HTTP Learning Objectives

HTTP stands for **HyperText Transfer Protocol.** Is a protocol for transmitting hypermedia documents, such as HTML. HTTP requests can have up to three parts: a request line, headers, and a body.

1. Match the header fields of HTTP with a bank of definitions

HTTP headers let the client and the server pass additional information with an HTTP request or response. Here are some common request headers you'll see:

- Host: specifies the domain name of the server.
- User-Agent: a string that identifies the operating system, software vendor or version of the requester.
- Referer: the address of the previous web page from which a link to the currently requested page was followed.
- Accept: informs the server about the types of data that can be sent back.
- Content-Type: Indicates the media type found in the body of the HTTP message.

2. Matching HTTP verbs (GET, PUT, PATCH, POST, DELETE) to their common uses

HTTP defines a set of request methods to indicate the desired action to be performed for a given resource.

- GET: a request to retrieve data. It will never have a body.
- POST: sends data to the server creating a new resource.
- PUT: updates a resource on the server.
- PATCH: similar to PUT, but it applies partial modifications to a resource.
- DELETE: deletes the specified resource.

3. Match common HTTP status codes (200, 302, 400, 401, 402, 403, 404, 500) to their meanings

HTTP response status codes indicate whether a specific HTTP request has been successfully completed.

- 200: OK. The request has succeeded.
- 302: Found. The URI of requested resource has been changed temporarily.
- 400: Bad Request. The server could not understand the request due to invalid syntax.
- 401: Unathorized. The client must authenticate itself to get the requested response.
- 402: Payment Required.
- 403: Forbidden. The client does not have access rights to the content.
- 404: Not Found. The server can not find the requested resource.
- 500: Internal Server Error. The range from 500-599 indicate server errors.

4. Send a simple HTTP request to google.com

netcat (nc) allows you to open a direct connection with a URL and manually send HTTP requests.

Request

```
nc -v google.com 80
GET / HTTP/1.1
```

Response

```
HTTP/1.1 200 OK
Date: Thu, 28 May 2020 20:50:17 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
<!doctype html>
<html>
</html>
```

5. Write a very simple HTTP server using ‘http’ in node with paths that will result in the common HTTP status codes

```
const http = require('http');

http.createServer(function(request, response) {
    if (request.url === '/') {
        response.writeHead(
            200,
            { 'Content-Type': 'text/html' }
        );
        response.write('<h1>OK</h1>');
        response.end();
    } else {
        response.writeHead(404);
        response.end();
    }
}).listen(8080, function() {
    console.log(
        'listening for requests on port 8080...'
    );
});
```

Promises Lesson Learning Objectives I

1. Instantiate a Promise object

```
const myPromise = new Promise((resolve, reject) => {
    try {
        // try some code, if it works, then we can call `resolve()`
        someAsynchronousFunctionThatMightFail(result => {
            // If the async function works, it'll call this callback
            // and pass us the result of whatever it did.
            resolve(result); // Then we can call resolve with the result.
        });
    }
    catch (error) {
        // if we get an error we can call `reject()` with the error
        reject(error);
    }
});
```

2. Use Promises to write more maintainable asynchronous code

Let's assume we want to wait 10 seconds, do a thing, then wait 10 more then do a different thing, then wait 30 seconds and do a final thing. We can use setTimeout but with callbacks it looks like this:

```
setTimeout(() => { // Look at all the deep nesting!
    doAThing();
    setTimeout(() => {
        doADifferentThing();
        setTimeout(() => {
            doAFinalThing();
        }, 30000)
    }, 10000)
}, 10000)
```

If we wrap setTimeout in a promise like this:

```
const sleep = (milliseconds) => {
    return new Promise((resolve) => {
        setTimeout(resolve, milliseconds);
    });
}
```

Then we can write code that looks like this using our `sleep` function.

```
sleep(1000)
    .then(() => {
        doAThing();  // We don't need to return a promise here `.then()`
                     // automatically returns one anyway.
    })
    .then(() => {
        return sleep(1000); // We have to return the promise we
                            // got from the sleep function
                            // then() is smart enough to know when
                            // we return a promise vs just returning a value
    })
    .then(() => {
        doADifferentThing();
    })
    .then(() => {
        return sleep(3000);
    })
    .then(() => {
        doAFinalThing();
    });
```

Using shortened arrow function syntax we can make this even shorter and make it look really synchronous, even though it's asynchronous.

```
sleep(1000)
.then(() => doAThing())
.then(() => sleep(1000))
.then(() => doADifferentThing())
.then(() => sleep(3000))
.then(() => doAFinalThing());
```

You can also use `Promise.all()` when you don't care about the order.

```
const fs = require('fs').promises // requires the promises version of fs

// Read in three files and concatenate them together.
Promise.all([
    fs.readFile("d1.md", "utf-8"),
    fs.readFile("d2.md", "utf-8"),
    fs.readFile("d3.md", "utf-8"),
])
.then((contents1, contents2, contents3) => { // Even though they run
                                             // asynchronously
                                             // in an indeterminate order,
                                             // Promise.all keeps them in the
                                             // right order in the arguments
                                             // list
    return contents1 + contents2 + contents3;
})
.then((concatted) => {
    console.log(concatted);
});
```

Imagine if we tried to do this without Promises:

```
const fs = require('fs');

fs.readFile("d1.md", "utf-8", contents1 => {
    fs.readFile("d1.md", "utf-8", contents2 => {
        fs.readFile("d3.md", "utf-8", contents3 => {
            console.log(contents1 + contents2 + contents3);
        });
    });
});
```

In fact this isn't doing the same thing at all! This is actually only reading the next file when the first one is completed. Promise.all() is probably faster since all the readFiles are kicked off at the same time.

3. Use the fetch API to make Promise-based API calls

```
const fetch = require('node-fetch'); // we need this to use fetch in node
                                     // It's built into the browser
fetch("https://ifconfig.me/all.json")
  .then((response) => {
    return response.json(); // the json() method returns a promise and is async
  })
  .then((data) => {
    console.log(data);
  });
```

Async and Await Learning Objectives

1. Use async/await with promise-based functions to write asynchronous code that behaves synchronously.

```
function slow() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve('That was slow.');
        }, 2000);
    });
}

function fast() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve('That was fast.');
        }, 1000);
    });
}

async function syncLike() {
    const slowVal = await slow();
    console.log(slowVal);
    const fastVal = await fast();
    console.log(fastVal);
}

syncLike(); // Prints 'that was slow.' after 2000ms, then 'that was fast.' after 1000ms
```

The real power comes in leveraging libraries that support promises like `fs` of `fetch`

Instead of:

```
const fs = require("fs");

function concatFiles() {
    fs.readFile("d1.md", "utf-8", (err, contents1) => {
        fs.readFile("d1.md", "utf-8", (err, contents2) => {
            fs.readFile("d3.md", "utf-8", (err, contents3) => {
            console.log(contents1 + contents2 + contents3);
            });
        });
    });
}

concatFiles();
```

We can do this:

```
const fs = require("fs").promises;

async function concatFiles() {
    const contents1 = await fs.readFile("d1.md", "utf-8");
    const contents2 = await fs.readFile("d2.md", "utf-8");
    const contents3 = await fs.readFile("d3.md", "utf-8");
    console.log(contents1 + contents2 + contents3);
}

concatFiles();
```

Or when using fetch, instead of this:

```
const fetch = require("node-fetch");

function getIpAddress(callback) {
    return new Promise((resolve) => {
        fetch("https://ifconfig.me/all.json")
        .then((response) => {
            return response.json();
        })
        .then((ipInfo) => {
            resolve(ipInfo);
        });
    }
}

getIpAddress().then(ipInfo => {
    console.log(ipInfo);
})
```

We can just write this:

```
const fetch = require('node-fetch');

async function getIpAddress() {
    const response = await fetch("https://ifconfig.me/all.json");
    const ipInfo = await response.json(); // Remember .json() returns a promise
    return ipInfo;
}

// Rememeber to use await, we must be inside an async function, so
// I just made an async IIFE.
(async() => {
    const ipInfo = await getIpAddress();
})()
```

Much better!

HTML Learning Objectives

1. You'll be able to create structurally and semantically valid HTML5 pages using the following elements: html, head, title, link, script, The six header tags, p, article, section, main, nav, header, footer, ul, ol, li, a, img, table, thead, tbody, tfoot, tr, th, td.

```
<!DOCTYPE html>
<html>
<head>
    <title>HTML Example</title>
    <link rel="stylesheet" href="style.css">
    <script async type="module" src="index.js"></script>
</head>
<body>
    <main>
        <h1>An HTML page example</h1>
        <p>
            This is a very basic HTML page. For more examples click
            <a href="https://open.appacademy.io/learn/js-py---apr-2020-online/week-6-apr-2020-online/brushing-up-on-your-ht
        </p>
    </main>
</body>
</html>
```

Testing Learning Objectives

1. Explain the "red-green-refactor" loop of test-driven development.

Red, Green, Refactor refers to the development loop at the heart of TDD.

- **RED:** We begin by writing a test (or tests) that speicify what we expect our code to do. We run the test, to see it fail, and in doing so we ensure that our test won't be a false positive.
- **GREEN:** We write the minimum amount of code to get our test to pass. This step may take just a few moments, or a longer time depending on the complexity of the task.
- **REFACTOR:** The big advantage of test driven development is that it means we always have a set of tests that cover our codebase. This means that we can safely make changes to our code, and as long as our tests still pass, we can be confident that the changes did not break any functionality. This gives us the confidence to be able to constantly refactor and simplify our code - we include a refactor step after each time we add a passing test, but it isn't always necessary to make changes.

2. Identify the definitions of SyntaxError, ReferenceError, and TypeError

- **SyntaxError:** These errors refer to problems with the *syntax* of our code, they usually refer to either missing or rogue characters that cause the compiler to be able to understand the code we are feeding it.
- **ReferenceError:** These errors refer to times in our code where we reference a variable that is *not* available in the current scope.
- **TypeError:** These errors refer to times in our code where we reference a variable of the *wrong type*. Modifying a value that cannot be changed, using a value in an inappropriate way, or an argument of an unexpected type being passed to a function, are all causes of TypeErrors.

3. Create, modify, and get to pass a suite of Mocha tests

A minimal example:

`test/reverse-string.spec.js`

```
const assert = require("assert");
const reverseString = require('../lib/reverse-string').reverseString;


describe("reverseString", () => {
    it("should reverse simple strings", () => {
        assert.equal(reverseString("fun"), "nuf");
    });
    it("should throw a TypeError if it doesn't receive a string", () => {
        assert.throws(() => reverseString(0));
    });
});
```

`lib/reverse-string.js`

```
const reverseString = (str) => {
    if (typeof str !== "string") {
        throw new TypeError("expecting a string arg");
    }
    return "nuf";
}

module.exports = {
    reverseString
}
```

This minimal example also shows the importance of thorough testing. Though all tests here pass, it would be trivial to come up with a case where our reverseString implementation doesn't do what we want. `reverseString('foo')` for example.

4. Use Chai to structure your tests using behavior-driven development principles

Chai gives us the human-readable verbage that have become popular in the BDD (Behavior Driven Developement) world:

> Chai has several interfaces that allow the developer to choose the most comfortable. The chain-capable BDD styles provide an expressive language & readable style, while the TDD assert style provides a more classical feel.

Should

```
chai.should();

foo.should.be.a('string');
foo.should.equal('bar');
foo.should.have.lengthOf(3);
tea.should.have.property('flavors')
  .with.lengthOf(3);
```

Visit Should Guide

Expect

```
var expect = chai.expect;

expect(foo).to.be.a('string');
expect(foo).to.equal('bar');
expect(foo).to.have.lengthOf(3);
expect(tea).to.have.property('flavors')
  .with.lengthOf(3);
```

Visit Expect Guide

Assert

```
var assert = chai.assert;

assert.typeOf(foo, 'string');
assert.equal(foo, 'bar');
assert.lengthOf(foo, 3)
assert.property(tea, 'flavors');
assert.lengthOf(tea.flavors, 3);
```

Visit Assert Guide

Both the `should` and `expect` style assertion terminology are considered BDD style, while the `assert` style expressions are a throwback to TDD style.

5. Use the pre- and post-test hooks provided by Mocha

Mocha provides four pre and post test hooks. These should be used to set up preconditions and clean up after your tests.

```
describe('hooks', function() {
  before(function() {
    // runs once before the first test in this block
  });

  after(function() {
    // runs once after the last test in this block
  });

  beforeEach(function() {
    // runs before each test in this block
  });

  afterEach(function() {
    // runs after each test in this block
  });

  // test cases
});
```

Hooks *run in the order they are defined*, as appropriate; all `before()` hooks run (once), then any `beforeEach()` hooks, tests, any `afterEach()` hooks, and finally `after()` hooks (once).

Learning Objectives - Big O analysis

Big O notation is ONLY concerned with performance relative to its input size.

Big O notation describes an algorithm's worst case. Big O describes how the runtime of an algorithm scales with the amount of data it has to work on

We can measure both time and space, but are mostly concerned with time (memory is cheap and abundant)

1. Order the common complexity classes according to their growth rate

Name	Big O Notation
Constant	O(1)
Logarithmic	O(log(n))
Linear	O(n)
Linear Logarithmic	O(n*log(n))
Polynomial	O(n^m)
Exponential	O(m^n)
Factorial	O(n!)

2. Identify the complexity classes of common sort methods

Algorithm	Runtime Complexity	Memory Efficiency
Bubble Sort	O(n^2)	O(1)
Selection Sort	O(n^2)	O(1)
Insertion Sort	O(n^2)	O(n)
Merge Sort	O(n * log(n))	O(n)
Quick Sort	O(n^2)	O(n)

3. Identify complexity classes of code

`O(1)` - Do a known number of things, these don't grow with input size.

```
const firstThing = li => { // n would be input size
  return li[0];
};
```

```
const threeHundredThousandTimesLog = name => {
  for (let i = 0; i < 300000; i++) {
    console.log(name);
  }
};
```

`O(log n)` - Typically "divide and conquer" type algorithms

```
const splitInHalf = n => {
  if (n <= 1) return n;

  return splitInHalf(n / 2);
}
```

`O(n)` - Where we do a fixed number of things per item in the input

```
const printAll = li => {
  li.forEach(ele => {
    console.log(ele);
  })
};
```

```
const find = (li, value) => {
  for (let i = 0; i < li.length; i++) {
    if (li[i] === value) return true;
  }

  return false
}
```

```
const printALot = (li) => {
  for (let i = 0; i < li.length; i++) {
    for (let j = 0; j < 300000; j++) {
      console.log(li[i]);
    }
  }
}
```

`O(n log n)`

```
const splitButIterate = (li) => { // [1,2,3,4,5,6,7,8]
  if (li.length < 2) return li;
  const midIdx = li.length / 2;

  splitButIterate(li.slice(0, midIdx)); // 1,2,3,4
  splitButIterate(li.slice(midIdx)); // 5,6,7,8

  li.forEach(ele => console.log(ele))
};
```

`O(n^2)`

```
const dreadedDubs = (li) => {
  for (let i = 0; i < li.length; i++) {
    for (let j = 0; j < li.length; j++) {
      print(j);
    }
  }
}
```

`O(2^n)`

```
const twoN = (n) => {
  if (n == 1) return n;

  twoN(n - 1);
  twoN(n - 1)
}
```

`O(n!)`

```
const factorial = (n) => {
  if (n === 1) return n;

  for (let i = 0; i < n; i++) {
    factorial(n - 1);
  }
};
```

Memoization and Tabulation Learning Objectives

1. Apply memoization to recursive problems to make them less than polynomial time

```
const fibonacci = (n, memo = { 0: 0, 1: 1 }) => {
  if (n in memo) return memo[n];
  memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo);
  return memo[n];
};
```

2. Apply tabulation to iterative problems to make them less than polynomial time

```
function tabulatedFib(n) {
  // create a blank array with n reserved spots
  let table = new Array(n);

  // seed the first two values
  table[0] = 0;
  table[1] = 1;

  // complete the table by moving from left to right,
  // following the fibonacci pattern
  for (let i = 2; i <= n; i += 1) {
    table[i] = table[i - 1] + table[i - 2];
  }

  return table[n];
}

console.log(tabulatedFib(7));      // => 13
```

Sorting Algorithms Learning Objectives

1. Explain the complexity of and write a function that performs bubble sort on an array of numbers

```
function swap(array, idx1, idx2) {
  [array[idx1], array[idx2]] = [array[idx2], array[idx1]]
}

function bubbleSort(array) {
  let swapped = false

  while (!swapped) {
    swapped = true;

    for (let i = 0; i < array.length; i++) {
      if (array[i] > array[i + 1]) {
        swap(array, i, i+1);
        swapped = false;
      }
    }
  }
}
```

Time Complexity: O(n^2)

The inner for loop contributes O(n) in isolation. In the worst case scenario, the while loop will need to run n times to bring all n elements into their final resting positions.

Space Complexity: O(1)

Bubble sort uses the same amount of memory and create the same amount of variables regardless of the size of the input.

2. Explain the complexity of and write a function that performs selection sort on an array of numbers

```
function swap(arr, index1, index2) {
  [arr[index1], arr[index2]] = [arr[index2], arr[index1]];
}

function selectionSort(list) {
  for (let i = 0; i < list.length; i++) {
    let min = i;

    for (let j = i + 1; j < list.length; j++) {
      if (list[j] < list[min]) {
        min = j;
      }
    }

    if (min !== i) {
      swap(list, i, min);
    }
  }
}
```

Time Complexity: O(n^2)

The outer loop i contributes O(n) in isolation. The inner loop j will contribute roughly O(n / 2) on average. The two loops are nested so our total time complexity is O(n * n / 2) = O(n^2).

Space Complexity: O(1)

We use the same amount of memory and create the same amount of variables regardless of the size of our input.

3. Explain the complexity of and write a function that performs insertion sort on an array of numbers

```
function insertionSort(list) {
  for (let i = 1; i < list.length; i++) {
    value = list[i];
    hole = i;

    while (hole > 0 && list[hole - 1] > value) {
      list[hole] = list[hole - 1];
      hole--;
    }

    list[hole] = value;
  }
}
```

Time Complexity: O(n^2)

The outer loop i contributes O(n) in isolation. The inner while loop will contribute roughly O(n / 2) on average. The two loops are nested so our total time complexity is O(n * n / 2) = O(n^2).

Space Complexity: O(1)

We use the same amount of memory and create the same amount of variables regardless of the size of our input.

4. Explain the complexity of and write a function that performs merge sort on an array of numbers

```
function merge(array1, array2) {
  let result = []
  while (array1.length && array2.length) {
    if (array1[0] < array2[0]) {
      result.push(array1.shift());
    } else {
      result.push(array2.shift());
    }
  }

  return [...result, ...array1, ...array2];
}

function mergeSort(array) {
  if (array.length <= 1) return array;

  const mid = Math.floor(array.length / 2)
  const left = mergeSort(array.slice(0, mid));
  const right = mergeSort(array.slice(mid));

  return merge(left, right);
}
```

Time Complexity: O(n log(n))

Since we split the array in half each time, the number of recursive calls is O(log(n)). The while loop within the merge function contributes O(n) in isolation and we call that for every recursive mergeSort call.

Space Complexity: O(n)

We will create a new subarray for each element in the original input.

5. Explain the complexity of and write a function that performs quick sort on an array of numbers

```
function quickSort(array) {
  if (array.length <= 1) return array;

  let pivot = array.shift();

  let left = array.filter(x => x < pivot);
  let right = array.filter(x => x >= pivot);

  let sortedLeft = quickSort(left);
  let sortedRight = quickSort(right);

  return [...sortedLeft, pivot, ...sortedRight];
}
```

Time Complexity

- *Avg Case: O(n log(n))*
  The partition step alone is O(n). We are lucky and always choose the median as the pivot. This will halve the array length at every step of the recursion O(log(n)).

- *Worst Case: O(n2)*
  We are unlucky and always choose the min or max as the pivot. This means one partition will contain everything, and the other partition is empty O(n).

Space Complexity: O(n)

Our implementation of quickSort uses O(n) space because of the partition arrays we create.

