.querySelector('ul')`
            - node: `const li = document.createElement('li')`
            - `ul.appendChild(li)` would add the newly created `li` element to the existing `ul` element
    - `remove()` - removes the given element from the DOM completely
        - ex:
            - HTML: `<div>Hello <span>World!</span></div>`
            - element: `const span = document.querySelector('span')`
            - `span.remove()` would remove the `span` element from the DOM so the `div` element would look like this: `<div>Hello </div>`

Week 9 Day 3 Study Guide

Use this outline to guide you through today's readings and practices. This will focus your attention on the most important points to prepare you for the weekly assessment.

Event Handling

Learning Objectives:

- Be very familiar with these common event listeners: `click`, `submit`, `change`, and `DOMContentLoaded`
- Research to discover and utilize a new event listener to accomplish a given task
- Add and/or remove an event listener from one or more HTML elements
- Manipulate the DOM as a response to an event
- Diagram how an event propagates
- Predict and prevent the default behavior of an event
- Add, remove, and read data on an HTML element

What are Events?

- **Event** - an action that happens on an HTML DOM element
    - common events that you need to know well for the assessment:
        - `click` - when the mouse presses and releases on an *HTML element*
            - MDN reference

        - ex: a `click` event on a `button` element
    - `submit` - when an *HTML `form` element* is submitted
        - MDN reference
        - ex: the `submit` event on a `form` element
    - `change` - when the value of an *HTML `input/select/textarea` element* is committed as a change
        - MDN reference
        - ex: the user selects a different option on a `select` element
        - to check the `value` of an HTML `input/select/textarea` element in JavaScript, use the `value` property on the HTML element:
            - ex: if you type "Hello World!" into an input of type "text", `input.value` would be "Hello World!"
    - `input` - whenever an *HTML `input/select/textarea` element*'s value changes
        - MDN reference
        - ex: the user types in a value in an `input` element
    - `DOMContentLoaded` - when the HTML DOM is fully loaded without waiting for stylesheets, images, and other assets to finish loading
        - MDN reference
        - `DOMContentLoaded` event is an action that can only be done on the `window` object (BOM) or the `document` object (DOM)
        - **NOTE**: this is different from the `window.onload` function because the `window.onload` gets called when the HTML DOM AND all assets are finished loading. It's more common to use

the `DOMContentLoaded` event when running JS files in the frontend that read/manipulate HTML DOM elements.

- there are a ton other events that you don't need to know how to use exactly, but you should be able to use the MDN documentation for any event type and figure out how to use it
    - all other event types: MDN Events
- **Event Capturing and Bubbling** - the order which the event handlers get called for an action
    - You can add an event handler that will be triggered for a specific action and let the browser know in what order it should be triggered
    - There are two phases which you can attach the event handler to are the capturing and bubbling phases
        - the default phase, if no phase is specified when attaching the event listener, is the bubbling phase
        - the capturing phase will happen first when the action is triggered, then the bubbling phase
        - **you don't need to know about the capturing phase for the assessment!**
    - **Event Propagation** - the order in which the event handlers on each element gets triggered in each phase:
        - In the capturing phase:
            - The browser checks to see if the element's outer-most ancestor `<html>` has a capturing event handler registered on it for the triggered action and runs it if so
            - Then it moves on to the next element inside `<html>` and does the same thing,

then the next one, and so on until it reaches the element that was actually selected
        - In the bubbling phase, the exact opposite occurs:
            - The browser checks to see if the element selected has a bubbling event handler registered on it for the triggered action and runs it if so
            - Then it moves on to the next immediate ancestor element and does the same thing, then the next one, and so on until it reaches the `<html>` element
        - to stop the propagation of the event to the next element, you can call `event.stopPropagation()` on the `event` object
            - See MDN docs on Event.stopPropagation()
- **Default Event Behavior** - default actions on an event
    - depends on the event itself
    - examples of default event behavior:
        - toggling a checkbox is the default action of clicking on a checkbox
        - loading a new page is the default action of submitting a `form`
        - automatically change the value of the text `input` whenever the user types something inside of it is the default text `input` action
    - to prevent the default action from happening, call `.preventDefault()` on the `event` object
        - ex: `event.preventDefault()`
    - See MDN docs on Event.preventDefault()

