JavaScript

- It is useful to distinguish between code that is run by the client, the user interacting with a web application, and the server, which is the Flask application running the website. The client makes an HTTP request to the server, which is a running some Python code. The server processes the response to understand what the client is asking for, and ultimately sends back some HTML and CSS content which is rendered in the client's browser. It is often useful, however, to have code that does run client-side. Client-side processes reduce load on the server and are often faster.

Using JavaScript with HTML and CSS

- JavaScript is a programming language designed to be run inside a web browser that is run client-side. There are many different versions of JavaScript that are supported by different browsers, but there are certain standard versions that are supported by most. In this class, one of the more popular, recent versions, ES6, will be used.
- When embedded directly inside the HTML code for a webpage, it is enclosed in `<script></script>` tags.

```
<script>
    alert('Hello, world!');
</script>
```

- The previous code example, if placed in the `head` element, for example, would run as soon as the page is loaded. JavaScript can also be run in response to events.

```
<html>
    <head>
        <script>
            function hello() {
                alert('Hello!');
            }
        </script>
        <title>My Website</title>
    </head>
    <body>
        <h1>Welcome!</h1>
        <button onclick="hello()">Click Here!</button>
    </body>
</html>
```

  - Now, the JavaScript code is contained inside a function. Note that the function is delimited by curly braces.
  - The function `hello` is never called inside the `script` element. Rather, there is a `button` element with the `onclick` attribute which has the `hello` function as its value. The clicking of a button is one events that JavaScript understands which can be used as a trigger. In this case, that trigger runs the `hello` function.
- Some other JavaScript events include:
  - `onmouseover` : triggers when an element is hovered over
  - `onkeydown` : triggers when a key is pressed
  - `onkeyup` : triggers when a key is released
  - `onload` : triggers when a page is loaded
  - `onblur` : triggers when an object loses focus (when moving away from an input box, for example)

Manipulating the DOM

- Beyond just displaying alerts, JavaScript has the power to actually change the contents of a webpage.

```
<html>
    <head>
        <script>
            // Function to change heading to say goodbye
            function hello() {
                document.querySelector('h1').innerHTML = 'Goodbye!';
            }
        </script>
    </head>
    <body>
        <h1>Welcome!</h1>
        <button onclick="hello()">Click Here!</button>
    </body>
</html>
```

  - `document` refers to the web page currently being displayed.
  - `querySelector('tag')` is a function that searches through the webpage for a particular CSS selector and returns that element. If there are multiple results, only the first result is returned.
    - This function can also be called as `document.querySelector('#id')` and `document.querySelector('.class')`. More sophisticated selectors, selecting only descendants of certain element,s for example, can also be used.
  - The `innerHTML` property of an element is the HTML content contained within the element's tags.
  - When the `button` is clicked, the text `Welcome!` changes to `Goodbye!`.
- This slightly more advanced example showcases the use of variables in JavaScript.

```
<html>
    <head>
        <script>
            let counter = 0;

            function count() {
                counter++;
                document.querySelector('#counter').innerHTML = counter;
```

```
            }
        </script>
    </head>
    <body>
        <h1 id="counter">0</h1>
        <button onclick="count()">Click Here!</button>
    </body>
</html>
```

  - `let` is a keyword used to define variables.
  - `counter++` is a shorthand to increment `counter` by 1.
- Conditional statements in JavaScript look like this:

```
<script>
    let counter = 0;

    function count() {
        counter++;
        document.querySelector('#counter').innerHTML = counter;

        if (counter % 10 === 0) {
            alert(`Counter is at ${counter}!`);
        }
    }
</script>
```