6. Explain the complexity of and write a function that performs a binary search on a sorted array of numbers.

```
function binarySearch(list, target) {
  if (list.length === 0) return false;

  let mid = Math.floor(list.length / 2);

  if (list[mid] === target) {
    return true;
  } else if (list[mid] > target) {
    return binarySearch(list.slice(0, mid), target);
  } else {
    return binarySearch(list.slice(mid+1), target);
  }
}
```

Time Complexity: O(log(n))

The number of recursive calls is the number of times we must halve the array until it's length becomes 0.

**Space Complexity: O(n)

Our implementation uses n space due to half arrays we create using slice.

Lists, Stacks and Queues Learning Objectives

1. Explain and implement a List

A Linked List is a list made of up individual nodes, each node containing a value
and a reference to the next node (and optionally the previous node) in the list.

The List itself only needs to keep track of the head node (and optionally the tail node if you want to add things to the end of the list in constant time)
of the list, since it can follow the references on the head (or tail) node to get to any
other node. The list can optionally also keep track of the length of the list.

Linked Lists enable the following operations:

- `addToTail` - Adds a new node to the tail of the list
- `addToHead` - Adds a new node to the head of the list
- `insertAt` - Inserts a new node at a certain position in the list
- `removeTail` - Removes the tail node from the list
- `removeHead` - Removes the head node from the list
- `remove` - Removes a node from a certain position in the list
- `contains` - Searches the list for a node with a particular value
- `get` - Gets the node at a specific position
- `set` - Sets the value of a node at a specific position
- `size` - Gets the length of the Linked List

A Node could be expressed as a class like this:

```
class Node {
    constructor(value, next) {
        this.value = value;
        this.next = next;
    }
}
```

while a basic Linked List class could look like this:

```
class LinkedList {
    constructor() {
        this.head = null;
        this.tail = null;
        this.length = 0;
    }

    addToTail(val) {
        const newNode = new Node(val);

        if (!this.head) {
            this.head = newNode;
        } else {
            this.tail.next = newNode;
        }

        this.tail = newNode;
        this.length++;
        return this;
    }

    removeTail() {
        if (!this.head) return undefined;
        let current = this.head;
        let newTail = current;
        while (current.next) {
            newTail = current;
            current = current.next;
        }
        this.tail = newTail;
        this.tail.next = null;
        this.length--;
        if (this.length === 0) {
            this.head = null;
            this.tail = null;
        }
        return current;
    }

    addToHead(val) {
        let newNode = new Node(val);
        if (!this.head) {
            this.head = newNode;
            this.tail = newNode;
        } else {
            newNode.next = this.head;
            this.head = newNode;
        }
        this.length++;
        return this;
    }

    removeHead() {
        if (!this.head) return undefined;
        const currentHead = this.head;
```

```
        this.head = currentHead.next;
        this.length--;
        if (this.length === 0) {
            this.tail = null;
        }
        return currentHead;
    }

    contains(target) {
        let node = this.head;
        while (node) {
            if (node.value === target) return true;
            node = node.next;
        }
        return false;
    }

    get(index) {
        if (index < 0 || index >= this.length) return null;
        let counter = 0;
        let current = this.head;
        while (counter !== index) {
            current = current.next;
            counter++;
        }
        return current;
    }

    set(index, val) {
        const foundNode = this.get(index);
        if (foundNode) {
            foundNode.value = val;
            return true;
        }
        return false;
    }

    insert(index, val) {
        if (index < 0 || index > this.length) return false;
        if (index === this.length) return !!this.addToTail(val);
        if (index === 0) return !!this.addToHead(val);

        const newNode = new Node(val);
        const prev = this.get(index - 1);
        const temp = prev.next;
        prev.next = newNode;
        newNode.next = temp;
        this.length++;
        return true;
    }

    remove(index) {
        if (index < 0 || index >= this.length) return undefined;
        if (index === 0) return this.removeHead();
        if (index === this.length - 1) return this.removeTail();
        const previousNode = this.get(index - 1);
        const removed = previousNode.next;
        previousNode.next = removed.next;
        this.length--;
        return removed;
    }

    size() {
        return this.length;
    }
}
```

2. Explain and implement a Stack

A Stack is a Last In First Out (LIFO) Data structure.

You can usually perform the following operations on a stack:

- `push` a value onto the top of the stack
- `pop` a value off the top of the stack
- `size` get the size of the stack

The example below uses the same Node class as our Linked List.

You could also use a plain array to implement a stack (and people often do)

```
class Stack {
  constructor() {
    this.top = null;
    this.length = 0;
  }

  push(val) {
    const newNode = new Node(val);
    if (!this.top) {
      this.top = newNode;
    } else {
      const temp = this.top;
      this.top = newNode;
      this.top.next = temp;
    }
    return ++this.length;
  }

  pop() {
    if (!this.top) {
      return null;
    }
    const temp = this.top;
    this.top = this.top.next;
    this.length--;
    return temp.value;
  }

  size() {
    return this.length;
  }
}
```

3. Explain and implement a Queue

Again this uses the same Node class as the Linked List and Stack.

You could also use a plain array to implement a queue (and people often do)

Queues are a First In First Out (FIFO) data structure.

Queues usually implement the following operations:

- `enqueue` Adds a value to the back of the queue
- `dequeue` Removes a value from the front of the queue
- `size` Gets the size of the queue

```
class Queue {
  constructor() {
    this.front = null;
    this.back = null;
    this.length = 0;
  }

  enqueue(val) {
    const newNode = new Node(val);
    if(!this.front) {
      this.front = newNode;
      this.back = newNode;
    } else {
      this.back.next = newNode;
      this.back = newNode;
    }
    return ++this.length;
  }

  dequeue() {
    if (!this.front) {
      return null;
    }
    const temp = this.front;
    if (this.front === this.back) {
      this.back = null;
    }
    this.front = this.front.next;
    this.length--;
    return temp.value;
  }

  size() {
    return this.length;
  }
}
```

Binary Trees and Binary Search Trees

1. Explain and implement a Binary Tree

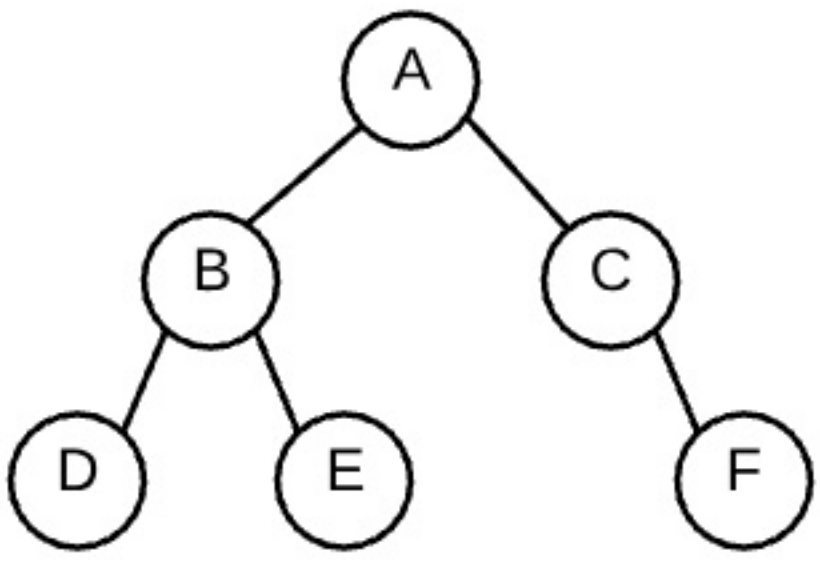
A Binary Tree is a Tree where nodes have at most 2 children, usually we represent these as 'left' and 'right'.

```
class TreeNode {
  constructor(val) {
    this.val = val;
    this.left = null;
    this.right = null;
  }
}
```

```
let a = new TreeNode('a');
let b = new TreeNode('b');
let c = new TreeNode('c');
let d = new TreeNode('d');
let e = new TreeNode('e');
let f = new TreeNode('f');

a.left = b;
a.right = c;
b.left = d;
b.right = e;
c.right = f;
```

Creates the following Binary Tree



Here's a complete implementation of a binary search tree

```
class TreeNode {
    constructor(val) {
        this.val = val;
        this.left = null;
        this.right = null;
    }
}


class BST {
    constructor() {
        this.root = null;
    }

    insert(val, currentNode=this.root) {
        if(!this.root) {
            this.root = new TreeNode(val);
            return;
        }

        if (val < currentNode.val) {
            if (!currentNode.left) {
                currentNode.left = new TreeNode(val);
            } else {
                this.insert(val, currentNode.left);
            }
        } else {
            if (!currentNode.right) {
                currentNode.right = new TreeNode(val);
            } else {
                this.insert(val, currentNode.right);
            }
        }
    }

    searchRecur(val, currentNode=this.root) {
        if (!currentNode) return false;

        if (val < currentNode.val) {
            return this.searchRecur(val, currentNode.left);
        } else if (val > currentNode.val){
            return this.searchRecur(val, currentNode.right);
        } else {
            return true;
        }
    }

    searchIter(val) {
        let currentNode = this.root;

        while (currentNode) {
            if (val < currentNode.val) {
                currentNode = currentNode.left;
            } else if (val > currentNode.val){
                currentNode = currentNode.right;
            } else {
                return true;
            }
        }

        return false;
    }
}

module.exports = {
    TreeNode,
    BST
};
```

2. Identify the three types of tree traversals: pre-order, in-order, and post-order

These are all depth first traversals, which means using recursion.

Pre-order

1. Access the data of the current node
2. Recursively visit the left sub tree

3. Recursively visit the right sub tree

In-Order

1. Recursively visit the left sub tree
2. Access the data of the current node
3. Recursively visit the right sub tree

Post-Order

1. Recursively visit the left sub tree
2. Recursively visit the right sub tree
3. Access the data of the current node

3. Explain and implement a Binary Search Tree

A Binary Search Tree is a special kind of Binary Tree where the following is true:

- given any node of the tree, the values in the left subtree must all be strictly less than the given node's value.
- and the values in the right subtree must all be greater than or equal to the given node's value

Or to say it with recursion:

- the left subtree contains values less than the root
- AND the right subtree contains values greater than or equal to the root
- AND the left subtree is a Binary Search Tree
- AND the right subtree is a Binary Search Tree

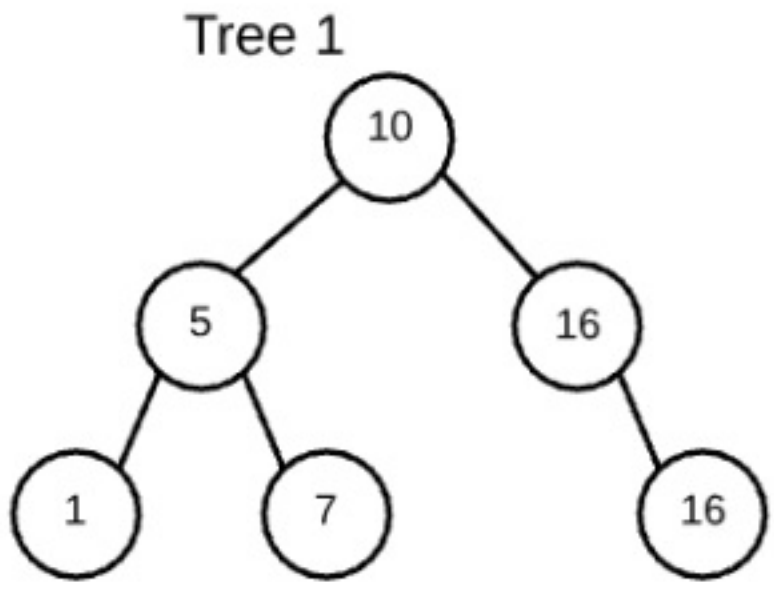
Some definitions of binary search trees allow duplicates and some do not. We can write our code to deal with the duplicates though.

Example of a Binary Search Tree:

```
let ten = new TreeNode('10');
let five = new TreeNode('5');
let sixteen = new TreeNode('16');
let one = new TreeNode('1');
let seven = new TreeNode('7');
let sixteenDuplicate = new TreeNode('16');

ten.left = five;
ten.right = sixteen;
five.left = one;
five.right = seven;
sixteen.right = sixteenDuplicate;
```

Generates this tree:



Usually we will write a class for our BST so we can have an "insert" method.
This way we can control the order of how we insert the nodes to make sure the tree is a binary search tree and also a balanced BST.

Graphs

1. Explain and implement a Graph

A graph is a collection of nodes and any edges between those nodes. It is a broad category. Linked lists and trees are both subclasses of graphs.

We can build a graph out of objects by making a GraphNode class

```
// Class based Graph
class GraphNode {
    constructor(val) {
        this.val = val;
        this.neighbors = [];
    }
}

let a = new GraphNode('a');
let b = new GraphNode('b');
let c = new GraphNode('c');

a.neighbors = [b];
b.neighbors = [c];
c.neighbors = [a];
```

Adjacency Lists are another way to make a graph

```
// adjacency list (Non-class based)
let graph = {
    'a': ['b'],
    'b': ['c'],
    'c': ['a']
}
```

Breadth-first Search on the adjacency list graph.

When doing breadth first, iterative is simpler and easier.
We can use a queue to do this.

Extra Challenge, use the Queue class we made last week as the queue to use inside of this function, instead of just using an array as a queue

```
function breadthFirstSearch(graph, startingNode, targetVal) {
    // Populate our queue with the starting Node
    let queue = [ startingNode ];
    // Create a new empty Set to hold the nodes we've visited
    let visited = new Set();

    //  Keep going until the queue is empty
    while(queue.length) {
        // Dequeue the first thing from the queue
        let node = queue.shift();

        // If we've visited this node before, then just continue, which goes
        // back up to the while loop
        if(visited.has(node)) continue;
        // Add the node to the visited set.
        visited.add(node);

        // Check to see if the node is the one we are looking for, if it is
        // return true
        if (node === targetVal) return true;
        // Enqueue the node's neighbors from the adjacency graph onto the queue
        queue.push(...graph[node]);
    }
    // If we made it through the loop without finding one, return false
    return false;
}

breadthFirstSearch(graph, 'a', 'c'); // true
```

Depth-first Search on the adjacency list graph.

It is easier to do a recursive solution for depth first.
Depth first usually uses a stack, in this case the recursive call-stack is acting as our stack.

```
function depthFirstSearch(graph, startingNode, targetVal, visited=new Set()) {
    // If we found the node, return true
    if (startingNode === targetVal) {
        return true;
    }

    let neighbors = graph[startingNode];
    for (let neighbor of neighbors) {
        // If the neighbor has already been visited, we can
        // skip it.
        if (visited.has(neighbor)) continue;

        // Add the neighbor to the visited set
        visited.add(neighbor);

        // Now we recurse to check the neighbor and return the result
        return depthFirstSearch(graph, neighbor, targetVal, visited);
    }
    // If we didn't find it, return true
    return false;
}

console.log(depthFirstSearch(graph, 'a', 'c'));
```

Network Models Objectives

1. Describe the structure and function of network models from the perspective of a developer

OSI is a reference model and doesn't match up very well to how things actually work in the real world. It has seven layers. Some descriptions of the OSI model try to fit our existing tech into the seven layer model but it doesn't match up exactly with how networks work today.

OSI Network Model

1. Application - HTTP
2. Presentation - JPEG/GIF
3. Session - RPC
4. Transport - TCP/UDP
5. Network - IP
6. Data Link - Ethernet
7. Physical - DSL, 802.11

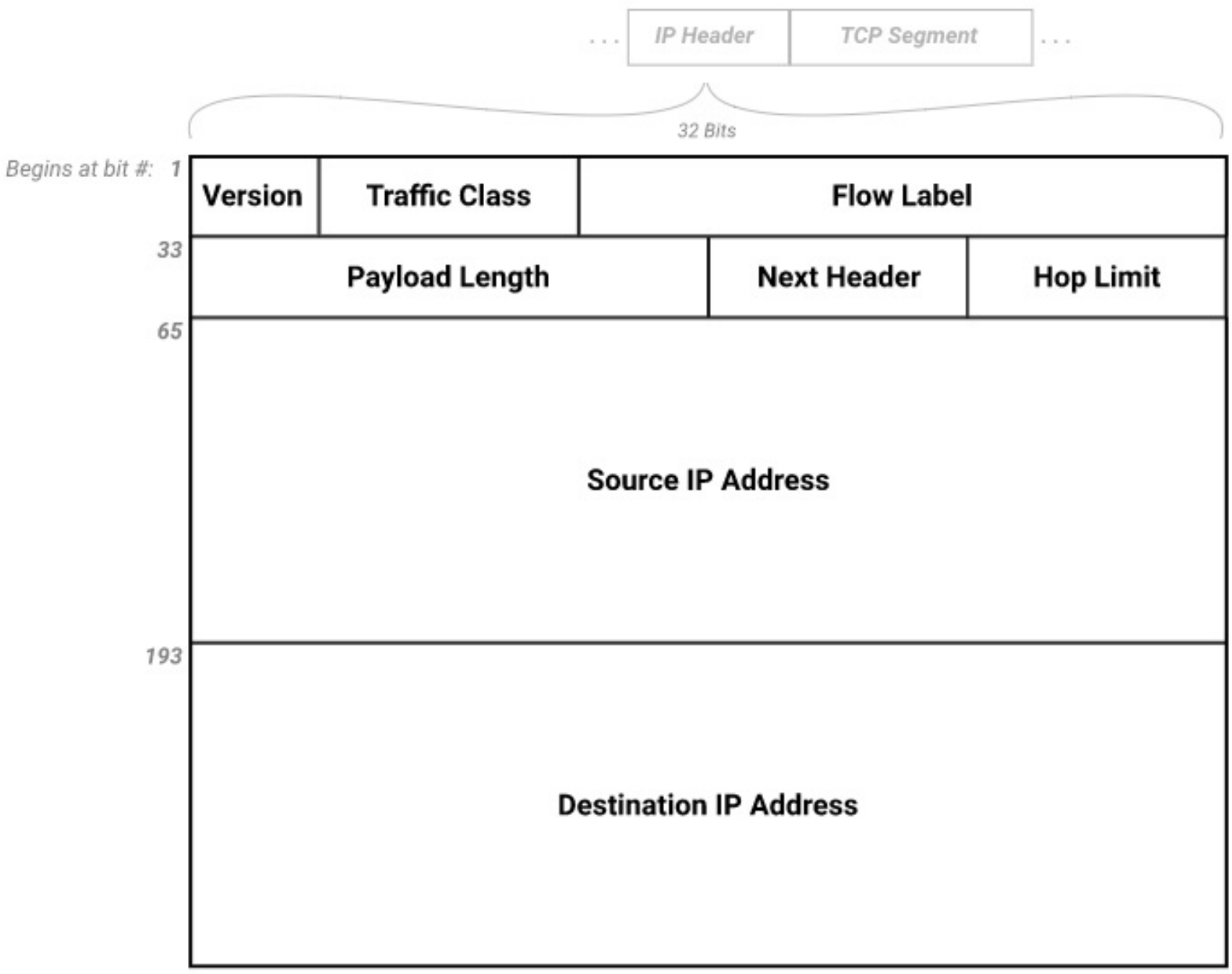
The TCP/IP model is the actual way networks today work, it is simpler than the OSI model and has only four layers.

TCP/IP Network Model

1. Application - HTTP, HTTPS, FTP, SMTP, etc
2. Transport - TCP or UDP
3. Internet - IP
4. Link - Ethernet

Internet Protocol Suite Objectives

1. Identify the correct fields of an IPv6 header



2. Distinguish an IPv4 packet from an IPv6

The Version number is stored in the headers of IP packets as a binary number.

```
IPv4 = 4 = 0100
IPv6 = 6 = 0110
```

IPv4 addresses are made up of 4 octets, each a 8-bit binary number converted to decimal.

Example: `192.168.1.1`

IPv6 addresses are made up of a 128bit number. It is usually resprentened in hexidecimal, with every four digits separated by a `:`

Example: `2001:0db8:85a3:0000:0000:8a2e:0370:7334`

In IPv6 addresses you can also compress the zeros to make it shorter to write:

The rules are:

- An entire string of zeros can be removed, you can only do this once.
- 4 zeros can be removed, leaving only a single zero.
- Leading zeros can be removed.