    - So how do you programmatically execute code whenever an event on an HTML element happens or gets triggered? By subscribing to the particular event on a specific HTML element using an event listener
- **Event Listener** - allows you to execute code whenever a particular event on a specific HTML element happens or gets triggered
    - like a subscription to an event
    - `element.addEventListener(eventType, callbackFunction)` - execute the `callbackFunction` when an event of the type of `eventType` happens to the specific HTML element, `element`, that you call the `.addEventListener()` method on
        - `callbackFunction(event)` will be invoked or called with an `event` object whenever the event gets triggered
            - `event` object - contains information about the elements that were involved in the event and some properties and methods that you can use to influence what happens next
            - useful properties on the `event` object:
                - `event.target` - the HTML element that triggered the event listener
                - `event.currentTarget` - the HTML element that you attached the event listener to
                - `event.defaultPrevented` - a boolean representing if the default behavior of the event has been prevented or not
            - useful methods on the `event` object:
                - `event.preventDefault()` - prevents the default behavior of the event to happen
                - `event.stopPropagation()` - prevents the event from bubbling or triggering event listeners on HTML elements higher than the HTML element that

you attached the event listener to in the HTML DOM tree
- `element.removeEventListener(eventType, callbackFunction)` - remove the event listener set by a previous `.addEventListener()` call for the event type of `eventType` on a specific HTML element, `element`, and a reference to the original function that you called `.addEventListener()` with

Examples of using event listeners:

To print "Hello World!" to the console when the HTML DOM is fully loaded, you would add an event listener to the `window` for the `eventType` of `DOMContentLoaded` and in the `callbackFunction`, print "Hello World!":

```
window.addEventListener('DOMContentLoaded', () =>
console.log("Hello World!"));
// can also do this:
// document.addEventListener('DOMContentLoaded', () =>
console.log("Hello World!"));
```

To print the value of a form `<input type="text">` when a user types a character in the input, you would add an event listener to the `<input>` element for the `eventType` of `input` and in the `callbackFunction`, print the value of the `<input>` using the information about the `<input>` element in the `event` object:

```
const textInput = document.querySelector('input[type="text"]')
textInput.addEventListener('input', (event) => {
  console.log(event.currentTarget.value));
});
```

Read/Update `data-` attributes

`data-` attributes are really useful for storing data that doesn't need to be displayed on HTML elements.

Given the following HTML:

```
<div data-banana="yellow"></div>
```

You can read and update `data-` attributes on an HTML element:

```
const div = document.querySelector('div');
div.dataset.banana; // "yellow"
div.dataset.coolInfo = "Hello World!";
```

The above code will change the HTML to be:

```
<div data-banana="yellow" data-cool-info="Hello World!"></div>
```

Week 9 Day 4 Study Guide

Use this outline to guide you through today's readings and practices. This will focus your attention on the most important points to prepare you for the weekly assessment.

Browser Storage

Learning Objectives:

- Select a strategy for storing data in the client (browser)
- Identify common use cases for storing data in the client
- Compare and contrast `localStorage`, `sessionStorage`, and cookies
- Identify, examine, and delete data storage on the browser using Chrome Developer Tools
- Construct JavaScript to add, modify, remove, and read data using the Web Storage API
- Construct JavaScript to add, modify, remove, and read data using cookies

Storing Data in the Client

- types of browser storage:
    - Web Storage API - will **not** be sent back and forth with every request/response to and from the server, can only be read/set on the client, easier to read and write than cookies
        - `localStorage` - or `window.localStorage`, all data will be persist until deleted by the user or the application, no expiration date