  - `%` is the modulus operator, which returns the remainder of the first number divided by the second.
  - `===` checks for exact equality in JavaScript; two things must be identical for it to return true.
  - The argument to `alert` is a template literal, which is like a Python format string. `${counter}` is replaced with whatever the value of the variable `counter` is. Backticks are used to delimit a template literal.
- JavaScript can also be factored out of the HTML code entirely.

```
<html>
    <head>
        <script>
            document.addEventListener('DOMContentLoaded', function() {
                document.querySelector('button').onclick = count;
            });

            let counter = 0;

            function count() {
                counter++;
                document.querySelector('#counter').innerHTML = counter;

                if (counter % 10 === 0) {
                    alert(`Counter is at ${counter}!`);
                }
            }
        </script>
    </head>
    <body>
        <h1 id="counter">0</h1>
        <button>Click Here!</button>
    </body>
</html>
```

  - Note that there is no `onclick` attribute in the HMTL tags for the `button` element. Nonetheless, the function `addEventListener`, which, likes it name suggests, 'listens', or waits, until the event `DOMContentLoaded` occurs. This event occurs when the entire HTML structure is loaded by the browser. Then, the second argument, a function, is called.
  - JavaScript makes use of 'higher order functions', which means functions can be passed around like any other value. The function being passed is called a 'callback' function. The callback is called when the event being listened for occurs. In this case, that callback sets the `onclick` property of the `button` element to the `count` function, ultimately resulting in the same functionality as before.
- Going one step further, the JavaScript code can be factored out of the `.html` file entirely into a separate `.js` file. Everything that's inside the `script` element from the last example would go into the `.js` file, and the `.html` would look like this:

```
<html>
    <head>
        <script src="counter3.js"></script>
    </head>
    <body>
        <h1 id="counter">0</h1>
        <button>Click Here!</button>
    </body>
</html>
```

  - This is exactly the same paradigm that was seen in factoring out CSS.

Variables

- There are three main keywords used to define variables in JavaScript.
  - `const` : defines a constant variable that cannot be redefined later
  - `let` : defines a variable is local to the scope of the innermost pair of curly braces surrounding it
  - `var` : defines a variable that is local to the function it is defined in
- Here is an example showcasing these different ways to define variables:

```
<script>
    // This variable exists even outside the loop
    if (true) {
```

```
        var message = 'Hello!';
    }

    alert(message);
</script>
```

- Because `var` was used to define `message`, there will be no errors running this code.

```
<script>
// This variable does not exist outside the loop
if (true) {
    let message = 'Hello!';
}

alert(message);
</script>
```

- Because `let` was used to define `message`, it cannot be passed to `alert`, which is outside the scope of `message`. If this were in an HTML page, when the page was opened, no alert would pop up. If the console were opened in the browser, there would be an `Uncaught ReferenceError`.

```
<script>
// The value of const variables cannot change
const message = 'Hello!';
message = 'Goodbye!';

alert(message);
</script>
```

- Similar to the last example, no alert will pop up. In the console, there would be an `Uncaught TypeError`, since there was an attempt to redefine a variable defined with `const`.

- The JavaScript console, accessable in the **Develop** or **Inspect** menu in a web browser, allows for the interactive entry of JavaScript, similiar to the Python console.
- Here's another example which uses JavaScript to read info from a form.

```
<html>
    <head>
        <script>
            document.addEventListener('DOMContentLoaded', function() {
                document.querySelector('#form').onsubmit = function() {
                    const name = document.querySelector('#name').value;
                    alert(`Hello ${name}!`);
                };
            });
        </script>
    </head>
    <body>
        <form id="form">
            <input id="name" autocomplete="off" autofocus placeholder="Name" type="text">
            <input type="submit">
        </form>
    </body>
</html>
```

- The callback function here selects the element with the id `form` and sets its `onsubmit` (another event) property to another callback function which sets the `const` variable `name` to the `value` property returned from the elemnt with id `name`. `name` is the input box of a form, so `value` will be whatever the user has entered into the form.
- So, this code produces an alert that says hello to whatever name the user entered into the form.