Example: `2001:db8:85a3::0:8a2e:370:7334`

3. Describe the following subjects and how they relate to one another: IP Addresses, Domain Names, and DNS

- *IP Address* - The internet protocol address assigned to a particular networking device (Ethernet adapter, Wi-Fi Adapter, etc). IPv4 example: `192.168.1.1`.
- *Domain Name* - A human readable name assigned to an IP address. Examples: `google.com` or `appacademy.io`
- *DNS* - Domain Name System: A protocol (on UDP port 53) that allows our computer to talk to a DNS Server and *resolve* a Domain Name into an IP Address. Example: `google.com` might resolve to `172.217.6.142`

Common DNS Record Types are:

- A - Directly maps a domain name to an IPv4 Address
- AAAA - Directly maps a domain name to an IPv6 Address
- CNAME - Maps a domain name to another domain name
- MX - Defines the mail server for a domain
- NS - Defines the DNS Servers for a zone (domain)
- SOA - Defines which DNS Server is the authority for a zone(domain)

4. Identify use cases for the TCP and UDP protocols

TCP - Transmission Control Protocol

TCP is used when you want reliable connections and you want the packets to reach the destination in the correct order. Web Browsing, Downloading Files, fetching Email from a server, Streaming Music or Video are all examples of TCP

UDP - user Datagram Protocol

UDP is used when you don't mind an more unreliable connection, but where real time interactivity is more important. If you drop a few packets, no big deal. Examples are Voice Over IP Telephone calls, and Video Chat systems like Facetime or Zoom.

5. Describe the following subjects and how they relate to one another: MAC Address, IP Address, and a port

- *MAC Address* - A hardware address assigned to every physical networking device on a network. These are assigned usually at the time the device was manufacturered, although in some case they can be changed via software. They look like a series of Hexidecimal values separated by `:` characters. Example: `ea:de:36:d9:5a:b8`. They are used to communicate on the local network *only*.
- *IP Address* - An address assigned to a networking device. These are usually assigned in software, and may be automatically assigned by an ISP or by a router on a local network using DHCP. They are used to `route` connections across multiple networks. Example: IPV4 Address `192.168.1.1`. A Router will map IP Addresses to MAC Addresses to keep track of connections.
- *Port* - Represents a TCP/UDP connection on an actual computer. Valid ports are numbers in the range from 0-65535. Used by the operating system of a computer to route TCP connections to the right program running on a computer. These programs can be said to be "listening" on a port. No two programs are allowed to listen on the same port at once. The default ports for web servers are HTTP(80) and HTTPS(443).

6. Identify the fields of a TCP segment

This was covered in the optional lectures and therefore isn't on the assessment

7. Describe how a TCP connection is negotiated

This was covered in the optional lectures and therefore isn't on the assessment

8. Explaining the difference between network devices like a router and a switch

1. *Hub* - A device which hooks multiple computers together over ethernet and blindly repeats ethernet packets to all the other devices on a local area network. These are not used much anymore
2. *Switch* - A device which intelligently hooks multiple computers together over ethernet and sends ethernet packets to the correct devices on a local area network based on MAC Addresses.
3. *Router* - A device which is reponsible for routing IP packets BETWEEN different networks.

Week 10 Study Guide

RDBMS And Database Entity Objectives

Define what a relational database management system is

The RDBMS is a software application that you run that your programs can connect to so that they can store, modify, and retrieve data.

Describe what relational data is

Relational Data is data that is related in some way. For instance user data in a photo sharing application might have a relationship with the photo data and photo data and user data might both have a relationship with comment data.

There are three types of relationships you can have

1. One to One
2. One to Many
3. Many to Many

Define what a database is

A database is a collection of structured data stored in a Database "System" or "Server"

Define what a database table is

A database can have many tables, a table is made up of several `Columns`. Each `Column` is of a certain type. A table `Schema` describes the columns in a table. Tables also contain `Rows` which hold the actual data for the table.

Describe the purpose of a primary key

A primary key is a single unique column in a database table.

Describe the purpose of a foreign key

A *foreign key* is an integer column in a table which holds the value of a matching *primary key* from another table. A *foreign key constraint* insures that the id stored in the foreign key column is a valid primary key in the related table.

Connect to an instance of PostgreSQL with the command line tool psql

Usage of psql:

```
psql -U <database username> -h <hostname> <database name>
```

- `-U` defaults to the same as your unix username
- `-h` does not have a default value. If you leave it out, psql connects through a unix socket instead of connecting to a hostname through the network
- `<database name>` defaults to be the same as whatever `<database username>` is

You can set these two *environment variables* (`PGHOST` and `PGUSER`) in your shell to override the default behavior `-U` and `-h`

For example, to always connect to localhost with the database user `postgres` you would set them to this:

```
export PGHOST=localhost
export PGUSER=postgres
```

You can create a hidden file called `.pgpass` and put it into your unix home directory to set the password for psql, so you don't have to type it everytime.

Note: PostgreSQL comes with a default database called 'postgres' and a default superuser names 'postgres'

Identify whether a user is a normal user or a superuser by the prompt in the psql shell

You use the `\du` command. If a user is a superuser it will have the `Superuser` attribute.

```
postgres-# \du
                                   List of roles
       Role name        |                         Attributes                         | Member of
------------------------+------------------------------------------------------------+-----------
 postgres               | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 my_user                |                                                            | {}
```

Create a user for the relational database management system

```
CREATE USER <database username> WITH PASSWORD <password> <attributes>;
```

Attributes can be things like `SUPERUSER` or permissions like `CREATEDB` , or any of the other attributes listed in the documentation

Create a database in the database management system

Configure a database so that only the owner (and superusers) can connect to it

```
CREATE DATABASE <database name> WITH OWNER <database username>
```

Create Database Documentation

View a list of databases in an installation of PostgreSQL

You can use the `\l` command in psql to do this.

```
postgres-# \l
                                      List of databases
      Name       |         Owner          | Encoding |  Collate   |   Ctype    |   Access privileges
-----------------+------------------------+----------+------------+------------+-----------------------
 aa_times        | aa_times               | UTF8     | en_US.utf8 | en_US.utf8 |
 postgres        | postgres               | UTF8     | en_US.utf8 | en_US.utf8 |
 project_manager | project_management_app | UTF8     | en_US.utf8 | en_US.utf8 |
 recipe_box      | recipe_box_app         | UTF8     | en_US.utf8 | en_US.utf8 |
 template0       | postgres               | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres          +
                 |                        |          |            |            | postgres=CTc/postgres
 template1       | postgres               | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres          +
                 |                        |          |            |            | postgres=CTc/postgres
(6 rows)
```

Create tables in a database

The basic format of `CREATE TABLE` is this:

```
CREATE TABLE <table name> (
  <column name> <data type>,
  <column name> <data type>,
  ...
  <column name> <data type>
);
```

You can find a list of possible data types in the PostgreSQL documentation

View a list of tables in a database

You can use the `\dt` command in psql to do this

```
aa_times-# \dt
 public | people   | table | aa_times
 public | sections | table | aa_times
 public | stories  | table | aa_times
```

Identify and describe the common data types used in PostgreSQL

Here are some of most common datatypes

- SERIAL
- VARCHAR
- TEXT
- NUMERIC
- INTEGER
- BOOLEAN
- TIMESTAMP

Describe the purpose of the UNIQUE and NOT NULL constraints, and create columns in database tables that have them

UNIQUE Constraint

Unique constraints ensure that the data contained in a column, is unique among all the rows in the table.

```
CREATE TABLE products (
    id integer UNIQUE,
    name text,
    price numeric
);
```

UNIQUE Documentation

NOT NULL Constraint

A not-null constraint simply specifies that a column must not assume the null value.

```
CREATE TABLE products (
    id integer NOT NULL,
    name text NOT NULL,
    price numeric
);
```

NOT NULL Documentation

Create a primary key for a table

A primary key constraint indicates that a column can be used as a unique identifier for rows in the table. This requires that the values be both unique and not null.

```
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name text,
    price numeric
);
```

PRIMARY KEY Documentation

Create foreign key constraints to relate tables

A foreign key constraint specifies that the values in a column (or a group of columns) must match the values appearing in some row of another table.

You can use the `REFERENCES` keyword:

```
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    id SERIAL PRIMARY KEY,
    product_id integer REFERENCES products(id),
    quantity integer
);
```

Or you can use the `FOREIGN KEY` syntax on a separate line

```
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    id SERIAL PRIMARY KEY,
    product_id integer,
    quantity integer,
    FOREIGN KEY (product_id) REFERENCES products(id)
);
```

FOREIGN KEY Documentation

Explain that SQL is not case sensitive for its keywords but is for its entity names

Both of these are valid SQL, although it is a convention to uppercase the SQL keywords.

```
SELECT name, quantity FROM orders;
```

```
select name, quantity from orders;
```

This is NOT the same:

```
SELECT "Name", "Quantity" FROM "Orders";
```

(Note, in PostgreSQL, if we do not put double quotes around the column and table names, postgres will lowercase them first before running the query)

So a query like this:

```
SELECT Name, Quantity FROM Orders;
```

Will be turned into this before postgres runs it.

```
SELECT name, quantity FROM orders;
```

So if you actually have capital letters in your column and table names, make sure to always double-quote them.

SQL Objectives

1. How to use the SELECT … FROM … statement to select data from a single table.

```
SELECT population
FROM countries;
```

2. How to use the WHERE clause on SELECT, UPDATE, and DELETE statements to narrow the scope of the command.

```
SELECT population
FROM countries
WHERE name = 'France';
```

```
UPDATE countries
SET population = 1000
WHERE name = "Vatican City";
```

```
DELETE FROM planets
WHERE name = 'Pluto';
```

3. How to use the JOIN keyword to join two (or more) tables together into a single virtual table.

```
SELECT player_name
FROM players
JOIN teams ON teams.id = players.team_id
WHERE team_name = 'Lakers';
```

4. How to use the INSERT statement to insert data into a table.

```
INSERT INTO nfl_players
VALUES (DEFAULT, 'Joe Burrow', 'Bengals');
```

5. How to use a seed file to populate data in a database.

```
-- seed_file.sql
CREATE TABLE IF NOT EXISTS bands (
    id SERIAL,
    name VARCHAR(50) NOT NULL,
    vocalist VARCHAR(50),
    PRIMARY KEY (id)
);

INSERT INTO bands
VALUES (DEFAULT, 'The Beatles', 'John Lennon');

INSERT INTO bands
VALUES (DEFAULT, 'Queen', 'Freddie Mercury');

INSERT INTO bands
VALUES (DEFAULT, 'U2', 'Bono');
```

SQL Learning Objectives Pt 2

1. How to perform relational database design

Database design is a difficult subject with few absolutes. Experience building applications and resolving design issues will help you to make judgement calls.

But as a starting point, start with these four steps:

```
1. What are the main entities in my application (nouns)?
2. How are they related to one another?
3. Can I normalize any information?
```

2. How to use transactions to group multiple SQL commands into one succeed or fail operation

```
```js
async function transferFunds(pool, account1, account2, amount) {
    const balanceQ = 'select balance from "AccountBalances" where account_id = $1';
    const updateBalanceQ = 'update "AccountBalances" set balance=$1 where account_id = $2';

    await pool.query("BEGIN;");
    try {
        const balance1 = await pool.query(balanceQ, [account1]);
        if (balance1 < amount) {
            throw ("Not enough funds");
        }
        const balance2 = await pool.query(balanceQ, [account2]);

        await pool.query(updateBalanceQ, [balance2 + amount, account2]);
        await pool.query(updateBalanceQ, [balance1 - amount, account1]);
        await pool.query("COMMIT;");
    } catch (e) {
        await pool.query("ROLLBACK;");
    }

}
```
```

3. How to apply indexes to tables to improve performance

Indexes are used to optimize queries. We add indexes to columns, in order to allow the Query Planner to more efficiently filter matching rows.

By evaluating the WHERE clause of a poorly performing query, we can determine which columns are involved in the query, and which indexes the Query Planner might be able to take advantage of. Knowing which indexes (there are lots of types) to add and when is an advanced topic in Database Management.

4. Explain what and why someone would use EXPLAIN

`EXPLAIN` and `EXPLAIN ANALYZE` are the tools that we have to improve poorly performing queries. By applying these keywords to the front of a query we are able to learn about which indexes Postgres is able to utilize, in comparison to which tables must be sequentially scanned.

Using `EXPLAIN` is also an advanced topic in Database Management. We should know that these tools exist, that they can give you insight into the performance of our queries, but we should not be expected to use them.

5. Demonstrate how to install and use the node-postgres library and its Pool object to query a PostgreSQL-managed database

To install node-postgres:

```
```bash
$ npm install node-postgres
```
```

Somewhere in your JS:

```
```js
const { Pool } = require('pg');

const pool = new Pool({
    database: <mydbname>,
    hostname: <mydbhostname>,
    username: <username>,
```
```

````
    password: <password>
});

const result = await pool.query("SELECT 1;");
```
````

6. Explain how to write prepared statements with placeholders for parameters of the form "$1", "$2", and so on

Prepared Statements allow us to create queries that don't need to know the constant values needed for the where clause in advance. Consider:

````
```js
const loginQuery = 'SELECT * FROM users WHERE username = $1 and password = $2;';

async function loginUser(username, password) {
    const results = await pool.query(loginQuery, [username, password]);
    ...
}
```
````

In addition the benefit of not having to use template strings, the `node-postgres` library is providing us a *huge* hidden security benefit! Prepared statements exist primarily to protect our applications from [https://developer.mozilla.org/en-US/docs/Glossary/SQL_Injection](SQL-injection attacks) - one of the most common type of *hacking* attacks that we see on the web.

ORM Objectives

How to install, configure, and use Sequelize, an ORM for JavaScript

We use Sequelize 5 in this course.

Installing Sequelize 5

Use npm to install `sequelize`, `sequelize-cli`, and `pg` because we are using PostgreSQL.

```
npm install sequelize@^5 sequelize-cli@^5 pg
```

Configuring sequelize

First you must initialize your project with sequelize

```
npx sequelize-cli init
```

Then edit the `config/config.json` file to add the appropriate database name, username, password and dialect.

The dialect should be `postgres` because that's the RDBMS we are using.

Using Sequelize

You generate Models, edit the models and migration files to your liking, and then use the model classes in your code to query, insert, update and delete data.

```
npx sequelize model:generate --name Cat --attributes "firstName:string,specialSkill:string"
```

This will generate two files, a migration file and a model file.

Now we can edit both to add any custom columns, constraints or associations (relationships).

After running our migrations, we can do queries.

We import our model like so:

```
const { Cat } = require('./models');
```

And now we can do queries against the model.

```
const cat = await Cat.findByPK(1);
```

How to use database migrations to make your database grow with your application in a source-control enabled way

Migrations are a set of instructions written in JavaScript to create, update, and remove tables and columns from our database.

Migrations always run ONCE in the order of the timestamps that prefix the filenames.

You can generate a migration file either by generating a model, or by generating a stand-alone migration like so.

```
npx sequelize-cli migration:generate --name add_last_name_column_to_users
```
```

How to perform CRUD operations with Sequelize

CRUD = "Create, Read, Update and Delete"

Create

You can use the `<Model>.create()` method, or the `<Model>.build()` along with `<instance>.save()`

Read

You can use `<Model>.findOne()`, `<Model>.findAll()`, or `<Model>.findByPk()` to query
and read data

Update

You can just set properties on an instance and call `.save()` or you can use the `.update()` method.

Delete

You can use the `.destroy()` method on an instance to destroy a single row, or `<Model>.destroy()`
with a `where` clause to destroy multiple rows.

Check out the Sequelize Documentation or Sequelize Cheatsheet for more examples and details.

How to query using Sequelize

We call static methods on the Model to query the database.

These are some of the most common ones.

```
const <instance> = await <Model>.findOne(<query options>);

const <array> = await <Model>.findAll(<query options>);

const <instance> = await <Model>.findByPk(<query options>);
```

Check out the Sequelize Documentation or Sequelize Cheatsheet for more examples and details.

How to perform data validations with Sequelize

You define data validations on the Sequelize Model:

You add a validate property to a column definition, and use one of the many
built in validations to define what is allowed for a column.

This uses the common `notNull` and `notEmpty` validations:

```
const Cat = sequelize.define('Cat', {
   firstName: {
     type: DataTypes.STRING,
     allowNull: false,
     validate: {
       notNull: {
         msg: "firstName must not be null",
       },
       notEmpty: {
         msg: "firstName must not be empty",
       }
     },
   }
```

The documentation contains an exhaustive list.
html#validations) of all possbile validations:

Note: Try not to get confused by the documentation's use of ES6 `class` based Sequelize models, for validation, the property still applies the same way, just in the `init()` method instead of the `define()` method.

How to use transactions with Sequelize

You call the sequelize.transaction() function and pass is a callback.

The callback will receive a copy of the transaction id `tx`.

You pass this id to any sequelize methods (like save()) that you want to be
a part of the transation.

If this callback succeeds without errors, sequelize will commit the transaction.

In this example if either `save()` fails, the entire transaction will be rolled
back and the database will not be changed.

```
await sequelize.transaction(async (tx) => {
    // Fetch Markov and Curie's accounts.
    const markovAccount = await BankAccount.findByPk(
        1, { transaction: tx },
    );
    const curieAccount = await BankAccount.findByPk(
        2, { transaction: tx }
    );

    // Increment Curie's balance by $5,000.
```

```
    curieAccount.balance += 5000;
    await curieAccount.save({ transaction: tx });

    // Decrement Markov's balance by $5,000.
    markovAccount.balance -= 5000;
    await markovAccount.save({ transaction: tx });
});
```

It's a good idea to wrap the call to sequelize.transaction in a `try catch` block, so we can handle the transaction failing in a graceful way.

```
async function main() {
  try {
    // Do all database access within the transaction.
    await sequelize.transaction(async (tx) => {
      // Fetch Markov and Curie's accounts.
      const markovAccount = await BankAccount.findByPk(
        1, { transaction: tx },
      );
      const curieAccount = await BankAccount.findByPk(
        2, { transaction: tx }
      );

      // Increment Curie's balance by $5,000.
      curieAccount.balance += 5000;
      await curieAccount.save({ transaction: tx });

      // Decrement Markov's balance by $5,000.
      markovAccount.balance -= 5000;
      await markovAccount.save({ transaction: tx });
    });
  } catch (err) {
    // Do something useful here like log the error or send a message to the user
  }

  await sequelize.close();
}

main();
```

Week 11 Study Guide

Table of Contents

Regular Expressions Objectives

1. Define the effect of the * operator and use it in a regular expression

`*` **star operator** - zero or more (of what's right before it)

Example of `*` operator

`/th*e/` t, zero or more h, e

2. Define the effect of the ? operator and use it in a regular expression

`?` **optional operator** - zero or one (of what's right before it)

Example of `?` operator

`at?t` - a, zero or one t, t

3. Define the effect of the + operator and use it in a regular expression

`+` **plus operator** - one or more (of what's right before it)

Example of `+` operator

`at+` - a, one or more t

4. Define the effect of the . operator and use it in a regular expression

`.` **dot operator** - any one character

Example of `.` operator

`e .` - e, space, any char

5. Define the effect of the ^ operator and use it in a regular expression

`^` **hat operator** - start of input anchor

Example of `^` operator

`^Is` - start of input, I, s

6. Define the effect of the $ operator and use it in a regular expression

`$` **money operator** - end of input

Example of `$` operator

`at.$` - a, t, any char, end of input

7. Define the effect of the [] bracket expression and use it in a regular expression

`[]` **square brackets** - your choice

Example of `[]`

`a[tm]e` - a, t or m, e

8. Define the effect of the - inside brackets and use it in a regular expression

`-` **dash operator** - (only works inside square brackets) - range of chars

Example of `-` operator

`[a-zA-Z]at` - any lower/uppercase char, a, t

9. Define the effect of the ^ inside brackets and use it in a regular expression

`^` **hat operator inside square brackets** - none of them

Example of `^` inside `[]`

`[^a-zA-Z]` - any non letter char

Other useful informaiton about regular expressions

Grouping operator

`()` **parenthesis** - used ($1, $2, $3) - primarily used to capture groups of chars

Example of `()` operator

`at( is)?` - a, t, optional (space, i, s)

Shorthands

- `\s` - white space
- `\d` - digit
- `\w` - word char
- `\S` - not white space
- `\D` - not a digit
- `\W` - not a word char

Using Regular expressions in Javascript

```
const re = /EX$/
const str = "We're learning REGEX"

const li = [ 1, 2, di ];
const re = /regex/i;

console.log(re.test(`learning about regex`));
console.log(re.test(`LEARNING ABOUT REGEX`));

let count = 0;

const newStr = str.replace(/e/ig, match => {
    count += 1
    return count;
  });
  console.log(newStr);

const di = { name: `Mimi`, age: 2 };
const str = `My name is %name% and I am %age% years old`;
const str2 = `My name is ${name} and I am ${age} years old`;

const re = /%\w+%/g

const replaced = str.replace(re, match => {
  // match = %name%
  const key = match.replace(/%/g, ``);
  // key = name
  return di[key];
});

console.log(replaced);
```

Node HTTP Objectives

1. Identify the five parts of a URL

Given this URL `https://example.com:8042/over/there?name=ferret#nose`

Scheme	Authority	Path	Query	Fragment
https	example.com:8042	/over/theme	name=ferret	nose

2. Identify at least three protocols handled by the browser

- https - Secure HTTP
- http - HTTP
- file - Opening a file
- ws - Websocket

3. Use an `IncomingMessage` object to

An `IncomingMessage` object is usually represented by the `req` variable.

access the headers sent by a client (like a Web browser) as part of the HTTP request

```
console.log(req.headers);
// Prints something like:
//
// { 'user-agent': 'curl/7.22.0',
//   host: '127.0.0.1:8000',
//   accept: '*/*' }
```

access the HTTP method of the request

```
console.log(req.method); // prints out the method like `GET` or `POST`
```

access the path of the request

```
console.log(req.url); // access the path of the request as a string
```

access and read the stream of content for requests that have a body

`IncomingMessage` is a subclass of `stream.Readable` in node, so we can read it like a stream.