        - to read a key-value pair in `localStorage`: `localStorage.getItem(key)` => value
        - to write a key-value pair in `localStorage`: `localStorage.setItem(key, value)`
        - to remove a key-value pair in `localStorage`: `localStorage.removeItem(key, value)`
    - `sessionStorage` - or `window.sessionStorage`, all data will be automatically be deleted when the browser is closed (session is over)
        - to read a key-value pair in `sessionStorage`: `sessionStorage.getItem(key)` => value
        - to write a key-value pair in `sessionStorage`: `sessionStorage.setItem(key, value)`
        - to remove a key-value pair in `sessionStorage`: `sessionStorage.removeItem(key, value)`
- Cookie - will be sent back and forth with every request/response to and from the server, can be read/set on both the client and the server, harder to read and write than Web Storage API, can only hold 4kB of data
    - **session cookie** - a key-value paired saved as a cookie that will automatically be deleted when the browser is closed (session is over)
        - to write a session cookie on the client: `document.cookie = "cookieName=cookieValue"`
            - ex: to add a `greeting` cookie with a value of "Hello World!" that will expire after the browser is closed: `document.cookie = "greeting=Hello World!"`

- **persistent cookie** - persists until an expiration date or age
    - to write a persistent cookie, you need to set a maximum age or expire date: `document.cookie = "cookieName=cookieValue; max-age=" + numSeconds` OR `document.cookie = "cookieName=cookieValue; expires=" + dateTimeInUTC`
        - ex: to add a `greeting` cookie with a value of "Hello World!" that will expire in 1 min: `document.cookie = "greeting=Hello World!; max-age=60"` OR `document.cookie = "greeting=Hello World!; expires=" + new Date(Date.now() + 60 * 1000).toUTCString()`
- to remove a cookie, simply set the maximum age to 0 or the expiration date to a date in the past
    - ex: to remove the `greeting` cookie: `document.cookie = "greeting=; max-age=0"` OR `document.cookie = "greeting=; expires=" + new Date('June 10, 1960')`
- `document.cookie` returns all the cookies in key-value pairs like so:
    - `console.log(document.cookie)` could print: `key1=value1; key2=value2; key3=value3;`
    - it does not show any other cookie information like the expire date or the maximum age
    - to read a cookie value, you need to parse the output of `document.cookie` by separating the key-value pairs, finding the key that you want to read, and then getting the value from that

- should be able to make an educated guess about when to use the following data storage options
    - `localStorage` (client-side)
        - easy to read and write
        - can store a decent amount of data
        - will persist until deleted
        - can only be read and updated by the client not the server
        - cannot be read by the server unless data is sent directly in the request
    - `sessionStorage` (client-side)
        - easy to read and write
        - can store a decent amount of data
        - will be automatically deleted when browser closes
        - can only be read and updated by the client not the server
        - cannot be read by the server unless data is sent directly in the request
    - session cookies (client-side)
        - hard to read and write on the client
        - total cookie storage is only 4kB of data
        - will be automatically deleted when browser closes
        - will be sent with every request to the server
        - can be updated by the server through any responses that come back from the server
    - persistent cookies (client-side)
        - hard to read and write on the client
        - total cookie storage is only 4kB of data
        - will persist until expire date passes or the maximum age in seconds is reached
        - will be sent with every request to the server

        - can be updated by the server through any responses that come back from the server
    - backend database (server-side)
        - can hold a lot more data
        - data can be accessed by all clients
        - persists until deleted by the server
        - data must be requested by the client and sent as a response by the server
        - can restrict information to be sent to the client

- use the "Application" tab in the Developer Tools to read/manipulate the different client storage options

Week 8 Day 5 Study Guide

Use this outline to guide you through today's readings and practices. This will focus your attention on the most important points to prepare you for the weekly assessment.

Network

Learning Objectives:

- Compare and contrast a MAC Address, an IP Address, and a port
- Compare and contrast IP Addresses, Domain Names, and DNS
- Diagram the process of sending data from a client to a server and back

What is a MAC Address?