Modifying Style

- JavaScript can also modify the CSS properties of elements.

```
<html>
    <head>
        <script>
            document.addEventListener('DOMContentLoaded', function() {

                // Change font color to red
                document.querySelector('#red').onclick = function() {
                    document.querySelector('#hello').style.color = 'red';
                };

                // Change font color to blue
                document.querySelector('#blue').onclick = function() {
                    document.querySelector('#hello').style.color = 'blue';
                };

                // Change font color to green
                document.querySelector('#green').onclick = function() {
                    document.querySelector('#hello').style.color = 'green';
                };
            });
        </script>
    </head>
    <body>
        <h1 id="hello">Hello!</h1>
        <button id="red">Red</button>
        <button id="blue">Blue</button>
        <button id="green">Green</button>
```

```
    </body>
</html>
```

  - There are three buttons, each of which (after the initial callback from loading the webpage) have their `onclick` properties set to a function which sets the `style.color` property of the `hello` element to a different color. Any CSS property could be modified, e.g. `style.background-color`, `style.margin`, etc.
- The repetitiveness of the last example can be reduced.

```
<html>
    <head>
        <script>
            document.addEventListener('DOMContentLoaded', function() {

                // Have each button change the color of the heading
                document.querySelectorAll('.color-change').forEach(function(button) {
                    button.onclick = function() {
                        document.querySelector('#hello').style.color = button.dataset.color;
                    };
                });

            });
        </script>
    </head>
    <body>
        <h1 id="hello">Hello!</h1>
        <button class="color-change" data-color="red">Red</button>
        <button class="color-change" data-color="blue">Blue</button>
        <button class="color-change" data-color="green">Green</button>
    </body>
</html>
```

  - `document.querySelectorAll('.color-change')` returns an array of all elements of the class `color-change`.
  - `forEach` is a built-in JavaScript function that can be called on an array that runs a function passed to it on each element of an array. The function being passed takes as an argument one particular element of the array.
  - Having all three buttons with the same class, `color-change`, allows for them to be selected together with `querySelectorAll`.
  - `data-color` is a data attribute. Data attributes allow for the association of additional information with an element without changing how the element is rendered by the browser. Data attributes can have any name as long as they start with `data-`.
  - Data atrributes can be accessed in the `dataset` property of an element. In this example, `data-color` is accessed in `dataset.color`.

Arrow Functions

- Since functions, especially anonymous functions, are so common in JavaScript, ES6 has introduced a new syntax for functions called arrow notation that allows for the definition of so-called arrow functions.

```
() => {
    alert('Hello, world!');
}

x => {
    alert(x);
}

x => x * 2;
```

  - An arrow function is defined without using the word `function`, but rather just with a pair of parentheses enclosing any arguments the function takes, followed by an arrow, and finally the function body, enclosed in curly braces.
  - Functions with only one argument can be defined without the use of parentheses enclosing the argument list.
  - Functions that have only one line in the body can drop the curly braces and have the body on the same line as the argument list and arrow.
- The previous example could be rewritten more succintly with arrow functions.

```
document.addEventListener('DOMContentLoaded', () => {

    // Have each button change the color of the heading
    document.querySelectorAll('.color-change').forEach(button => {
        button.onclick = () => {
            document.querySelector('#hello').style.color = button.dataset.color;
        };
    });

});
```

- One last variation of this color example could use a drop-down menu to select colors instead of buttons.

```
<html>
    <head>
        <script>
            document.addEventListener('DOMContentLoaded', () => {

                // Change the color of the heading when dropdown changes
                document.querySelector('#color-change').onchange = function() {
                    document.querySelector('#hello').style.color = this.value;
                };

            });
        </script>
    </head>
    <body>
        <h1 id="hello">Hello!</h1>
        <select id="color-change">
```

```
            <option value="black">Black</option>
            <option value="red">Red</option>
            <option value="blue">Blue</option>
            <option value="green">Green</option>
        </select>
    </body>
</html>
```

- `onchange` is the event fired when the selection in a drop-down menu is changed.
- `this` refers to whatever value the function is operating on, which in this case is `document.querySelector('#color-change')`, which is the drop-down menu itself. The selected item is extracted using the `value` attribute of the drop-down menu, which corresponds to one of the color options.
  - Note that using `this` with arrow functions will produce different behavior. `this` inside an arrow function will be bound to whatever value `this` would have taken on inside the code that is enclosing the arrow function. By writing out `function ()`, then `this` takes on the value of whatever the function is being called on.