Since `stream.Readable` is an `async iterable` we can use `for await..of` with it, and we'll get each chuck of data from the stream and we can concatenate them back together.

You don't need to know all the details of how async iterators work to use them like this:

```
let body = '';
for await (let chunk of req) {
  body += chunk;
}
```

4. Use a ServerResponse object to

write the status code, message, and headers for an HTTP response

Status code

```
res.statusCode = 404;
```

message

```
res.statusMessage = 'Page not found';
```

headers

```
res.setHeader('Content-Type', 'text/html');
```

write the content of the body of the response

Assuming body is a big string of HTML…

```
res.write(body);
```

properly end the response to indicate to the client (like a Web browser) that all content has been written

You can optionally pass some more data to write in the call to end.

```
// Without a string
res.end()
// With a string (assuming rest_of_body is some more HTML)
res.end(rest_of_body)
```

Express Objectives

1. Send plain text responses for any HTTP request.

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
    res.send('Hello World!');
});

const port = 8000;
app.listen(port, () => console.log(`App listening on port ${port}...`));
```

2. Use pattern matching to match HTTP request paths to route handlers.

```
// This matches /pictures followed by anything: /pictures/summer, /pictures/prom
app.get(/^\/pictures\/.+$/i, (req, res) => {
    res.send('Here are some pictures');
});

// This route will only match URL paths that have a number in the route parameter's position
app.get('/pictures/:id(\\d+)', (req, res) => {
    res.send('Here is a single picture');
});
```

3. Use the Pug template engine to generate HTML from Pug templates to send to the browser.

```
html
  head
    title Pug Demo
  body
    h1 Pug Demo
    h2 Pug allows you to do many things
    ul
      li: a(href='google.com') This is a link to google.com
      li: span.yellow This is a span with class='yellow'
    h2 And now some colors
```

This will render into the following HTML

```
<html>
  <head>
    <title>Pug Demo</title>
  </head>
  <body>
    <h1>Pug Demo</h1>
    <h2>Pug allows you to do many things</h2>
    <ul>
      <li><a href="google.com">This is a link to google.com</a></li>
      <li><span class="yellow">This is a span with class='yellow'</span></li>
    </ul>
    <h2>And now some colors</h2>
  </body>
</html>
```

4. Pass data to Pug templates to generate dynamic content

```
// app.js
app.get('/pug', (req, res) => {
  res.render('eod', {
    title: 'EOD demo',
```

```
    header: 'Pug demo',
    colors: ['blue', 'red', 'green']});
});
```

```
html
  head
    title= title
    style
      include style.css
  body
    h1 This is the #{header} page which does string interpolation
    h2 Pug allows you to do many things
    #violet This is a div with an id='violet'
    .purple This is a div with a class='purple'

    p
      - const a = 1 + 2; // Look!, we can do inline javascript that doesn't output!
      = `The answer is ${a}` // Look this runs some inline javascript that does output!

    ul
      li: a(href='http://google.com') This is a link to google.com
      li: span.yellow This is a span with class='yellow'
    h2 And now some colors
    ul
    each color in colors
      li= color
```

This outputs this html

```
<html>
  <head>
    <title>EOD demo</title>
    <style><!-- CSS would be here from the styles.css file --></style>
  </head>
  <body>
    <h1>This is the Pug demo page which does string interpolation</h1>
    <h2>Pug allows you to do many things</h2>
    <div id="violet">This is a div with an id='violet'</div>
    <div class="purple">This is a div with a class='purple'</div>
    <p>The answer is 3
    </p>
    <ul>
      <li><a href="http://google.com">This is a link to google.com</a></li>
      <li><span class="yellow">This is a span with class='yellow'</span></li>
    </ul>
    <h2>And now some colors</h2>
    <ul></ul>
    <li>blue</li>
    <li>red</li>
    <li>green</li>
  </body>
</html>
```

5. Use the Router class to modularize the definition of routes

```
// routes.js
const express = require('express');

const eodRoutes = express.Router();

// Because, this sub-router is registered on /eod "app.use('/eod', eodRoutes);" in app.js), it will match the route /eod/questions
eodRoutes.get('/questions', (req, res) => {
  res.send('Answers');
});

module.exports = eodRoutes;
```

```
// app.js
const express = require('express');
const eodRoutes = require('./routes');

// Create the Express app.
const app = express();

app.use('/eod', eodRoutes);

// Define a port and start listening for connections.
const port = 8081;

app.listen(port, () => console.log(`Listening on port ${port}...`));
```

Pug Objectives

1. Declare HTML tags and their associated ids, classes, attributes, and content.

```
html
  head
    title
  body
    div#main
      div.blue
      div.yellow
        a(href="http://google.com") Click here
```

2. Use conditional statements to determine whether or not to render a block.

```
// app.js
res.render('layout', { isEOD: true });
```

```
if isEOD
  h2 Welcome back!
else
  h2 Keep coding!
```

3. Use interpolation to mix static text and dynamic values in content and attributes.

We use the `#{}` syntax to do interpolation in pug templates

```
res.render('layout', {
  title: 'Pug demo page',
  header: 'interpolation'
})
```

```
html
  head
    title= title
    style
      include style.css
  body
    h1 Pug does #{header}
    h2 Pug allows you to do many things
    ul
      li: a(href='http://google.com') This is a link to google.com
```

4. Use iteration to generate multiple blocks of HTML based on data provided to the template.

```
// app.js
app.get('/pug', (req, res) => {
  res.render('eod', { colors: ['blue', 'red', 'green'] });
});
```

```
ul
  each color in colors
    li= color
```

HTML Form Objectives

1. Describe the interaction between the client and server when an HTML form is loaded into the browser, the user submits it, and the server processes it

```
<form action="/users" method="post">
  <label>Username:
    <input type="text" name="user[username]">
  </label>
  <label>Email:
    <input type="email" name="user[email]">
  </label>
  <label>Age:
    <input type="number" name="user[age]">
  </label>
  <label>Password:
    <input type="password" name="user[password]">
  </label>
  <input type="submit" value="Sign Up">
</form>
```

action attribute of the form element defines the url that the request is made to

- absolute URL - https://www.wellsfargo.com/transfers
- relative URL - /users - localhost:3000/users
- no URL - form will be sent to the same page(url) the form is present on

method attribute defines how the data will be sent

if method "post" is used, form data is appended to the body of the HTTP request

the server receives a string that will be parsed in order to get the data
as a list of key/value pairs

only the `get` and `post` methods may be specified on the form. in order to do `put`, `delete`, or other methods, we must us `AJAX` requests using the `fetch` API, for example.

2. Create an HTML form using the Pug template engine

```
form(method="post" action="/users")
  input(type="hidden" name="_csrf" value=csrfToken)
  label(for="username") Username:
  input(type="username" id="username" name="username" value=username)
  label(for="email") Email:
  input(type="email" id="email" name="email" value=email)
  label(for="age") Age:
  input(type="age" id="age" name="age" value=age)
  label(for="password") Password:
  input(type="password" id="password" name="password")
  input(type="submit" value="Sign Up")
```

3. Use express to handle a form's POST request

```
// localhost:3000/about
```

```
app.get(`/`, csrfProtection, (req, res) => {
  res.render(`index`, {
    title: `User List`,
    users, errors: [],
    csrfToken: req.csrfToken(),
  });
});

const users = new Array();

app.post(`/users`, [csrfProtection, checkFields], (req, res) => {
  if (req.errors.length >= 1) {
    res.render(`index`, { errors: req.errors, users, });
    return
  }
  const { username, email, password, } = req.body;
  const user = { username, email, password, };
  users.push(user);
  res.redirect(`/`);
});
```

4. Use the built-in express.urlencoded() middleware function to parse incoming request body form data

```
app.use(express.urlencoded({
  extended: true,
}));
```

5. Explain what data validation is and why it's necessary for the server to validate incoming data

- username - limit number of chars
- password - special char requirement and/or minimum length
- email - valid email

Data validation is the process of ensuring that the incoming data is correct.

Even though you could add add validations on the client side, client-side
validations are not as secure and can be circumvented. Because client-side validations can be circumvented, it's necessary to implement server-side data validations.

Handling bad request data in our route handlers allows us to return 400 level messages with appropriate messaging to allow the user to correct their bad request. Versus allowing the bad data go all the way to the database, and the system returning the generic "500: Internal Server Error".

6. Validate user-provided data from within an Express route handler function

7. Write a custom middleware function that validates user-provided data

8 .Use the csurf middleware to embed a token value in forms to protect against Cross-Site Request Forgery exploits

```
const csrf = require("csurf");
const csrfProtection = csrf({ cookie: true })

app.use((req, res, next) => {
  req.errors = [];
})

const checkFields = (req, res, next) => {
  const { username, email, password } = req.body;
  // Alternatively:
  // const username = req.body.username
  // const email = req.body.email
  // const password = req.body.password
  if (!username || !email || !password) {
    req.errors.push(`you are missing required fields`);
  }

  next();
};
```

```
app.get(`/`, csrfProtection, (req, res) => {
  res.render(`index`, { title: `User List`, users, errors: [], csrfToken: req.csrfToken() });
});

const users = new Array();

app.post(`/users`, [csrfProtection, checkFields], (req, res) => {
  if (req.errors.length >= 1) {
    res.render(`index`, { errors: req.errors, users });
    req.errors = [];
    return
  }
  const { username, email, password, } = req.body;
  const user = { un: username, email, password, };
  users.push(user);
  res.redirect(`/`);
});
```

Data-Driven Web Sites Objectives

1. Use environment variables to specify configuration of or provide sensitive information for your code

What environment variables are and how to access them in Node.js

Environment variables are global variables that are part of your unix shell.
You can access these variables inside of Node.js by using the built-in
`process.env` object.

For instance, assume we have the following script named `test.js`

```
console.log(process.env.NODE_ENV);
```

And we run it like this, setting an environment variable `inline` on the command prompt.

```
NODE_ENV=production node test.js
```

We would expect "production" to be printed.

We can now store sensitive information in these variables

Assuming we have an `app.js` that tries to do something with a secret password

```
// Somewhere inside app.js
const mySecretPassword = process.env.MY_SECRET_PASSWORD;
// Now we have access to the secret password
// without checking it into our code!
```

We can pass in the secret password at runtime.

```
MY_SECRET_PASSWORD=zweeble node app.js
```

This isn't super convenient though, so that's where the next learning objective comes in….

2. Use the dotenv npm package to load environment variables defined in an .env file

dotenv is a npm package that is designed to read values from a `.env` file and populate them as *environment variables*. In this way we can keep the variables in a file instead of having to put them on the command line. We can then put the `.env` file into our `.gitignore` file so we don't check it in to git and github.

A typical `.env` file might look like this:

```
PORT=8080
DB_USERNAME=mydbuser
DB_PASSWORD=mydbuserpassword
DB_DATABASE=mydbname
DB_HOST=localhost
```

You can install the dotenv package using this command

```
npm install dotenv --save-dev
```

There are **three** ways to use `dotenv`.

Using dotenv in our JavaScript code

```
// app.js
// Just requiring dotenv will cause it to
// Load the environment variables from the .env file
require(`dotenv`).config();
```

Using dotenv as a module with `node` or `nodemon`

The `-r` command line option to node (or nodemon) makes node require in the module BEFORE it runs our script.

```
node -r dotenv/config app.js
```

We can also use this inside package.json in the `scripts` section:

```
"scripts": {
  "start": "node -r dotenv/config app.js"
```

Using dotenv from the command line

This requires that you install the following package:

`npm install dotenv-cli --save-dev`

You can now run the dotenv command in front
of any other unix command and it'll populate the environment.

For instance to use it with the sequelize command line:

```
npx dotenv sequelize-cli db:migrate
```

3. Recall that Express cannot process unhandled Promise rejections from within route handler (or middleware) functions

If you have a route like this with an asynchronous callback function

```
app.get("/", async (req, res) => {
   // And some code here throws an error
});
```

If the code inside the callback throws an error, then you will get an "Unhandled Promise Rejection" error from express in the console and the web server will hang.

4. Use a Promise catch block or a try/catch statement with async/await to properly handle errors thrown from within an asynchronous route handler (or middleware) function

To properly handle this, you must catch the error and call "next()" passing it then error. Then express will know it was an error and handle it gracefully.

Using async/await

```
app.get('*', async (req, res, next) => {
 try {
   // Assume some command is here that throws an error
 } catch (err) {
   // We catch it here to make express happy
   next(err);
 }
});
```

Using `.then/.catch`

```
app.get('*', async(req, res, next) => {
  someAsyncFunction().then(() => {
    // Assume some command is here that throws an error
  })
  .catch(err => {
    // We catch it here to make express happy
    next(err);
  })
});
```

5. Write a wrapper function to simplify catching errors thrown within asynchronous route handler (or middleware) functions

Here's the super shortened version:

```
const asyncHandler = (handler) => (req, res, next) => handler(req, res, next).catch(next);
```

We can expand this to better understand what it's doing, let's remove the arrow functions and put back all the curly braces.

```
function asyncHandler(handler) {
  // This is going to be the new function that express will call in the route
  return function(req, res, next) {
    // This will call the original function we passed into `asyncHandler`
    return handler(req, rew, next).catch(function (err) {
      // When there's an error it gets caught and we call next with it
      // to make express happy
      next(err);
    });
  }
}
```

We can now use this helper function in routes like this:

```
// Notice that asyncHandler is being passed an arrow function, so this
// gets called when our server starts up, NOT when a request arrives.
app.get("/", asyncHandler((req, res) => {
  // In here:
  // We call some async function that might return an error, but it's okay
  // because we wrapped it in our async handler that catches them and calls
  // next();
}))
```

6. Use the morgan npm package to log requests to the terminal window to assist with auditing and debugging

install morgan:

`npm install morgan`

Then add it as middleware to your `app.js`

```
const morgan = require('morgan');

app.use(morgan('dev'));
```

And it magically logs your requests to the console!

7. Add support for the Bootstrap front-end component library to a Pug layout template

CSS into the head of the pug template

Follow the directions on the bootstrap site, but convert the HTML into pug.

```
link(rel='stylesheet'
    href='https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css'
    integrity='sha384-Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9Ifjh'
    crossorigin='anonymous')
```

JS into bottom of the pug template

```
script(src="https://code.jquery.com/jquery-3.5.1.slim.min.js"
       integrity="sha384-DfXdz2htPH0lsSSs5nCTpuj/zy4C+OGpamoFVy38MVBnE+IbbVYUew+OrCXaRkfj" crossorigin="anonymous")
script(src="https://cdn.jsdelivr.net/npm/popper.js@1.16.1/dist/umd/popper.min.js"
       integrity="sha384-9/reFTGAW83EW2RDu2S0VKaIzap3H66lZH81PoYlFhbGU+6BZp6G7niu735Sk7lN"
       crossorigin="anonymous")
script(src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"
       integrity="sha384-B4gt1jrGC7Jh4AgTPSdUtOBvfO8shuf57BaghqFfPlYxofvL8/KUEfYiJOMMV+rV"
       crossorigin="anonymous")
```

8. Install and configure Sequelize within an Express application.

Install sequelize like normal

`npm install sequelize@^5.0.0 pg@^8.0.0`

`npm install sequelize-cli@^5.0.0 --save-dev`

A .sequelizerc is mostly optional, it changes the folder structure sequelize uses and also allows us to have our `config.json` be `database.js` so we can use it as a JavaScript file instead of a JSON file. (Which means we can now use process.env to grab our environment variables)

```
// .sequelizerc

const path = require('path');

module.exports = {
  'config': path.resolve('config', 'database.js'),
  'models-path': path.resolve('db', 'models'),
  'seeders-path': path.resolve('db', 'seeders'),
  'migrations-path': path.resolve('db', 'migrations')
};
```

Now we need to setup our database and our user

```
create database reading_list;
create user reading_list_app with encrypted password '«a strong password for the reading_list_app user»';
grant all privileges on database reading_list to reading_list_app;
```

Now we can initialize sequelize

```
$ npx sequelize init
```

And create a `.env` file to hold our environment variables

```
// env
PORT=8080
DB_USERNAME=reading_list_app
DB_PASSWORD=«the reading_list_app user password»
DB_DATABASE=reading_list
DB_HOST=localhost
```

We create a config/index.js to hold any of our express configuration, including our sequelize configuration values.

```
// ./config/index.js

module.exports = {
  environment: process.env.NODE_ENV || 'development',
  port: process.env.PORT || 8080,
  db: {
    username: process.env.DB_USERNAME,
    password: process.env.DB_PASSWORD,
    database: process.env.DB_DATABASE,
    host: process.env.DB_HOST,
  },
};
```

Now we can make sequelize's `database.js` config file require our main express `config/index.js` file to get the values it needs. (This replaces the old config.json file we used to use with sequelize)

```
// ./config/database.js

const {
  username,
  password,
  database,
  host,
} = require('./index').db;

module.exports = {
  development: {
    username,
    password,
    database,
    host,
```

```
    dialect: 'postgres',
  },
};
```

9. Use Sequelize to test the connection to a database before starting the HTTP server on application startup

```
#!/usr/bin/env node

const { port } = require('../config');

const app = require('../app');
const db = require('../db/models');

// Check the database connection before starting the app.
db.sequelize.authenticate()
  .then(() => {
    console.log('Database connection success! Sequelize is ready to use...');

    // Start listening for connections.
    app.listen(port, () => console.log(`Listening on port ${port}...`));
  })
  .catch((err) => {
    console.log('Database connection failure.');
    console.error(err);
  });
```

10. Define a collection of routes (and views) that perform CRUD operations against a single resource using Sequelize

Reminder, CRUD means "Create, Read, Update, Delete"

A Resource is an entity within your application, usually this is directly related to a model or a table in your database.

11. describe how an Express.js error handler function differs from middleware and route handler functions

Express middleware functions define three parameters (req, res, next) and route handlers define two or three parameters (req, res, and optionally the next parameter):

```
// Middleware function.
app.use((req, res, next) => {
  console.log('Hello from a middleware function!');
  next();
});

// Route handler function.
app.get('/', (req, res) => {
  res.send('Hello from a route handler function!');
});
```

Error handling functions look the same as middleware functions except they define four parameters instead of three—err, req, res, and next:

```
app.use((err, req, res, next) => {
  console.error(err);
  res.send('An error occurred!');
});
```

12. Define a global Express.js error-handling function to catch and process unhandled errors

Since this is to catch and process 'unhandled' errors, it needs to be the very last error handler in your application.

```
// Custom error handler.
app.use((err, req, res, next) => {
  console.error(err);
  res.status(err.status || 500);
  res.send('An error occurred!');
});
```

13. Define a middleware function to handle requests for unknown routes by returning a 404 NOT FOUND error

First we setup a middleware to catch everything that routes didn't match.
This should be the last middleware before we start defining error handlers

It will just create a `new Error` and set the status to 404, so that we can catch and handle it in the error handlers later.