- **MAC Address** - Media Access Control Address, permanent identifiers assigned to network interface hardware
    - hardcoded into the device, so can't be changed without physically changing the hardware
    - the only protocol addressing scheme that is considered "permanent"
    - used to differentiate devices & interface from each other
    - **MAC filtering** is used to limit access in corporate computer networks or on multiplayer gaming services
        - you should never rely on MAC filtering as a surefire security feature

        - you can easily change what MAC address is reported by an operating system, this is called **spoofing**
    - referred to as a physical address
    - represented by 6 pairs of hexadecimal digits
        - ex: `A1-B2-C3-D4-E5-F6` or `a1b2c3d4e5f6`
    - used by network devices to map network interfaces to one another
- **frame** - external wrapper for data transmitted through a network
    - data is passed in frames
    - each frame includes:
        - a source MAC address
        - a destination MAC address
        - a payload containing the transport protocol wrapper (TCP vs. UDP)
        - IP wrapper (IP Address)
        - and any additional application data

What is a Port?

- **port** - a virtual interface, acts as a connection point for a particular service or application to the network
    - each transport protocol (TCP, UDP) has different set of ports and port numbers
    - defined by a number from 0 - 65535 and its transport protocol
        - To distinguish a port between a TCP or a UDP port, you specify the protocol followed by a port number
        - ex: you have used port `TCP 5000` for development, but not `UDP 5000` yet
        - ex: `TCP 25` and `UDP 25` are different ports even though they have the same port number

    - when we talk about port numbers, we usually mean TCP ports
    - similar to a gate at an airport terminal
    - TCP port assignments today are managed by the IANA and are broken down into three separate ranges:
        - TCP 0-1023 are System ports
            - reserved for well-known services and use cases
        - TCP 1024-49151 are User ports
            - used to be reserved for services that identified themselves to the IANA
            - now used for any custom software
            - 3000, 5000, and 8080 are reserved for development
        - TCP 49152-65535 are Dynamic ports
            - ephemeral or short-living
            - a system service is likely to use these when building sockets for TCP connections
    - Treat any ports below 1024 as untouchable for your own custom servers, and only work above 49152 if you have a very good reason to do so

What is DNS?

- **DNS** - Domain Name System, a method of translating long numeric identifiers into friendly, human-readable addresses
    - similar to a phone book where a phone book translates phone numbers into people/businesses and home/business addresses
    - ex: Google Public DNS IP addresses (IPv4) are `8.8.8.8` and `8.8.4.4`

- **domain registry** - an organization that holds records of domain names to their numeric identifier
    - kind of like the creator of a phone book
    - these organizations must be registered by ICANN
- **domain name** - the "friendly" name for the website's host, or the server providing the site's content
    - differs from a URL in that the domain is only the server's identifier, not other application or protocol-related data in the URL
    - ex: `students.appacademy.io`
    - split into three sections
        - **top-level domain** or TLD - last part of the domain name, to the right of the last dot in the domain name
            - ex: domain name: `students.appacademy.io`, top-level doman: `.io`
            - common TLD's include `.com`, `.net` and `.org`
            - handled by a single domain registry
                - ex: `.gov` is managed by the General Services Agency
        - **second-level domain** - the left of the last dot in the domain name
            - can be purchased from a **domain register** - sells second-level domains
            - ex: domain name: `students.appacademy.io`, second-level doman: `appacademy`
        - other level domains - to the left of the second-level domain, don't need to be purchased after domain is purchased
            - ex: domain name: `students.appacademy.io`, third-level doman: `students`

    - the domain usually means just the second-level and top-level domains

Describe Sending a Request using Network Terminology

Deployment

Learning Objectives

- Deploy an application on Netlify and use your own domain name

Use a Custom Domain Name by Registering a DNS Name

Follow the instructions in the practice to create your own custom domain name to use for the project you deployed to Netlify on Week 7!