More with JavaScript

- In the next example, the goal will to be create a to-do list application. Here's the starting point:

```
<html>
    <head>
        <script>
            document.addEventListener('DOMContentLoaded', () => {

                document.querySelector('#new-task').onsubmit = () => {

                    // Create new item for list
                    const li = document.createElement('li');
                    li.innerHTML = document.querySelector('#task').value;

                    // Add new item to task list
                    document.querySelector('#tasks').append(li);

                    // Clear input field
                    document.querySelector('#task').value = '';

                    // Stop form from submitting
                    return false;
                };

            });
        </script>
        <title>Tasks</title>
    </head>
    <body>
        <h1>Tasks</h1>
        <ul id="tasks">
        </ul>
        <form id="new-task">
            <input id="task" autocomplete="off" autofocus placeholder="New Task" type="text">
            <input type="submit">
        </form>
    </body>
</html>
```

  - The `tasks` unordered list starts empty, but will be populated with user input.
  - In the JavaScript code, when the form is submitted, a new `li` element is assigned to the `const` variable `li` using the function `document.createElement('li')`. Then, the `innerHTML` of that `li` is set to be whatever the `value` of the `task` input field is.
  - The new `li` is then added to the `ul` `tasks` with the `append(li)` function, called on the `ul`.
  - Finally, the input field is cleared and the default behavior for a form, which is to go to some other website and reload the page, is suppressed by returning `false`.
- Blank submissions can be omitted by conditionally enabling the submit button.

```
// By default, submit button is disabled
document.querySelector('#submit').disabled = true;

// Enable button only if there is text in the input field
document.querySelector('#task').onkeyup = () => {
    document.querySelector('#submit').disabled = false;

// ...same code as before...

// Disable button again after submit
document.querySelector('#submit').disabled = true;

// Stop form from submitting
return false;
};
```

  - This results in the button only being pressable once some keypress has been registered, assuming that the field is then populated.
- The previous implementation would still allow for submission if text was entered and then erased from the form. This can be remedied by checking that the `length` property of the `value` attribute of the form input is indeed greater than 0 after every keystroke.

```
// Enable button only if there is text in the input field
document.querySelector('#task').onkeyup = () => {
    if (document.querySelector('#task').value.length > 0)
        document.querySelector('#submit').disabled = false;
    else
```

```
        document.querySelector('#submit').disabled = true;
};
```

- Another feature of JavaScript is the ability to wait for a certain amount of time.

```
<html>
    <head>
        <script>
            document.addEventListener('DOMContentLoaded', () => {
                setInterval(count, 1000);
            });

            let counter = 0;

            function count() {
                counter++;
                document.querySelector('#counter').innerHTML = counter;
            }
        </script>
    </head>
    <body>
        <h1 id="counter">0</h1>
    </body>
</html>
```

  - The `setInterval` function takes another function and then the interval (in milliseconds), after which the passed function will be automatically called over and over.
  - The result, in this example, is an automatically incrementing counter without the need for any buttons.
- If the previous example were reloaded, the counter would be reset. The maintain some persistence, JavaScript can use local storage to keep track of some state information.

```
<html>
    <head>
        <script>
            // Set starting value of counter to 0
            if (!localStorage.getItem('counter'))
                localStorage.setItem('counter', 0);

            // Load current value of  counter
            document.addEventListener('DOMContentLoaded', () => {
                document.querySelector('#counter').innerHTML = localStorage.getItem('counter');

                // Count every time button is clicked
                document.querySelector('button').onclick = () => {
                    // Increment current counter
                    let counter = localStorage.getItem('counter');
                    counter++;

                    // Update counter
                    document.querySelector('#counter').innerHTML = counter;
                    localStorage.setItem('counter', counter);
                }
            });
        </script>
    </head>
    <body>
        <h1 id="counter"></h1>
        <button>Click Here!</button>
    </body>
</html>
```