```
app.use((req, res, next) => {
  const err = new Error('The requested page couldn\'t be found.');
  err.status = 404;
  next(err);
});
```

Then somewhere in our error handlers, we check to see if the status is 404 and if it is, we render a custom template for "Page not Found". Don't forget to call `next(err)` and pass the error on down to other error handlers!

```
// Error handler for 404 errors.
app.use((err, req, res, next) => {
  if (err.status === 404) {
    res.status(404);
    res.render('page-not-found', {
      title: 'Page Not Found',
    });
```

```
  } else {
    next(err);
  }
});
```

Week 12 Study Guide

Table of Contents

Authentication Learning Objectives

Define the term authentication

Authentication = Who

Authentication is verifying a user is who they say they are. This involves
the user having a *secret* or *credential* of some kind and supplying that to the server
and having the server verify that secret.

This differs from Authorization.

Authorization = What

Authorization is *what* Role or Permissions the user has.

Types of Secrets

- **password** - Usually entered directly by the user
- **token** - Like a password but generated by the server and stored on the client
  so that the client can continue to make requests on the user's behalf.
  Examples of this include OAUTH tokens and JWT Tokens.
- **session id** - Similar to a token, this is used by the browser to keep track
  of a user's session with the server, thus verifying that the user is the
  correct user. (This isn't enough on it's own to authenticate, you must use a
  token or password first). But it still should be protected just like a
  password or token.
- **two factor authentication code** - A code a user enters in addition to
  their password, can be sent via SMS, Email or generated by some software
  on the user's device.

Describe the difference between asymmetric and symmetric cryptographic algorithms

- **Symmetric Encryption** - A single *private* key is used to encrypt some data
  using a lossless algorithm. You can use the *private* key to decrypt and
  restore the original data

  Examples:

  - 1Password or LastPass encryption
  - Encrypting your hard drive with Filevault
    or Bitlocker.
  - Using GNUPG to encrypt a file.

- **Asymmetric Encryption** - A system with two keys, a *private* key
  and a *public* key. If Bob wants to send an encrypted message to Alice then
  Bob uses his *private* key and Alice's *public* key to encrypt the message.
  Now not even Bob can decrypt the message. The only way to decrypt the message
  is for Alice to use her *private* key anb Bob's *public* key.

  Examples:

  - Web Servers use SSL (Secure Socker Layer) to create an encrypted
    communication channel between the client's *private* key and the web server's

public key.
    - GNUPG can be used to send encrypted Emails to other users as long as you have that user's public key.
    - SSH (Secure Shell) creates an encrypted connection between a client and server to provide remote access (github also uses this for SSH git access)

Identify "strong" vs. "broken" hash functions

Hash functions are algorithms that generate a lossy *hash* of a value such that the original value cannot be recovered if someone has the *hash*

A Salt is a random value added to a password *before* hashing and stored alongside the password in the database. This makes rainbow tables of pre-computed hashes less useful for figuring out the original password.

Broken hash functions

Currently these hash functions are "broken", meaning they should not be used to store user passwords in a database at rest.
(Either because the algorithm has been solved, or the algorithm has been shown to have a flaw)

1. MD5
2. SHA-1

Strong hash functions

These are currently "strong" hash functions:

1. PBKDF2
2. bcrypt
3. Argon2

Neither of these are an exhaustive list, but these are common ones available for use with NodeJS libraries and covered in this course.

It's still important to protect the hashes in your database because if they are leaked, they can be attacked in several different ways.

- *Brute force* - This is just where you try to hash every possible combination of characters until you get a matching hash. This is slow.
- *Rainbow Attack* - You use a list of pre-computed hashes (A Rainbow Table) and compare the hash you want to crack against it. This is still slow but faster than the brute force attack

Implement session-based authentication in an Express application

Session based authentication is where you use a session-id, stored in a cookie to authenticate a user once they have first authenticated with a password.

This is the package we use for express session authentication:

```
npm install express-session
```

We configure it as express middleware like so.

```
app.use(session({
  secret: 'a5d63fc5-17a5-459c-b3ba-6d81792158fc', // Secret used to verify the session-id
  re`sa`ve: false, // Recommended by the express-session package
  saveUninitialized: false, // Recommended by the express-session package
}));
```

> The secret should probably be stored in an *environment variable* in your `.env` file instead of hard coded like this.

The middleware creates a `.session` property on the express `req` (Request) object. You can use this to store information you would like to persist for the user, such as the user's `id`, or other information that is relevant for your application.

By default `express-session` stores the session data in memory. Which means if you restart your express server you are going to lose your session data (all users will suddenly be logged out of the sytem).

To remedy this you can configure a different mechanism for the session store.

We can use the `connect-pg-simple` module to connect to postgreSQL and store the session in a table in our database.

```
const store = require('connect-pg-simple');

app.use(session({
  store: new (store(session))(),
  secret: 'a5d63fc5-17a5-459c-b3ba-6d81792158fc',
  resave: false,
  saveUninitialized: false,
}));
```

It will need an environment variable named `DATABASE_URL` to know which postgreSQL database to connect to.

The postgreSQL url should look something like this:

```
DATABASE_URL=postgresql://<username>:<password>@<host>:<port>/<database name>
```

Implement a strong hash function to securely store passwords

We can use `bcrypt` to store a password hash in the database instead of storing the plain text password, which is insecure.

Installing bcrypt

```
npm install bcryptjs
```

Generating a hashedPassword

Then wherever you register a new user you can add a line like this to generate a hash to store in the `user` table in the database

```
const hashedPassword = await bcrypt.hash(password, 10);
```

Verifying a password against a hashed password

Whenever we need to login a user in, we just check the password they give us against the hashed password from the database like so:

```
const passwordMatch = await bcrypt.compare(password, user.hashedPassword.toString());
```

passwordMatch will be a boolean.

Describe and use the different security options for cookies

- **httpOnly** - Makes the cookie only be able to be read and written by the browser and *not* by JavaScript.
- **secure** - The browser will only send this cookie if the connection is HTTPS.
- **domain** - If this isn't set it will assume the cookie should be the same as the site's domain.
- **expires** - The date and time when the cookie expires
- **maxAge** - The number of milliseconds before the cookie expires (usually easier to use than expires)
- **path** - Defaults to '/' but it's a prefix for the path in the URL that the cookie is valid for (seldom used)

Application Programming Interfaces Objectives

Recall that REST is an acronym for Representational State Transfer

Describe how RESTful endpoints differ from traditional remote-procedure call (RPC) services

For example, with a RESTful API, you would see a GET request to this path to retrieve a specific tweet: `http://localhost/tweets/12`. In an RPC-based API, you could see a path similar to this `http://localhost/getTweetById?id=12`. You need comprehensive documentation to know all of the different methods available by an RPC-designed API. You immediately know how to perform the basic CRUD (Create, Read, Update, Delete) operations on RESTful resources by using the HTTP verbs.

In RESTful APIs the URIs are the nouns while the HTTP methods are the verbs.

In RPC based APIs, you are calling a remote *function* on the server from the client, so you will see a method name and parameters in the request or URI.

Identify and describe the RESTful meanings of the combinations of HTTP verbs and endpoint types for both HTML-based applications and APIs

HTTP Verbs

- `GET` - Get a representation of resource
- `POST` - Create a new resource
- `PUT` - Update a resource with complete data
- `PATCH` - Update a resource with partial data
- `DELETE` - Delete a resource

Resources

- `/tweets` would be a *collection resource*
- `/tweets/17` would be a *single resource*

Given a data model, design RESTful endpoints that interact with the data model to define application functionality

Given the table:

Tweets
id
message
createdAt
updatedAt

These might be some good RESTful endpoints for an API (These would return JSON)

Path	HTTP Verb	Meaning
/api/tweets	GET	Get an list of your tweets
/api/tweets	POST	Create a new tweet
/api/tweets/17	GET	Get the details of a tweet with the id of 17
/api/tweets/17	PUT/PATCH	Update a tweet with the id of 17
/api/tweets/17	DELETE	Delete a tweet with the id of 17

While these might be good RESTful endpoints for a frontend (These would return HTML)

Path	HTTP Verb	Meaning
/tweets	GET	Show a page of tweets
/tweets/new	GET	Show a form to add a new tweet
/tweets	POST	Create a new tweet
/tweets/17	GET	Show a page displaying a single tweet
/tweets/17/edit	GET	Show a form to edit a tweet with the id of 17
/tweets/17	PUT/PATCH	Update a tweet with the id of 17
/tweets/17	DELETE	Delete a tweet with the id of 17

Recall that RESTful is not a standard (like ECMAScript or URLs), but a common way to organize data interactions

The reason we call it *'RESTful'* is because it is more of an idea or a set of guidelines for building APIs which conform nicely with how the web already works.

Explain how RESTful APIs are meant to be stateless

This means that there is no necessary session between the client and the server. Data received from the server can be used by the client independently. This allows you to have short discrete operations. Luckily, this is a natural fit for HTTP operations in which requests are intended to be independent and short-lived.

Use the express.json() middleware to parse HTTP request bodies with type application/json

Since you are creating an API working with JSON data, you'll want to update your app.js file to have your application use the express.json() middleware. The express.json() middleware is needed to automatically parse request body content formatted in JSON so that it is available via the req.body property.

```
app.use(express.json());
```

Determine the maximum data an API response needs and map the content from a Sequelize object to a more limited data object

Instead of this which will return everything about the user, including the hashed password…

```
app.get('/users/:id', asyncHandler(async (req, res) => {
  const user = await User.findByPk(req.params.id);
  res.json(user);
}));
```

We can build a new POJO to return instead of the user object like this:

```
app.get('/users/:id', asyncHandler(async (req, res) => {
  const user = await User.findByPk(req.params.id);
  // A new POJO containing only user.id and user.email
  const userData = {
    id: user.id,
    email: user.email
  };
  res.json(userData);
}));
```

If you have a resource that returns a *collection* you can use map in a handy way to build a new POJO for each model object including only the info you want.

```
app.get('/users', asyncHandler(async (req, res) => {
  const users = await User.all()
  // A new array of POJOs containing only the user.id ans user.email fields
  const usersData = users.map(user => {
    id: user.id,
    email: user.email
  });
  res.json(usersData)
}));
```

Define a global Express error handler to return appropriate responses and status codes given a specific Accept header in the HTTP request

> Note: we didn't do this in class, so you aren't required to know this, but here's an example you can use for projects.

You would need to do this if your application is a *single* express server with both a JSON API and Pug templates that return HTML.

```
// Catch unhandled requests and forward to error handler.
app.use((req, res, next) => {
  const err = new Error("The requested resource couldn't be found.");
  err.status = 404;
  next(err);
});

// Error handler that returns the correct type of data
// based on the `Accept` header
app.use((err, req, res, next) => {
  res.status(err.status || 500);
  const acceptHeader = req.get("Accept");

  const errorData = {
    title: err.title || "Server Error",
    message: err.message,
    stack: isProduction ? null : err.stack
  }

  if (acceptHeader === 'text/html') {
    res.render('error-page', errorData)
  } else if (acceptHeader === 'application/json') {
    res.json(errorData);
  } else {
    res.send("Server Error");
  }
});
```

Define Cross-Origin Resource Sharing (CORS) and how it is implemented in some Web clients

By default JavaScript running in the browser can only access API resources from the same origin (i.e. domain, protocol, port).

CORS works by setting HTTP Headers on the API endpoint which tell the browser if a particular domain, protocol, port combination is allowed to access that resource on the API.

Configure your API to use CORS to prevent unauthorized access from browser-based Web requests

```
npm install cors
```

This would allow an application running on localhost port 4000 to access this API

```
app.use(cors({ origin: "http://localhost:4000" }))
```

> You probably want to store the CORS origin in an environment variable in your `.env` file instead of hard coding it.

API Security Objectives

Explain the fundamental concepts of OAuth as a way to authenticate users

Oauth 2.0 is a standardized mechanism for authenticating users using a 3rd party Oauth provider.

Describe the workflow of OAuth resource owner password credentials grant (RFC 6749 Section 4.3)

1. Your application requests authorization from the person
2. The person informs the authorization server that they want to issue an authorization grant for your application
3. The application uses the authorization grant with its secret information to request an access token
4. The authorization server returns an access token
5. Your application uses the access token to gain access to the user's data from the service API
6. The service API returns the resource that the token allows access to

Describe the components of a JSON Web Token (JWT) and how it is constructed

JWT can be used for *authentication* and *authorization*, but they can't be used to decrypt data.

- *header* - Contains basic info about the JWT such as the `typ` (type) and `alg` (algortihm)
- *payload* - Contains any extra info you would like to store inside the JWT
- *signature* - a digital signature created using a hashing algorithm used to verify that the payload hasn't been tampered with. It also verifies the origin of the JWT.

The JWT token consists of a single string with these three parts separated by the `.` character. Each part of the token is Base64 encoded.

Week 14 Study Guide

Table of Contents

Getting Started with React - EOD Lecture Notes

Explain how React uses a tree data structure called the "virtual DOM" to model the DOM

The Virtual DOM is an in-memory tree representation of the browser's Document Object Model. React's philosophy is to interact with the Virtual DOM instead of the regular DOM for developer ease and performance.

By abstracting the key concepts from the DOM, React is able to expose additional tooling and functionality increasing developer ease.

By trading off the additional memory requirements of the Virtual DOM, React is able to optimize for efficient subtree comparisons, resulting in fewer, simpler updates to the less efficient DOM. The result of these tradeoffs is improved performance.

Supporting performant Virtual DOM comparisons informs a key aspect of the React philosophy:

> In our experience, thinking about how the UI should look at any given moment, rather than how to change it over time, eliminates a whole class of bugs.

Use React.createElement to create virtual DOM nodes

Every Virtual DOM Node is eventually constructed using the `React.createElement` function.

`createElement` accepts three arguments, which represent the three key components of every Virtual DOM Node, it's `type`, it's `props`, and it's `children` (i.e. it's contents).

```
React.createElement(
    type, // Either an html element "h1", _or_ a React Component `Component`
```

```
    props, // These are _static_ properties && any html attributes
    children // An array of Virtual DOM Nodes, or a string of content
    );
```

Use ReactDOM.render to have React render your virtual DOM nodes into the actual Web page

`ReactDOM.render` is a simple function which accepts 2 arguments: what to render and where to render it:

```
ReactDOM.render(
    component, // The react component to render
    target // the browser DOM node to render it to
    );
```

Use JSX to create virtual DOM nodes

Describe how JSX transforms into React.createElement calls

JSX is a special format to let you construct virtual DOM nodes using familiar HTML-like syntax. You can put the JSX directly into your `.js` files, however you must run the JSX through a pre-compiler like Babel in order for the browser to understand it.

```
const Clock = (props) => {
    return (
        <div>
            <h1>Clock</h1>
            <div className='clock'>
                <p>
                    <span>
                        Time:
                    </span>
                    <span>
                        {props.hours}:{props.minutes}:{props.seconds}
                    </span>
                </p>
            </div>
        </div>
    );
}
```

Here we initialize a `Clock` component using JSX instead of `React.createElement`.

Using Babel this code is compiled to a series of recursively nested `createElement` calls:

```
const Clock = (props) => {
    return React.createElement(
        "div",
        null,
        [
            React.createElement("h1", null, "Clock"),
            React.createElement("div", {className: "clock"}, [
                React.createElement("p", null, [
                    React.createElement("span", null, "Time:"),
                    React.createElement("span", props, "{props.hours}:{props.minutes}:{props.seconds}")
                ])
            ])
        ]
    );
}
```

JSX is a convenience syntax, but not magic.

Use Array#map to create an array of virtual DOM nodes while specifying a unique key for each created virtual DOM node

Since Array.prototype.map is often used to change an array of values into another array of values of equal length, it's a perfect way to convert an array of *data* or *state* into a list of React Components.

There's one *gotcha* when doing this, though. In order for React to keep track of which components in the list it's rendered and which ones it needs to re-render when the data changes, you **MUST** provide a **unique** `key` attribute to the rendered Components.

```
const rootDiv = document.getElementById("root");

const StarTrekCard = (props) => {
    return (
        <div className="card">
            <div className="card-image">
                <figure className="image">
                    <img src={props.imgUrl} />
                </figure>
            </div>
            <div className="card-content">
                <div className="content">
                    {props.content}
                </div>
            </div>
        </div>
    );
}

const StarTrekCardDeck = (props) => {
    return (
        <div>
            // Here we are mapping the cards into <StarTrekCard> components
            {props.cards.map(card =>
                <StarTrekCard
                    imgUrl={card.imgUrl}
                    content={card.content}
```

```
                    key={card.name} />
            )}
        </div>
    );
}

const cards = [
  {
    name: "Martok",
    imgUrl: "http://guide.fleetops.net/images/avatars/martok.png",
    content: "Ferocious Klingon"
  },
  {
    name: "Mijural",
    imgUrl: "http://guide.fleetops.net/images/avatars/mijural.png",
    content: "Shrike Class Romulan"
  }
  ];

ReactDOM.render(
    <StarTrekCardDeck cards={cards} />,
    rootDiv
)
```

React Class Components Objectives

Create a simple React application by removing items and content from a project generated by the Create React App default template

To generate a React project

```
npx create-react-app my-app-name
```

Files you can delete

Remove the following files from the public folder:

- favicon.ico
- robots.txt
- logo192.png
- logo512.png
- manifest.json

You can replace all the content of index.html with the boilerplate HTML generated by the html:5 command. Don't forget to add a div inside body with an id of "root".

Remove the following files from the src folder:

- App.css
- App.test.js
- logo.svg
- serviceWorker.js
- setupTests.js

Create a simple React application using a custom Create React App template

To use aA's custom template:

```
npx create-react-app my-app-name --template @appacademy/simple
```

Create a React component using ES2015 class syntax

```
import React from 'react'

class Hello extends React.Component {
    constructor() {
        super();
        this.state = {
            name: this.pickRandomName()
        }
    }

    pickRandomName() {
        const randomNum = Math.floor(Math.random() * 3);
        const names = ['LeBron', 'Messi', 'Serena']
        return names[randomNum];
    }

    changeName = () => {
        this.setState({
            name: this.pickRandomName()
        });
    }
    render() {
        return (
            <>
                <h1>Hello {this.state.name}!</h1>
                <button onClick={this.changeName}>Change name</button>
            </>
        )
    }
}

export default Hello;
```

Describe when it's appropriate to use a class component

Since the release of React Hooks (which we will learn about next week) any advantages for class based components have evaporated, so this Learning

Objective does not make a lot of sense anymore and you can safely ignore it.

Initialize and update state within a class component

Initialize the state inside the component's constructor:

```
this.state = {
    name: '',
    email: '',
    password: ''
}
```

The update can happen inside any instance method except for render():

```
handleEmail = (e) => {
    this.setState({
        email: e.target.value
    });
}
```

Provide default values for a class component's props

```
import React from 'react'

class Books extends React.Component {
    render() {
        return (
            <ul>
                {this.props.books.map(book => {
                    return (
                        <li key={book.title}>
                            {book.title} by {book.author}
                        </li>
                    );
                })}
            </ul>
        )
    }
}

Books.defaultProps = {
    books: [
        {title: "Don Quixote", author: "Miguel De Cervantes"},
        {title: "Pedro Paramo", author: "Juan Rulfo"}
    ]
};

export default Books;
```

Add event listeners to elements

```
render() {
    return (
        <form onSubmit={this.handleSubmit}>
            <input type="text" value={this.state.value}/>
            <button>Submit!</button>
        </form>
    )
}
```

Prevent event default behavior

```
handleSubmit = e => {
    e.preventDefault();
    const inputVal = this.state.value;
    // this.props.submitForm is an example of a method passed
    // as part of the props. It is not a built-in method.
    // The software engineer writing this code would need to write it.
    // Tipically the method will take the response and attach
    // it to the body of an AJAX request
    this.props.submitForm(inputVal);
}
```

Safely use the this keyword within event handlers

We can use the experimental syntax that consist of using a fat arrow function:

```
changeEmail = e => {
    this.setState({
        email: e.target.value
    });
}
```

We can also bind the method inside the constructor:

```
constructor() {
    super();

    this.state = { email: ''};

    this.changeEmail = this.changeEmail.bind(this);
}
```

Describe what the React SyntheticEvent object is and the role it plays in handling events

The SyntheticEvent object mimics the characteristics of the event passed to the callback in an event handler but adding more data to it.

These are two common uses:

```
e.preventDefault();
e.stopPropagation();
```

Create a React class component containing a simple form

```
class Simpleform extends React.Component {
    constructor() {
        super();

        this.state = {
            response: ''
        }
    }

    handleResponse = e => {
        this.setState({
            response: e.target.value
        });
    }

    handleSubmit = e => {
        e.preventDefault();
        this.props.submitForm(this.state.response);
    }

    render() {
        return (
            <form onSubmit={this.handleSubmit}>
                <input
                    onChange={this.handleResponse}
                    type="text"
                    value={this.state.response}
                />
                <button>Submit</button>
            </form>
        );
    }
}
```

Define a single event handler method to handle onChange events for multiple "input" elements

```
onChange = (e) => {
    const { name, value } = e.target;
    this.setState({ [name]: value });
}

render() {
    return(
        <form onSubmit={this.onSubmit}>
            <input
                onChange={this.onChange}
                name='email'
                type="text"
                value={this.state.email}
            />
            <input
                onChange={this.onChange}
                name='password'
                type="text"
                value={this.state.password}
            />
            <button>Submit</button>
        </form>
    );
}
```

Add a "textarea" element to a form

The textarea element in React uses a value attribute instead of inner content. It is handled the same way you would handle an input element. It will be a self-closing tag.

```
<textarea value={this.state.text} onChange={this.onChange}/>
```

Add a "select" element to a form

```
<select name='timezone' onChange={this.onChange} value={this.state.timezone}>
    <option>PST</option>
    <option>CST</option>
    <option>EST</option>
</select>
```

Implement form validations

Create a validate method and invoke it on the submit handler method.

```
validate(spiritAnimal) {
    const validationErrors = [];

    if (!spiritAnimal) {
        validationErrors.push('Please provide a spirit animal');
    }

    return validationErrors;
}

onSubmit = e => {
    e.preventDefault();
```

```
    const { spiritAnimal } = this.state;

    const validationErrors = this.validate(spiritAnimal);

    if (validationErrors.length > 0) {
        this.setState({ validationErrors });
    } else {
        this.props.submitForm(spiritAnimal);

        this.setState({ spiritAnimal: ''});
    }
}
```

Describe the lifecycle of a React component

A React component usually will go through three stages: mounting, updating, and unmounting.

Recall that the commonly used component lifecycle methods include componentDidMount, componentDidUpdate, and componentWillUnmount

When a React component is added to the virtual DOM:

- The constructor method is called.
- The render method is called.
- The DOM gets updated.
- The componentDidMount method is called.

When a React component is updated:

- The render method is called.
- The DOM gets updated.
- The componentDidUpdate method is called.

Just before a React component gets removed from the virtual DOM, the componentWillUnmount method is called.

Use the componentDidMount component lifecycle method to fetch data from an API

You should not fetch data from within the constructor. Use componentDidMount instead.

```
componentDidMount() {
    fetch(url)
        .then(response => response.json())
        .then(data => this.setState({ movieInfo: data }));
}
```

Utilize official documentation to gain an understanding of how new technology works

React Docs

React Router Objectives

Use the react-router-dom package to set up React Router in your applications

```
npm install --save react-router-dom@^5.1.2
```

```
import { BrowserRouter } from 'react-router-dom`;

const App = () => {
  return (
    <BrowserRouter>
      <div> // Note that BrowserRouter can only have a single child Component!
      // Include components here you want to be controlled by the Router, including
      // the <Route> tag
      </div>
    </BrowserRouter>
  );
};
```

Create routes using the `<Route>` component from the react-router-dom package

the `<Route>` Component takes the following arguments:

- `path` - A pattern to match the URL in the browser. You can include placeholders by prefixing them with a `:` character.

  ```
  <Route path='/users/:userId'/>
  <Route path='/users/:userId/stories/:storyId'/>
  ```

- `component` - You usually pass a single component to this attribute. When the path is matched, this component will be rendered and passed props containing `history` `location` and `match` properties.

  Example: When someone visits `/users/1` this will render the Component `UserProfile`

  ```
  <Route path='/users/:userId' component={UserProfile}/>
  ```

- `render` - An alternative to `component` this will cause a component to render, you can pass this an arrow function that renders multiple components. Usually

we use this so we can pass custom props to the component. Using the `component` tag doesn't let us do this.

Example: Assume we want to pass an extra prop called "token" to the user Profile page
We can do it like this, but we need to pass the regular props from `<Route>` down as well by using the spread operator : `{...props}` .

```
<Route
  path='/users/:userId'
  render={props => <UserProfile {...props} token={token} }/>
```

Generate navigation links with the and components from the react-router-dom package

`<Link>` simple renders an anchor tag `<a href=""></a>` that when clicked, changes the `window.location.href` to the url, which triggers `BrowserRouter` to parse the url and look for matching `<Route>` components.

```
<Link to="/">App</Link>
<Link to="/users">Users</Link>
<Link to="/users/1">Andrew's Profile</Link>
```

`<NavLink>` is just like `<Link>` except that it has an optional `activeClassName` and `activeStyle` props that will set a css class or css styles when the `NavLink` is the currently selected hyperlink.

```
<NavLink to="/">App</NavLink>
<NavLink activeClassName="red" to="/users">Users</NavLink>
<NavLink activeClassName="blue" to="/hello">Hello</NavLink>
<NavLink activeClassName="green" to="/users/1">Andrew's Profile</NavLink>
<NavLink to="/" onClick={handleClick}>App with click handler</NavLink>
```

Create `<Switch>` routes and manage the order of rendered components

By default React Router matches paths for the `<Route>` component by treating it as a *prefix*. This means all routes here will match when the url is `/`

```
<Route path='/' ... >
<Route path='/users' ... >
<Route path='/users/:userId' ... >
```

To change this behavior so it doesn't match all three but only the exact matching route you can either change all three to include the `exact` prop set to true, or you can wrap them in a `<Switch>` component which causes only one of the routes inside the to be rendered at any given time. By combining `exact` and `<Switch>` you can solve many complex routing problems.

Remember the first route to match in a `<Switch>` is the winner!

```
<Switch>
  <Route path="/users/:userId" component={(props) => <Profile users={users} {...props} />} />
  <Route exact path="/users" render={() => <Users users={users} />} />
  <Route path="/hello" render={() => <h1>Hello!</h1>} />
  <Route exact path="/" component={App} />
  <Route render={() => <h1>404: Page not found</h1>} />
</Switch>
```

Use the React Router match prop to access router parameters

Given the following `<Route>` :

```
<Route path='/users/:userId/stories/:storyId' component={UserStories}/>
```

If the url were `/users/1/storties/2` then `props.match.params` inside of `UserStories` would look like this:

```
console.log(props.match.params);
// Will log this
// {
//   userId: 1,
//   storyId: 2
// }
```

Use the React Router history prop to programmatically change the browser's URL

`props.history` is one of the props that a `<Route>` sends to your component.

There are two functions on `props.history`

- *push* - Pushes a new url onto the history stack. The user can use the Back button in the browser to go back to the previous url.
- *replace* - Replaces the current url with the new one. The back button in the browser won't work to take you to the previous url.

```
// Pushing a new URL (and adding to the end of history stack):
const handleClick = () => this.props.history.push('/some/url');

// Replacing the current URL (won't be tracked in history stack):
const redirect = () => this.props.history.replace('/some/other/url');
```

Redirect users by using the component in a route

To redirect the user to a different URL (Like say after they've logged in)
use the `<Redirect>` component.

```
<Redirect to="/">
```

Describe what nested routes are and how to create them

Nested routes are when we use a `<Route>` component inside a component
that hasn't been rendered yet instead of creating them in your top level
component directly under `<BrowserRouter>`.

This allows us to keep those routes close to the components they render, and it
means we are dynamically adding routes on the fly.

For instance inside the UserProfile component you might render more routes to the
sub components of that component.

```
// Destructure `match` prop
const UserProfile = ({ match: { url, path, params }) => {

  // Custom call to database to fetch a user by a user ID.
  const user = fetchUser(params.userId);
  const { name, id } = user;

  return (
    <div>
      <h1>Welcome to the profile of {name}!</h1>

      {/* Replaced `/users/${id}` URL with `props.match.url` */}
      <Link to={`${url}/posts`}>{name}'s Posts</Link>
      <Link to={`${url}/photos`}>{name}'s Photos</Link>

      {/* Replaced `/users/:userId` path with `props.match.path` */}
      <Route path={`${path}/posts`} component={UserPosts} />
      <Route path={`${path}/photos`} component={UserPhotos} />
    </div>}
  );
};
```

React Builds Objectives

Describe what frontend builds are and why they're needed

Frontend Builds are where we run a series of programs on the code we write to
transpile (convert) and *bundle* our javascript code.

Frontend Builds are needed for the following reasons:

1. To convert (`Transpile`) code written in newer JavaScript or JSX into code
   that is compatible with certain browsers, even if those browsers do not
   support those features of the language yet.
2. To compress your front end assets (JS, CSS, etc) into a few large files that can
   be loaded into the browser more efficiently. This can use the processes of `Minification` , `Bundling` and `Tree Shaking` to accomplish this.
3. To check your code for errors (`Linting`)

Describe at a high level what happens in a Create React App when you run `npm start`

Create React App runs a development web server when you run `npm start` .

This webserver is controlled by a piece of software called WebPack. The webserver
transpiles and bundles your JS, CSS and other files in memory, and then serves it up
to the browser. By doing it in emory it can support many nice features such as
hot reloading of the code, and nice error reporting in the browser.

Here's the steps that occur:

- Environment variables are loaded
- The list of browsers to support are checked
- The configured HTTP port is checked to ensure that it's available;
- The application compiler is configured and created;
- webpack-dev-server is started;
- webpack-dev-server compiles your application;
- The index.html file is loaded into the browser; and
- A file watcher is started to watch your files, waiting for changes.

Prepare to deploy a React application into a production environment

Set up Environment variables

You can setup the environment variables in a `.env` file just like we did
with express, the only difference is, they must be prefixed with `REACT_APP_`
to be available inside the WebPack build process.

Set up the browser list

You can setup the list of supported browsers in your package.json file. This will
inform Webpack on which rules it should use when transpiling your code.

```
{
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

Build the app

When you get ready to deploy a React application into production, you will run the `npm run build` command, which will use Webpack to bundle and transpile your code into a folder called `build`.

You should not check this folder into source control!

You will deploy this folder onto any webserver (Express, Nginx, Apache etc.). The webserver you use doesn't matter, but it needs to be configured so it serves up the `index.html` file no matter which URL the user visits. This is because React is a Single Page Application (SPA) framework.

React Context Objectives

Context is a way of connecting two components together so they can both access the same state. Think of context like a pipe that connects a Context.Provider to a Context.Consumer. You will still need a regular React component to store the state in, but by adding that state to the Context.Provider and pulling it out with a Context.Consumer you can allow multiple components to access the state without having to pass the piece of state down via nested props.

Use Context to share and manage global information within a React application

Creating the context

You create the Context by using `createContext`. We can put this in it's own module and export the created context object so we can use it in multiple components…

```
// PupContext.js
import { createContext } from 'react';

const PupContext = createContext(); // You can pass arguments here to setup
                                    // a default context if you need to.
export default PupContext;
```

Create a wrapper component with Context.Provider to set a component's default context

Recommended example (Using a 'Provider' component)

Using a custom provider component that can wrap any element, means our provider is reusable. We use `this.props.children` to make it so we can wrap any component with this.

```
import PupContext from './PupContext';
// PuppyProvider.js
class PuppyProvider extends React.Component {
  constructor() {
    super();
    this.state = {
      puppyType: speedy,
    };
  }

  render() {
    return (
      <PupContext.Provider value={this.state}>
        {/* this allows us to wrap this component around any other component */}
        {this.props.children}
      </PupContext.Provider>
    );
  }
}

export default PuppyProvider;
```

In `App.js` we can simply wrap this provider around any part of the app that needs to access the context. The context will be available to any component in the component tree under the Provider. (Any descendant)

```
// App.js
import React from 'react';
import PuppyProvider from './PuppyProvider';
import banana from './pups/banana-pup.jpg'
import sleepy from './pups/sleepy-pup.jpg'
import speedy from './pups/speedy-pup.jpg'

const App = ({ puppyType }) => (
  <PuppyProvider>
    <div id="app">
      <Puppy/>
    </div>
```

```
    </PuppyProvider>
  );
```

Create a wrapper component with Context.Consumer to share the global context through render props

…Somewhere deep inside of the `<Puppy>` component might live a `<PuppyImage>` component…

There are two ways to access the puppyType stored in our Provider. With the static contextType variable, or by wrapping our component with `<PupContext.Consumer>`.

This is by far the simplest way to access context.

`static contextType` and `this.context`

```
import React from 'react';
import PupContext from './PupContext';

class PuppyImage extends React.Component {
  static contextType = PupContext

  render() {
    <img src={this.context.puppyType}/>
  }
}
```

Wrapping a `<PupContext.Consumer>` around our component

You must pass a `Consumer` tag a *function* that receives the value stored in the context.

```
import React from 'react';
import PupContext from './PupContext';

class PuppyImage extends React.Component {
  static contextType = PupContext

  render() {
    return (
      <PupContext.Consumer> {
        puppy => { // This is the function the consunmer will call
                   // so we can get the data from the context provider.
          return (
            <img src={puppy.puppyType}/>
          );
        }
      }
      </PupContext.Consumer>
    );
  }
}
```

This method is handy if you need to access multiple contexts in a single component… Imagine we also need the UserContext to get details on the user of the application as well so we can put a caption under the image.

```
import React from 'react';
import PupContext from './PupContext';
import UserContext from './UserContext';

class PuppyImage extends React.Component {
  static contextType = PupContext

  render() {
    return (
      <UserContext.Consumer> {
        user => {
          <PupContext.Consumer> {
            puppy => {
              return (
                <figure>
                  <img src={value.puppyType}/>
                  <figcaption>{`by ${user.name}`}</figcaption>
                </figure>
              );
            }
          }
          </PupContext.Consumer>
        }
      }
      </UserContext.Consumer>
    );
  }
}
```

As you can see this can get ugly in a hurry, but don't worry, next week we will be learning React Hooks, which will make this much easier and cleaner.

Create and pass a method through Context to update the global state from a nested component

If you take our example PuppyProvider component from earlier, you can see how we can just add a function which updates the state of the PuppyProvider into the PuppyProvider's state.

```
import PupContext from './PupContext';
// PuppyProvider.js
class PuppyProvider extends React.Component {
  constructor() {
    super();
    this.state = {
      puppyType: speedy,
      // We can just add the function to the state
```

```
        // So we can call it from the Consumer
        setPuppyType: this.setPuppyType
      };
    }

    // This function updates the puppyType in the state
    setPuppyType = (puppyType) => {
      this.setState({
        puppyType
      });
    }

    render() {
      return (
        <PupContext.Provider value={this.state}>
          {this.props.children}
        </PupContext.Provider>
      );
    }
  }

  export default PuppyProvider;
```

Since it's now in the Provider's state, it's accessible from the Consumer.

Imagine we have a form that lets you pick a new puppy type.
We just have to call the function we added to the context to set the
puppy type when we submit the form.

```js
import React from 'react';
import PupContext from './PupContext';

class PuppyTypeForm extends React.Component {
static contextType = PupContext;

// Bonus: We can setup the initial state this way
// instead of using a constructor! Fancy!
state = {
chosenPuppyType: ''
}

updateChosenPuppyType = e => {
this.setState({
chosenPuppyType = e.target.value
});
}

handleSubmit = e => {
e.preventDefault();
// This is the magic, we reach into the context and set
// the puppy type by calling the function we stored there.
this.context.setPuppyType(this.state.chosenPuppyType);
}

render() {
return (

{this.chosenPuppyType}/ Update Puppy Type

)
}
}
```

Week 17 Study Guide

Table of Contents

Python Learning Objectives (Day 1)

The None value

This is the same as `null` in javaScript. It represents the lack of existence of a value. You must type it `None` with a capital letter.

Functions that do not return an explicit value, return `None` by default.

```
def print_hello(name):
    """
    This is a function which prints hello to a user, but does not
    return anything.
    """
    print(f"Hello, {name}")

value = print_hello('Bob') # value will be `None`
```

Boolean values

These work the same as in JavaScript, but you **must** capitalize `True` and `False`.

```
a = True
b = False

c = true # This will try to use a variable named `true`!
d = false  # This will try to use a variable named `false`!
```

The logical operators in Python read like English

Javascript	Python
&&	and
||	or
!	not

```
# Logical AND
print(True and True)    # => True
print(True and False)   # => False
print(False and False)  # => False

# Logical OR
print(True or True)     # => True
print(True or False)    # => True
print(False or False)   # => False

# Logical NOT
print(not True)             # => False
print(not False and True)   # => True
print(not True or False)    # => False
```

Truthiness

Everything is `True` unless it's one of these:

- `None`
- `False`
- `''`
- `[]`
- `()`
- `set()`
- `range(0)`

Number values

Integer

```
print(3)         # => 3
print(int(19))   # => 19
print(int())     # => 0
```

Float

```
print(2.24)      # => 2.24
print(2.)        # => 2.0
print(float())   # => 0.0
print(27e-5)     # => 0.00027
```

Type casting

You can convert (cast) numbers in python from one number type to another number type.

```
# Integer to Float
print(17)               # => 17
print(float(17))        # => 17.0

# Float to integer
print(17.0)             # => 17.0
print(int(17.0))        # => 17

# Float and integer to string
print(str(17.0) + ' and ' + str(17)) # => 17.0 and 17
```

Python does not automatically convert types like JavaScript does.

So this is an error

```
print(17.0 + ' and ' + 17)
# TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

Arithmetic Operators

Operator	JavaScript	Python
addition	`+`	`+`
subtraction	`-`	`-`
multiplication	`*`	`*`
division	`/`	`/`
modulo	`%`	`%`
exponent	`Math.pow()`	`**`
integer division		`//`

There is no `++` or `--` in Python.

String values

You can use both `'` and `"` for strings and escaping works the same as it does in JavaScript

```
# Escaping single quote
'Jodi asked, "What\'s up, Sam?"'
```

Triple quotes `'''` can be used for multiline strings.

```
print('''My instructions are very long so to make them
more readable in the code I am putting them on
more than one line. I can even include "quotes"
of any kind because they won't get confused with
the end of the string!''')
```

Both `'''` and `"""` work for these, but convention is to reserve `"""` for multiline comments and function docstrings.