  - `localStorage` is the variable that JavaScript can store information at. `getItem` and `setItem` can be called on `localStorage` to either load or save data. This example first tries to load `counter`, and if it's not there, saves a new `counter` with value `0`.
  - Then, the `counter` element is initially set to that `counter` item in storage. After that, a variable called `counter` is used to reference the `counter` item, and after every update of the `counter` variable, the `counter` item in `localStorage` has its value updated.
  - Now, closing and reloading the page will not reset the value of the counter.

Integrating JavaScript with Python and Flask

- Ajax, is used to get more information from server without needing to reload an entirely new page. As an example, Ajax can be used with the currency conversion example from last week to display a conversion without needing to load a new page. This is not done by pre-loading all possible exchange rates, but by making an Ajax request to the Flask server, which will get a particular exchange rate whenever it is asked for. JavaScript can then be used to update the DOM to render the new content.
- Here's the interesting part of `application.py`. There's not much different here from last week, but note that what's being returned is not a new webpage, but rather just a JSON object.

```
@app.route("/convert", methods=["POST"])
def convert():

    # Query for currency exchange rate
    currency = request.form.get("currency")
    res = requests.get("https://api.fixer.io/latest", params={
        "base": "USD", "symbols": currency})

    # Make sure request succeeded
    if res.status_code != 200:
        return jsonify({"success": False})

    # Make sure currency is in response
```

```
    data = res.json()
    if currency not in data["rates"]:
        return jsonify({"success": False})

    return jsonify({"success": True, "rate": data["rates"][currency]})
```

- The HTML is simply a basic form. The JavaScript code is in a different file, but linked in the `head`.

```
<html>
    <head>
        <script src=""></script>
        <title>Currency Converter</title>
    </head>
    <body>
        <form id="form">
            <input id="currency" autocomplete="off" autofocus placeholder="Currency" type="text">
            <input type="submit" value="Get Exchange Rate">
        </form>
        <br>
        <div id="result"></div>
    </body>
</html>
```

  - `url_for('static', filename='index.js')` is Flask's way of incorporating `.js` files. `static` is a separate folder.
  - The `result` `div` will contain the conversion, but is currently blank.
- The interesting code is inside of `index.js`.

```
document.addEventListener('DOMContentLoaded', () => {

    document.querySelector('#form').onsubmit = () => {

        // Initialize new request
        const request = new XMLHttpRequest();
        const currency = document.querySelector('#currency').value;
        request.open('POST', '/convert');

        // Callback function for when request completes
        request.onload = () => {

            // Extract JSON data from request
            const data = JSON.parse(request.responseText);

            // Update the result div
            if (data.success) {
                const contents = `1 USD is equal to ${data.rate} ${currency}.`
                document.querySelector('#result').innerHTML = contents;
            }
            else {
                document.querySelector('#result').innerHTML = 'There was an error.';
            }
        }

        // Add data to send with request
        const data = new FormData();
        data.append('currency', currency);

        // Send request
        request.send(data);
        return false;
    };

});
```

  - An `XMLHttpRequest` is just an object that will allow an Ajax request to be made.
  - `request.open` is where the new request is actually initialized, with the HTTP method and route being specified.
  - `JSON.parse` converts the raw response (`request.responseText`) into an object that can be indexed by keys and values.
  - The rest of the callback simply updates the HTML using template literals to reflect the result of the conversion.
  - `FormData` is just an object that holds whatever the user input is.