```
def print_hello(name):
    """
    This is a docstring that explains what the function does
    It can be multiple lines, handy!
    You can use any combination of ' and " in these because
    python is looking for the ending triple " characters
    to determine the end.
    """
    print(f"Hello, {name}")
```

Length

```
print(len("Spaghetti")) # => 9
```

Indexing

```
# Normal indexing
print("Spaghetti"[0])    # => S
print("Spaghetti"[4])    # => h

# You can use negative indexes to start at the end.
print("Spaghetti"[-1])    # => i
print("Spaghetti"[-4])    # => e

# return a series of characters
print("Spaghetti"[1:4])  # => pag
print("Spaghetti"[4:-1])    # => hett
print("Spaghetti"[4:4])  # => (empty string)
print("Spaghetti"[:4])  # => Spag
print("Spaghetti"[:-1])    # => Spaghett
print("Spaghetti"[1:])  # => paghetti
print("Spaghetti"[-4:])    # => etti

# indexing past the beginning or end gives an error
print("Spaghetti"[15])    # => IndexError: string index out of range
print("Spaghetti"[-15])    # => IndexError: string index out of range

# but ranges past the beginning or end do not.
print("Spaghetti"[:15])    # => Spaghetti
print("Spaghetti"[15:])    # => (empty string)
print("Spaghetti"[-15:])    # => Spaghetti
print("Spaghetti"[:-15])    # => (empty string)
print("Spaghetti"[15:20])    # => (empty string)
```

String Functions

index

Similar to JavaScript's `indexOf` function

```
print("Spaghetti".index("h"))    # => 4
print("Spaghetti".index("t"))    # => 6
```

count

counts how many times a substring appears in a string

```
print("Spaghetti".count("h"))    # => 1
print("Spaghetti".count("t"))    # => 2
print("Spaghetti".count("s"))    # => 0
print('''We choose to go to the moon in this decade and do the other things,
not because they are easy, but because they are hard, because that goal will
serve to organize and measure the best of our energies and skills, because that
challenge is one that we are willing to accept, one we are unwilling to
postpone, and one which we intend to win, and the others, too.
'''.count('the '))               # => 4
```

Concatenation

You can use the `+` operator just like in JavaScript

```
print("gold" + "fish")    # => goldfish
```

You can use the `*` operator to repeat a string a given number of times

```
print("s"*5)             # => sssss
```

Formatting

```
first_name = "Billy"
last_name = "Bob"
# Using the format function
print('Your name is {0} {1}'.format(first_name, last_name))  # => Your name is Billy Bob
# Using the `f` format flag on the string
print(f'Your name is {first_name} {last_name}') # => Your name is Billy Bob
```

Useful string methods

Value	Method	Result
`s = "Hello"`	`s.upper()`	`"HELLO"`
`s = "Hello"`	`s.lower()`	`"hello"`
`s = "Hello"`	`s.islower()`	`False`
`s = "hello"`	`s.islower()`	`True`
`s = "Hello"`	`s.isupper()`	`False`
`s = "HELLO"`	`s.isupper()`	`True`
`s = "Hello"`	`s.startswith("He")`	`True`
`s = "Hello"`	`s.endswith("lo")`	`True`
`s = "Hello World"`	`s.split()`	`["Hello", "World"]`
`s = "i-am-a-dog"`	`s.split("-")`	`["i", "am", "a", "dog"]`

Method	Purpose
`isalpha()`	returns `True` if the string consists only of letters and is not blank.
`isalnum()`	returns `True` if the string consists only of letters and numbers and is not blank.
`isdecimal()`	returns `True` if the string consists only of numeric characters and is not blank.
`isspace()`	returns `True` if the string consists only of spaces, tabs, and newlines and is not blank.
`istitle()`	returns `True` if the string consists only of words that begin with an uppercase letter followed by only lowercase letters.

Variables

Duck typing

If it looks like a duck and quacks like a duck, then it must be a duck.

Assignment

Just like JavaScript, but there are no special keywords. Scope is block scoped, much like `let` in JavaScript. You can also reassign variables just like `let`.

```
a = 7
b = 'Marbles'
print(a)         # => 7
print(b)         # => Marbles

# You can do assignment chaining
count = max = min = 0
print(count)           # => 0
print(max)             # => 0
print(min)             # => 0
```

Comparison operators

Python uses these same equality operators as JavaScript.

- `>` (greater than)
- `<` (less than)
- `>=` (greater than or equal to)
- `<=` (less than or equal to)
- `==` (equal to)
- `!=` (not equal to)

Precendence in Python:

- Negative signs (`not`) are applied first (part of each number)
- Multiplication and division (`and`) happen next
- Addition and subtraction (`or`) are the last step

Be careful using `not` along with `==`

```
print(not a == b)     # => True
# This breaks
print(a == not b)     # Syntax Error
# This fixes it
print (a == (not b))    # => False
```

Python does short-circuit evaluation

Expression	Right side evaluated?
`True` and …	Yes
`False` and …	No
`True` or …	No
`False` or …	Yes

Assignment operators

`=` is the normal assignment operator.

Python includes these assignment operators as well

- `+=`
- `-=`
- `*=`
- `/=`
- `%=`
- `**=`
- `//=`

Flow-control statements: if, while, for

`if-elif-else`

```
if name == 'Monica':
    print('Hi, Monica.')
elif age < 12:
    print('You are not Monica, kiddo.')
else:
    print('You are neither Monica nor a little kid.')
```

`for`

```
# Looping over a string
for c in "abcdefg":
    print(c)

# Looping over a range
print('My name is')
for i in range(5):
   print('Carlita Cinco (' + str(i) + ')')

# Looping over a list
lst = [0, 1, 2, 3]
for i in lst:
    print(i)

# Looping over a dictionary
spam = {'color': 'red', 'age': 42}
for v in spam.values():
    print(v)

# Loop over a list of tuples and
# Destructuring to values
# Assuming spam.items returns a list of tuples
# Each containing two values (k, v)
for k, v in spam.items():
    print('Key: ' + k + ' Value: ' + str(v))
```

`while`, `break` and `continue`

`while` loop as long as the condition is `True`.

`break` allows you to break out of the loop.

`continue` skips this iteration of the loop and goes to the next iteration.

```
spam = 0
while True:
  print('Hello, world.')
  spam = spam + 1
  if spam < 5:
    continue
  break
```

Functions

You use the `def` keyword to define a function in Python.

```
# Basic function with no arguments and no return value
def printCopyright():
    print("Copyright 2020. Me, myself and I. All rights reserved.")

# Function with positional parameters and a return value
```

```python
def average(num1, num2):
    return (num1/num2)

# Calling it with positional arguments
print(average(6, 2))        # => 3.0

# Calling it with keyword arguments
# (note that order doesn't matter)
print(average(num2=2, num1=6));

# Default parameters
# Here the string "Hello" is the default for `saying`
def greeting(name, saying="Hello"):
    print(saying, name)

greeting("Monica") # => Hello Monica

greeting("Monica", saying="Hi") # => Hi Monica

# A common 'gotcha' is using an mutable object for a default parameter.
# Python doesn't do what you expect. All invocations of the function
# reference the same mutable object

# Everytime we call this we use the exact same `itemList` list
def appendItem(itemName, itemList = []):
    itemList.append(itemName)
    return itemList
print(appendItem('notebook')) # =>  ['notebook']
print(appendItem('pencil')) # =>  ['notebook', 'pencil']
print(appendItem('eraser')) # => ['notebook', 'pencil', 'eraser']
```

Lambdas

In python we have anonymous functions called lambdas, but they are only
a single python statement.

```python
toUpper = lambda s: s.upper()

toUpper('hello') # => HELLO
```

is the same as this in JavaScript

```javascript
const toUpper = s => s.toUpperCase();
toUpper('hello'); // # => HELLO
```

Errors

Unlike JavaScript, if you pass the wrong number of arguments to a function
it will throw an error.

```python
average(1)
# => TypeError: average() missing 1 required positional argument: 'num2'

average(1,2,3)
# => TypeError: average() takes 2 positional arguments but 3 were given
```

Python Learning Objectives (Day 2)

Functions

- `*` - Get the rest of the position arguments as a tuple
- `**` - Get the rest of the keyword arguments as a dictionary

variable length positional arguments

```python
def add(a, b, *args):
    # args is a tuple of the rest of the arguments
    total = a + b;
    for n in args:
        total += n
    return total

# args is None
add(1, 2)  # Returns 3

# args is (4, 5)
add(2, 3, 4, 5) # Returns 14
```

variable length keyword arguments

```python
def print_names_and_countries(greeting, **kwargs):
    # kwargs is a dictionary of the rest of the keyword arguments
    for k, v in kwargs.items():
        print(greeting, k, "from", v)

# kwargs would be:
# {
#   'Monica': 'Sweden',
#   'Charles'='British Virgin Islands',
#   'Carlo':'Portugal
# }
print_names_and_countries("Hi",
                          Monica="Sweden",
                          Charles="British Virgin Islands",
                          Carlo="Portugal")
# Prints
# Hi Monica from Sweden
# Hi Charles from British Virgin Islands
# Hi Carlo from Portugal
```

You can combine all of these together

```
def example2(arg_1, arg_2, *args, kw_1="shark", kw_2="blowfish", **kwargs):
  pass
```

Lists

Lists are mutable arrays.

```
# Can be made with square brackets
empty_list = []
departments = ['HR','Development','Sales','Finance','IT','Customer Support']

# list built-in function makes a list too
specials = list()

# You can use `in` to test if something is in the list
print(1 in [1, 2, 3]) #> True
print(4 in [1, 2, 3]) #> False
```

Dictionaries

Dictionaries are similar to JavaScript POJOs or `Map`. They have key value pairs.

```
# With curlies
a = {'one':1, 'two':2, 'three':3}
# With the dict built-in function
b = dict(one=1, two=2, three=3)

# You can use the `in` operator with dictionaries too
print(1 in {1: "one", 2: "two"})    #> True
print("1" in {1: "one", 2: "two"})  #> False
print(4 in {1: "one", 2: "two"})    #> False
```

Sets

Just like JavaScript's `Set`, it is an unordered collection of distinct objects.

```
# Using curlies (dont' confuse this with dictionaries)
school_bag = {'book','paper','pencil','pencil','book','book','book','eraser'}

# Using the set() built in
school_bag = set('book','paper','pencil','pencil','book','book','book','eraser')

# You can use the `in` operator with sets
print(1 in {1, 1, 2, 3})  #> True
print(4 in {1, 1, 2, 3})  #> False
```

Tuples

Tuples are *immutable* lists of items.

```
# With parenthesis
time_blocks = ('AM','PM')

# Without parenthesis
colors = 'red','blue','green'
numbers = 1, 2, 3

# with the tuple buit-in function which can also be used to
# convert things to tuples
tuple('abc')        # returns ('a', 'b', 'c')
tuple([1,2,3])      # returns (1, 2, 3)

# you can use the `in` operator with tuples
print(1 in (1, 2, 3)) #> True
print(4 in (1, 2, 3)) #> False
```

Ranges

A *range* is simply a list of numbers in order which can't be changed (immutable). Ranges are often used with `for` loops.

A `range` is declared using one to three parameters

- start - optional (`0` if not supplied) - first number in the sequence
- stop - required - next number past the last number in the sequence
- step - optional (`1` if not supplied) - the difference between each number in the sequence

For example

```
range(5)            # [0, 1, 2, 3, 4]
range(1,5)          # [1, 2, 3, 4]
range(0, 25, 5)     # [0, 5, 10, 15, 20]
range(0)            # [ ]
```

Built-in functions: filter, map, sorted, enumerate, zip, len, max, min, sum, any, all, dir

filter

```
def isOdd(num):
    return num % 2
filtered = filter(isOdd, [1,2,3,4])
# It returns a filter iterable object
# but we can cast it to a list
print(list(filtered)) # => [1, 3]
```

map

```
def toUpper(str):
    return str.upper()

upperCased = map(toUpper, ['a','b','c'])

print(list(upperCased)) # => ['A','B','C']
```

sorted

```
sortedItems = sorted(['Banana', 'orange', 'apple'])
print(list(sortedItems)) # => ['Banana', 'apple', 'orange']

# Notice Banana is first because uppercase letters come first

# Using a key function to control the sorting and make it sort
# so the case doesn't matter
sortedItems = sorted(['Banana', 'orange', 'apple'], key=str.lower)
print(list(sortedItems)) # => ['apple', 'Banana', 'orange']

# Reversing the sort
sortedItems = sorted(['Banana', 'orange', 'apple'], key=str.lower, reverse=True)
print(list(sortedItems)) # => ['orange', 'Banana', 'apple']
```

enumerate

```
quarters = ['First', 'Second', 'Third', 'Fourth']
print(enumerate(quarters))
print(enumerate(quarters, start=1))
```

```
(0, 'First'), (1, 'Second'), (2, 'Third'), (3, 'Fourth')
(1, 'First'), (2, 'Second'), (3, 'Third'), (4, 'Fourth')
```

zip

```
keys = ("Name", "Email")
values = ("Bob", "Bob@bob.com")

zipped = zip(keys, values)

print(list(zipped))
# => [('Name', 'Bob'), ('Email', 'Bob@bob.com')]

# You can zip more than two
x_coords = [0, 1, 2, 3, 4]
y_coords = [2, 3, 5, 3, 5]
z_coords = [3, 5, 2, 1, 4]

coords = zip(x_coords, y_coords, z_coords)

print(list(coords))
# => [(0, 2, 3), (1, 3, 5), (2, 5, 2), (3, 3, 1), (4, 5, 4)]
```

len

```
len([1,2,3]) # => 3
len((1,2,3)) # => 3
len({
    'Name': 'Bob',
    'Email': 'bob@bob.com'
}) # => 2
```

Can also work on any object which contains a `__len__` method.

max

```
max(1, 4, 6, 2) # => 6
max([1, 4, 6, 2]) # => 6
```

min

```
min(1, 4, 6, 2) # => 1
min([1, 4, 6, 2]) # => 1
```

sum

```
sum([1,2,3]) # => 6
```

any

```
any([True, False, False]) # => True
any([False, False, False]) # => False
```

all

```
any([True, False, False]) # => False
any([True, True, True]) # => False
```

dir

Returns all the attributes of an object including it's methods and dunder methods

```
user = {
    'Name': 'Bob',
    'Email': 'bob@bob.com'
}

dir(user)

# => ['__class__', '__contains__', '__delattr__', '__delitem__', '__dir__',
#  '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
# '__getitem__', '__gt__', '__hash__', '__init__', '__init_subclass__',
# '__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__',
# '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__setattr__',
# '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'clear', 'copy',
# 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault',
# 'update', 'values']
```

Importing packages and modules

- Module - Python code in a file **or** directory
- Package - A module which is a directory containing a `__init__.py` file.
- submodule - A module which is contained within a package
- name - an exported function, class or variable in a module

Unlike JavaScript, modules export ALL names contained within them without any special `export` keywords

Assuming we have the following package with four submodules

```
math
|   __init.py
|   addition.py
|   subtraction.py
|   multiplication.py
|   division.py
```

if we peek into the addition.py file we see there's an add function

```
# addition.py
# We can import `add` from other places because it's a `name` and is
# AUTOMATICALLY exported
def add(num1, num2):
    return num1 + num2
```

Our `__init__.py` has the following lines:

```
# This imports the `add` function
# and now it's also re-exported in here as well!
from .addition import add
# These import and re-export the rest of the functions from the sub modules
from .subtraction import subtract
from .division import divide
from .multiplication import multiply
```

Remember any names that exist within a module are automatically exported.

> Notice the `.` syntax because this package can import it's own submodules.

So if we have a `script.py`, and we want to import the add function, we could do it lots of different ways

```
# This will load and execute the `math/__init__.py` file
# and give us an object with the exported names in `math/__init.py__`
import math

print(math.add(1,2)) # => 3
```

```
# This imports JUST the add from `math/__init__.py`
from math import add

print(add(1,2)) # => 3
```

```
# This skips importing from `math/__init__.py` (although it still runs)
# and imports directly from the addition.py file
from math.addition import add

print(add(1,2)) # => 3
```

```python
# this imports all the functions individually from `math/__init.py`
from math import add, subtract, multiply, divide

print(add(1,2)) # => 3
print(subtract(2, 1)) # => 1
```

```python
# This imports `add` and renames it to `addSomeNumbers`
from math import add as addSomeNumbers

print(addSomeNumbers(1, 2)) # => 3
```

Python Learning Objectives (Day 3)

Classes, methods, and properties

```python
class AngryBird:
    # Slots optimize property access and memory usage
    # and prevent you from arbitrarily assigning new properties to the instance
    __slots__ = ['_x', '_y']

    # constructor
    def __init__(self, x=0, y=0):
        # Doc string
        """
        Construct a new AngryBird by setting its position to (0, 0).
        """
        ## Instance variables
        self._x = x
        self._y = y

    # Instance method
    def move_up_by(self, delta):
        self._y += delta

    # Getter
    @property
    def x(self):
        return self._x

    # Setter
    @x.setter
    def x(self, value):
        if value < 0:
            value = 0
        self._x = value

    @property
    def y(self):
        return self._y

    @y.setter
    def y(self, new_y):
        self._y = new_y

    # Dunder Repr... called by `print`
    def __repr__(self):
        return f"<AngryBird ({self._x}, {self._y})>"
```

JavaScript to Python Classes cheat table

	Javascript	Python
Constructor	`constructor()`	`def __init__(self):`
Super Constructor	`super()`	`super().__init__()`
Instance properties	`this.property`	`self.property`
Calling Instance Methods	`this.method()`	`self.method()`
Defining Instance Methods	`method(arg1, arg2) {}`	`def method(self, arg1, arg2):`
Getter	`get someProperty() {}`	`@property`
Setter	`set someProperty() {}`	`@someProperty.setter`

List comprehensions

List comprehensions are a way to transform a list from one format to another.

They are a Pythonic alternative to using `map` or `filter`.

Syntax of a list comprehension:

`newList = [value loop condition]`

Using a for loop

```python
squares = []
for i in range(10):
    squares.append(i**2)

print(squares)
# Prints [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

You can change it to a list comprehension

```
# value = i ** 2
# loop = for i in range(10)
squares = [i**2 for i in range(10)]

print(list(squares))
# Prints [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

They can be used with a condition to do what `filter` does

```
sentence = 'the rocket came back from mars'
vowels = [character for character in sentence if character in 'aeiou']

print(vowels)
# Prints ['e', 'o', 'e', 'a', 'e', 'a', 'o', 'a']
```

You can also use them on dictionaries. We can use the `items()` method
for the dictionary to loop through it getting the keys and values out at once.

```
person = {
    'name': 'Corina',
    'age': 32,
    'height': 1.4
}

# This loops through and capitalizes the first letter of all the keys
newPerson = { key.title(): value for key, value in person.items() }
# Prints {'Name': 'Corina', 'Age': 32, 'Height': 1.4}
```

Week 4 Study Guide

Notes

DOM vs BOM

- the `Document Object Model` is the hierarchy/representation of the objects that comprise a document on the web (i.e. how all elements in a document are organized). The DOM is a part of the `Browser Object Model`, the hierarchy/representation of all browser objects associated with the web browser.

Browser Diagram

User Interface
v
Browser Engine ---> Data Persistence
v
Rendering Engine
v v v
Networking JS interpreter UI backend
==========

- `User Interface` : everything in browser interface except requested page content
- `Browser Engine` : manages interactions btwen UI and rendering engine
- `Rendering Engine` : renders requested page content, parsing HTML & CSS
- `Networking` : handles network calls, i.e. HTTP requests
- `JS Interpreter` : parses & executes JS code
- `UI Backend` : used for drawing basic widgets, using operating system UI methods
- `Data Persistence` : persistence of data stored in browser, i.e. cookies

```
- A browser's main role in the request/response cycle is:
    1. Parsing HTML, CSS, JS
    2. Rendering that information to the user by constructing a DOM tree and rendering it.
```

Request-Response Diagram

```
Click --->          Request --->        Server <----> Data
    Browser
Page <---           <--- Response

Your Computer       The Internet        Data Center
```

Window API

- the `Window API` includes the methods and properties that you can use on the `window object`, the core of the `BOM`.
- using the `Window API` to resize the browser window:

```
//opens a new window
newWindow = window.open("url", "name", "width=100, height=100")
//resizes new window
newWindow.resizeTo(500,500)
//also resizes by given amount; use `-` to shrink
newWindow.resizeBy(xDelta, yDelta)
```

- the `context` of an `anonymous function` fun in the browser will be the `window object`. Remember that every function has a context, which we can think of as which object OWNS the function, and context is most often determined by how a function is invoked.

Running Scripts

- Insert a script via a .js document into an .html document:

```
<html>
<head>
    <script type="text/javascript" src="dom-ready-script.js"></script>
</head>
<body></body>
<html></html>
</html>
```

- Run the script on DOMContentLoaded (when the doc has been loaded, but without waiting for stylesheets, images, and subframes):

```
window.addEventListener("DOMContentLoaded", event => {
    console.log("This script loaded when the DOM was ready.");
});
```

- Run the script on page load using window.onload (wait for EVERYTHING to load):

```
window.onload = () => {
    console.log(
        "This script loaded when all the resources and the DOM were ready."
    );
};
```

- Three ways to prevent script from running until page loads:
  1. Use `DOMContentLoaded`
  2. Place `script tag` at very bottom of HTML file
  3. Add attribute like `async` or `defer`

async vs defer

- `<script>` without any attributes will pause HTML parsing, and a request will be made to fetch the file (if it

is external). The script will be executed before parsing is resumed.
- `async` downloads the file during HTML parsing and will pause the HTML parser to execute it once it has downloaded.
- `defer` downloads the file during HTML parsing, but will only execute it after the parser has completed.
- the standard is to use `async`, then `defer`.

Cookies vs Web Storage API

- Cookies - stores stateful info about a user, transfers data to server, under 4KB storage limit
- `sessionStorage` - stores data only for a session, until browser window/tab is closed, does not transfer data to server, 5MB storage limit.
- `localStorage` - stores data w/no expiration date, does not transfer data to server, deleted when browser cache is cleared; maximum storage limit
- Create a cookie:

```
const firstCookie = "favoriteCat=million";
document.cookie = firstCookie;
const secondCookie = "favoriteDog=bambi";
document.cookie = secondCookie;
document.cookie; // Returns "favoriteCat=million; favoriteDog=bambi"
```

- Delete a cookie by setting a cookie's expiration date to the past (or delete in Developer Tools):

```
document.cookie = "favoriteCat=; expires = Thu, 01 Jan 1970 00:00:00 GMT";
  document.cookie; // Returns "favoriteDog=bambi"
```

- Create localStorage data:

```
//set new localStorage item
localStorage.setItem("firstThing", "firstValue");
//retrieve that localStorage item
localSTorage.getItem("firstThing");
```

- When to use the Web Storage API?
  - shopping cart
  - input data on forms
  - info on user i.e. preferences or buying habits
- When to use cookies:
  - Session cookie, stores session info on user login/validation (lost once browser is closed unless you use a persistent cookie)
- You can view cookies and web storage info with Developer Tools (inspect -> Application tab).

Learning Objectives

BROWSER BASICS & STORAGE

1. ✓ Explain the difference between BOM (browser object model) and the DOM (document object model).
2. ✓ Given a diagram of all the different parts of the Browser, identify each part.
3. ✓ Use the Window API to change the innerHeight of a user's window.
4. ✓ Identify the context of an anonymous function running in the Browser (the window).
5. ✓ Given a JS file and an HTML file, use a script tag to import the JS file and execute the code therin when all elements on a page load (using DOMContentLoaded).
6. ✓ Given a JS file and an HTML file, use a script tag to import the JS file and execute the code therein when the page loads.
7. ✓ Identify three ways to prevent JS code from executing until an entire HTML page is loaded.
8. ✓ Label a diagram on the Request/Response cycle.
9. ✓ Explain the Browser's main role in the request/response cycle (1. Parsing HTML, CSS, JS; 2. Rendering that information to the user by constructing a DOM tree and rendering it.)
10. ✓ Given several detractors - identify which real-world situations could be implemented with the Web Storage API (shopping cart, forms savings inputs, etc.).
11. ✓ Given a website to visit that depends on cookies (like Amazon), students should be able to go to that site, add something to their cart, and then delete that cooking using Chrome Developer tools in order to empty their cart.
12. ✓ Write JS to store the value "I ♡ falafel" with the eky "eatz" in the browser's local storage.
13. ✓ Write JS to read the value stored in local storage for the key "paper-trail".

ELEMENT SELECTION

1. Given HTML that includes `<div id="catch-me-if-you-can>HI!</div>` , write a JS statement that stores a reference to the HTMLDivElement with the id "catch-me-if-you-can" in a variable named "divOfInterest".