Websockets

- The request-response model, which has been the basis for how HTTP requests and client-server interaction has been discussed so far, is useful as long as data is only being passed when a request is made. But, with 'full-duplex communication', more simply described as real-time communication, there is (or shouldn't be) a need for reloading a webpage and making a new request just to check, for example, if someone sent a message in a chat room. Websockets are a protocol that allow for this type of communication, and Socket.IO is a particular JavaScript library that supports this protocol.
- This example will be based around a voting application that will count and display votes in real-time. Here's the full `application.py`, with all the setup and import statements.

```
import os
import requests

from flask import Flask, jsonify, render_template, request
from flask_socketio import SocketIO, emit

app = Flask(__name__)
app.config["SECRET_KEY"] = os.getenv("SECRET_KEY")
socketio = SocketIO(app)
```

```
@app.route("/")
def index():
    return render_template("index.html")

@socketio.on("submit vote")
def vote(data):
    selection = data["selection"]
    emit("announce vote", {"selection": selection}, broadcast=True)
```

  - `flask_socketio` is a library that allows for websockets inside a Flask application. This library allows for the web server and client to be emitting events to all other users, while also listening for and receiving events being broadcasted by others.
  - `submit vote` is an event that will be broadcasted whenever a vote is submitted. The code for this will be in JavaScript.
  - Once a vote is received, the vote is announced to all users (`broadcast=True`) with the `emit` function.
- `index.html`:

```
<html>
    <head>
        <script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/socket.io/1.3.6/socket.io.min.js"></script>
        <script src=""></script>
        <title>Vote</title>
    </head>
    <body>
        <ul id="votes">
        </ul>
        <hr>
        <button data-vote="yes">Yes</button>
        <button data-vote="no">No</button>
        <button data-vote="maybe">Maybe</button>
    </body>
</html>
```

- `index.js`:

```
document.addEventListener('DOMContentLoaded', () => {

    // Connect to websocket
    var socket = io.connect(location.protocol + '//' + document.domain + ':' + location.port);

    // When connected, configure buttons
    socket.on('connect', () => {

        // Each button should emit a "submit vote" event
        document.querySelectorAll('button').forEach(button => {
            button.onclick = () => {
                const selection = button.dataset.vote;
                socket.emit('submit vote', {'selection': selection});
            };
        });
    });

    // When a new vote is announced, add to the unordered list
    socket.on('announce vote', data => {
        const li = document.createElement('li');
        li.innerHTML = `Vote recorded: ${data.selection}`;
        document.querySelector('#votes').append(li);
    });
});
```

  - First, the websocket connection is established using a standard line to connect to wherever the application is currently running at.
  - `submit vote` is the name of the event that's being submitted on a button click. That event just sends whatever the vote was.
  - `announce vote` is an event received from the Python sever, which triggers the updating of the vote list.
- An improvement to this application would be to display a total vote count, instead of just listing every individual vote, and making sure that new users can see past votes.
- Changes to `application.py`:

```
votes = {"yes": 0, "no": 0, "maybe": 0}

@app.route("/")
def index():
    return render_template("index.html", votes=votes)

@socketio.on("submit vote")
def vote(data):
    selection = data["selection"]
    votes[selection] += 1
    emit("vote totals", votes, broadcast=True)
```

  - Now, any vote submissions are first used to update the `votes` dictionary to keep a record of vote totals. Then, that entire dictionary is broadcasted.
- Changes to `index.html`:

```
<body>
    <div>Yes Votes: <span id="yes"></span></div>
    <div>No Votes: <span id="no"></span></div>
    <div>Maybe Votes: <span id="maybe"><span></div>
    <hr>
    <button data-vote="yes">Yes</button>
    <button data-vote="no">No</button>
    <button data-vote="maybe">Maybe</button>
</body>
```

    - The `span` elements allocate a space for vote tallies to be filled in later.
- Changes to `index.js`:

```
socket.on('vote totals', data => {
    document.querySelector('#yes').innerHTML = data.yes;
    document.querySelector('#no').innerHTML = data.no;
    document.querySelector('#maybe').innerHTML = data.maybe;
});
```