```
let divOfInterest = document.getElementById("catch-me-if-you-can");
```

2. Given HTML that includes seven SPAN elements each with the class "cloudy", write a JS statement that stores a reference to a NodeList filled with references to the seven HTMLSpanElements in a variable named "cloud

```
let cloudyNodes = document.querySelectorAll("span.cloudy");
```

3. Given an HTML file with HTML, HEAD, TITLE, and BODY elements, create and reference a JS file that in which the JS will create and attach to the BODY element an H1 element with the id "sleeping-giant" with the content "Jell-O, Burled!".

```
<script type="text/javascript" src="location.file"></script>
```

```
let newHeader = document.createElement("h1");
newHeader.setAttribute("id", "sleeping-giant");
newHeader.innerHTML = "Jell-O, Burled!";
//const newContent = document.createTextNode("Jell-O, Burled!");
document.body.appendChild(newHeader);
```

4. Given an HTML file with HTML, HEAD, TITLE, SCRIPT, and BODY elements with the SCRIPT's SRC attribute referencing an empty JS file, write a script in the JS file to create a DIV element with the id "lickable-frog" and add it as the last child to the BODY element.

```
<script type="text/javascript" src="location.file"></script>


let newDiv = document.createElement("div");
newDiv.setAttribute("id", "lickable-frog");
document.body.appendChild(newDiv);
```

5. Given an HTML file with HTML, HEAD, TITLE, SCRIPT, and BODY elements with no SRC attribute on the SCRIPT element, write a script in the SCRIPT block to create a UL element with no id, create an LI element with the id "dreamy-eyes, add the LI as a child to the UL element, and add the UL element as the first child of the BODY element.

```
<script type="text/javascript">
    let newList = document.createElement("ul");
    let newItem = document.createElement("li");
    newItem.setAttribute("id", "dreamy-eyes");
    newList.appendChild(newItem);
    document.body.prepend(newList);
</script>
```

6. Write JS to add the CSS class "i-got-loaded" to the BODY element when the window fires the DOMContentLoaded event.

```
document.addEventListener("DOMContentLoaded", event => {
   document.body.className("i-got-loaded");
});
```

7. Given an HTML file with a UL element with the id "your-best-friend" that has six non-empty ILs as its children, write JS to write the content of each LI to the console.

```
let parentList = document.getElementById("your-best-friend");
let childNodes = parent.childNodes;
for (let value of childNodes.values()) {
    console.log(value);
}
```

8. Given an HTML file with a UL element with the id "your-worst-enemy" that has no children, write JS to construct a string that contains six LI tags each containing a random number and set the inner HTML

property of ul#your-worst-enemy to that string.

```
const getRandomInt = max => {
    return Math.floor(Math.random() * Math.floor(max));
}
const liArr = [];
for (let i = 0, i < 6, i++) {
    liArr.push("<li>" + getRandomInt(10) + "</li>")
}
const liString = liArr.join(" ");
const listElement = document.getElementById("your-worst-enemy")
listElement.innerHTML = liString;
```

9. Write JS to update the title of the document to the current time at a reasonable interval such that it looks like a real clock.

```
<title id="title"></title>


const title = document.getElementById("title");
const time = () => {
    const date = new Date();
    const seconds = date.getSeconds();
    const minutes = date.getMinutes();
    const hours = date.getHours();

    title.innerHTML = `${hours}:${minutes}:${seconds}`
};
setInterval(time, 1000);
```

EVENT HANDLING

1. Given an HTML page that includes
`<button id="increment-count">I have been clicked <span id="clicked-count">0</span> times</button>`
, write JS that increases the value of the content of `span#clicked-count` by 1 every time `button#increment-count` is clicked.

```
let incrementButton = document.getElementById("increment-count");
let incrementSpan = document.getElementById("clicked-count");
let count = 0
incrementButton.addEventListener("click", event => {
    count++
    incrementSpan.innerHTML = count
});
```

2. Given an HTML page that includes
`<input type="checkbox" id="on-off"><div id="now-you-see-me">Now you see me</div>` , write JS that sets the display of div#now-you-see-me to "none" when input#on-off is checked and "block" when input#on-off is not checked.

```
let inputBox = document.getElementById("checkbox");
let divSee = document.getElementById("now-you-see-me");
inputBox.addEventListener("click", event => {
    if (inputBox.checked) {
        divSee.style.display = "block";
    } else {
        divSee.style.display = "none";
    };
});
```

3. Given an HTML file that includes `<input id="stopper" type="text" placeholder="Quick! Type STOP">`, write JS that will change the background color of the page to cyan five seconds after a page loads unless the field input#stopper contains only the text "STOP".

```
let inputStopper = document.getElementById("stopper");
const stopCyanMadness = () => {
    if (inputStopper.value !== "STOP") {
        document.body.style.backgroundColor = "cyan";
    }
};
setTimeout(stopCyanMadness, 5000);
```

4. Given an HTML page that includes `<input type="text" id="fancypants">`, write JS that changes the background color of the textbox to #E8F5E9 when the caret is in the textbox and turns it back to its normal color when focus is elsewhere.

```
const input = document.getElementById("fancypands");
input.addEventListener("focus", event => {
    event.target.style.backgroundColor = "#E8F5E9";
});
input.addEventListener("blur", event => {
    event.target.style.backgroundColor = "initial";
})
```

5. Given an HTML page that includes a form with two password fields, write JS that subscribes to the forms submission event and cancels it if the values in the two password fields differ.

```
let form = document.getElementById("signup-form");
let passwordOne = document.getElementById("password);
let passwordTwo = document.getElementById("password);
form.addEventListener("submit", event => {
    if (passwordOne.value !== passwordTwo.value) {
        event.preventDefault();
        alert("Passwords must match!");
    } else {
        alert("The form was submitted!");
    }
})
```

6. Given an HTML page that includes a div styled as a square with a red background, write JS that allows a user to draf the square around the screen.
    1. Mark the element as draggable:

```
<div id="red-square" draggable="true"></div>
```

7. Given HTML page that has 300 DIVs, create one click event subscription that will print the id of the element clicked on to the console.

```
document.body.addEventListener("click", event => {
    console.log(event.target.id);
})
```

8. Identify the definition of the bubbling principle.
   When an event happens on an element, it first runs the handlers on it, then on its parent, then all the way up on other ancestors.

JSON

1. Identify and generate valid JSON-formatted strings.
   String in JS: `'this is "text"'`
   String in JSON: `"this is \"text\""`
    - use \n for line breaks
    - keys in JSON objects must be surrounded by " quotes
2. Use `JSON.parse` to deserialize JSON-formatted strings.

```
const str = '[1,"hello, \\"world\\"",3.14,{"id":17}]';
console.log(JSON.parse(str));
    // prints an array with the following entries:
    //   0: 1
    //   1: "hello, \"world\""
    //   2: 3.14
    //   3: { id: 17 }
```

3. Use `JSON.stringify` to serialize JS objects.

```
const array = [1, 'hello, "world"', 3.14, { id: 17 }];
console.log(JSON.stringify(array));
    // prints [1, "hello, \"world\"", 3.14, {"id":17}]
```

4. Correctly identify the definition of "serialize".
   Converting your data into a format that can be sent to another computer.
5. Correctly idenfity the definition of "deserialize".
   Getting a message from another computer and converting that message into usable data.

Week 18 Study Guide

Table of Contents

- Reason about the way Alembic orders your migrations; and,
- Handle branching and merging concerns
- Configuring a Flask application to use Alembic;
- Run commands to manage your database through the flask command; and,
- Instead of alembic init…
- Check the help for the rest of the commands, which are the same as Alembic
- Autogenerate migrations from your models!
- Instead of alembic migrate…

Python Unit Testing Objectives

Use the built-in unittest package to write unit tests

- unittest
  - Built in to python
  - Requires that you build a class that inherits from `unittest.TestCase`
  - test functions must start with `test_`
  - Has a collection of assertion functions

Install and use the pytest package to write unit tests

- pytest
  - Has better output than unittest
  - Just requires a test file full of test methods
  - Can also run `unittest` based tests
  - Uses python built in assert keyword

Python Environment Management Objectives

- pyenv
  - Installs versions of python inside your home directory in a `.pyenv` folder
  - Allows you to easily switch between python versions with the `pyenv global` command.
  - Closest Node.JS equivalennt would be `nvm`

Describe pip

- pip
  - Installs Python packages into python's library path folders.
  - Can use a `requirements.txt` file to install a set of packages.
  - Can be used standalone but we used it mostly by levaraging pipenv which uses it under the hood
  - Closest Node.JS equivalent would be `npm install -g`

Describe virtualenv

- virtualenv
  - Creates a virtual installation of python. Uses symbolic links and adjustments to certain environment variables to isolate python packages from one project to another.
  - Can be used standalone but we used it mostly by leveraging pipenv which uses it under the hood
  - No Node.JS equivalent

Demonstrate how to use pipenv to initialize a project and install dependencies

- pipenv
  - Combines `pip` and `virtualenv` into one command.
  - Creates a virtual environment using `virtualenv`
  - Uses `pip` internally to install packages listed in a `Pipfile`.
  - Locks packages to specific versions with a `Pipfile.lock`.
  - Uses an environment variable named `PIPENV_VENV_IN_PROJECT`. When set to `1` it causes pipenv to create the `virtualenv` inside your project directory in a folder named `.venv` instead of in your home directory
  - Will read a `.env` file and populate the environment variables inside the `virtualenv`
  - Can generate a `requirements.txt` file for use with regular `pip`
  - Closest Node.JS equivalent would be `npm`

Demonstrate how to run a Python program using pipenv using its shell

```
pipenv shell
```

This will start a new shell inside the virtual environment.

Then you can run python programs and they will run with the right set of packages and environment variables

```
python someprogram.py
```

When you are finished running commands in the virtual environment don't forget to exit the shell by issuing the `exit` command, or using Control-D.

Demonstrate how to run a Python program using pipenv using the run command

If you just need to run a single command inside the virtual environment you can use the `pipenv run` command.

```
pipenv run python someprogram.py
```

Describe how modules and packages are found and loaded from import statements

First some definitions:

Module : a single .py file or a directory with a `__init__.py` file can be considered a module

Package : a collection of modules and submodules in a directory

Submodule : a python module inside a sub directory of a module

Python Path

The Python Path is a list of directories python looks for modules in.

When you import a module, python searches these directories for a file module or directory module (with a **init**.py file in it) that matches the name you are trying to import.

You can inspect the python path from python by printing `sys.path`

You can add directories to the python path by setting the `PYTHON_PATH` environment variable.

Luckily we have tools like `virtualenv` and `pipenv` which means we do not have to worry as much about setting the Python path manually.

Exporting

Inside a python script, any variables, functions or classes are automatically exported and can be imported by name.

If you want to control which things get exported from a python module you can set the variable `__all__` equal to a list of strings representing the things to export.

The Rules

Using the `import` statement

- When you import a .py file as a module, it searches `sys.path` for a file with that name and runs that file.
- When you import a directory as a module, it also searches `sys.path` for a directory with that name and runs the `__init__.py` contained in that directory.

Using the `python` command line interpreter

- When you run a .py file it runs that file
- When you run a directory it runs `__main__.py`
- When you run a directory with the `-m` option, it searches `sys.path` for the module and runs both the `__init__.py` and the `__main__.py`

Most of the time we'll use `__init__.py` not `__main__.py` when we build our own modules.

Documentation on import

- Import System
- Import Statement

Describe the purpose of and when init.py runs

When you run a directory with the `-m` option, or when you import a directory, the `__init__.py` file executes. The purpose of `__init__.py` to be able to build python packages and subdivide the packages into multiple sub-modules.

Describe the purpose of and when main.py runs

When you run a directory as a regular python program (not with `-m`) the `__main__.py` file is executed. The purpose of `__main__.py` is to allow us to execute a directory as if it was a python program.

Flask Objectives

Setup a new Flask project

Flask is a python based web application server. It is a backend framework similar to Express.js

First, you should install Flask into a virtual environment

```
pipenv install flask
```

Create a python script to start your application. This might be `app.py` or another script which imports an `app/__init.py` module.

This is the bare minimum needed to make Flask application:

```
from flask import Flask
app = Flask(__name__)
```

Flask requires that you set an environment variable called `FLASK_APP` before it will run. It needs to be set to the name of your flask application script or module. You could put this into a `.env` file and let `pipenv` load it or use the `python-dotenv` module to load a `.flaskenv` file.

Often you might use the `.flaskenv` file to load environment variables like `FLASK_APP` and checking it into source control, and reserve the `.env` file for secret information like passwords or database configurations.

Run a simple Flask web application on your computer

Once you have your application setup, you can just run it with flask.

```
pipenv run flask run
```

Utilize basic configuration on a Flask project

You can use the `app.config` dictionary to hold Flask configuration values.

An even better way to setup your flask app is to create a python module with a configuration class in it. This class just needs properties for each configuration variable. Then you can import the class, and use the `from_object()` method to load it into the app's config dictionary.

```
# config.py

class Config:
    SOME_CONFIG_VARIABLE = 'Some value'
```

```
# app.py
# Import the config class
from config import Config

app = Flask(__name__)

# Load the config into Flask.
app.config.from_object(Config)
```

You can access any config variables in your flask app by just referencing them on the `app.config` dictionary.

```
app.config['SOME_CONFIG_VARIABLE']
```

Create a static route in Flask

A static route is one that just routes to a path without any parameters.

```
# Examples
@app.route('/')
def index():
    """Put code here to execute when `/` is visited"""
    pass

@app.route('/somepath')
def some_path():
    """Put code here to execute when `/somepath` is visited"""
    pass
```

Create a parameterized route in Flask

A parameterized route uses `<>` characters to declare that part of a path should be a parameter.

```
# the <id> parameter will be captured and passed into the function as the first
# argument
@app.route('/item/<id>')
def item(id):
    return f'<h1>Item {id}</h1>'
```

```
# You can also specify the type of the parameter by prepending it with the type
# and a colon
@app.route('/item/<int:id>')
def item(id):
    return f'<h1>Item {id}</h1>'
```

Use decorators run code before and after requests

The `@app.before_request` and `@app.after_request` happen before and after every request to the server. Use them to do any initialization or cleanup you need to happen on each request

```
@app.before_request
def before_request_function():
```

```
    print("before_request is running")


@app.after_request
def after_request_function(response):
    print("after_request is running")
    return response
```

`@app.before_first_request` only happens once before the very first request to the server

```
@app.before_first_request
def before_first_function():
    print("before_first_request happens once")
```

Identify the "static" route

Don't confuse this with declaring a static route above. This is a special built in route you don't have to define at all.

If you create a folder called `static` then any requests to `/static` on your server will cause flask to serve up the files contained in this folder.

```
http://localhost:5000/static/styles/main.css
```

```
.
├── Pipfile
├── Pipfile.lock
├── app              <- directory where Flask is created
│   ├── __init__.py  <- file in which Flask is created
│   ├── routes.py
│   ├── static       <- static files served from here
│   │   └── styles
│   │       └── main.css
│   └── templates
│       └── main.html
└── app_loader.py
```

Use WTForms to define and render forms in Flask

WTForms is a python package that allows you to easily generate forms and form fields. Flask-WTF is a companion python package that allows you to parse POST data from a form and render the form fields.

You define your form as a class that inherits from the FlaskForm base class.

```
from flask_wtf import FlaskForm

class SampleForm(FlaskForm):
```

Then inside the class use WTForm fields on properties of the class.

```
class SampleForm(FlaskForm):
    name = StringField('Name')
```

In your route, you can instantiate an instance of your form and then pass it to a view to be rendered.

```
from app.sample_form import SampleForm

# Create an instance of our form
form = SampleForm()

# And pass it to the view template
return render_template('form.html', form=form)
```

Inside the view template, you can access the fields from the form to output HTML for the form and it's fields.

```
<form action="" method="post" novalidate>
    {{ form.csrf_token }}
    <p>
        {{ form.name.label }}
        {{ form.name(size=32) }}
    </p>
    <p>{{ form.submit() }}</p>
</form>
```

The calls inside of the `{{ }}` will output HTML.

Because of some special python magic (the **call** and **str** methods on FlaskForm), you can just use the properties without calling them, or call them with extra parameters, and both will work!

Passing extra keyword parameters to the field instances will add HTML attributes for those parameters. However, because `class` is a reserved word in Python, you will have to use `class_` when you want to add a CSS class.

```
    form.name(size=32, class_='name')
```

Use WTForms to validate data in a POST with the built-in validators

To validate a form with Flask-WTF you can call the `validate_on_submit` method on your form instance. This must be done inside of a route that handles `POST` requests.

```
@app.route('/submit', methods=['POST'])
def handle_form_submit():
    if form.validate_on_submit():
        # Do something with the form data.
        # and return something
        return
    # You can put code here to handle what happens when
    # the form fails validation, like redirecting or rendering
    # the form again.
    return
```

It should be noted that `validate_on_submit` *automatically* reads the incoming parameters from the `request` object in Flask, so there's no reason to import it or use it manually.

CSRF

To protect against Cross-Site Request Forgery attacks, Flask-Wtf automatically generates and checks CSRF tokens. However we must add one of these two fields in our form in order to print out the CSRF token.

```
# This one prints out ALL the hidden fields including the CSRF that are
# defined on the form class
{{ form.hidden_tag() }}
```

or

```
# While this one only prints out the CSRF token hidden field
{{ form.csrf_token() }}
```

Use the following basic field types in WTForms

You use these by creating a class property on your class which inherits from FlaskForm

```
class MyForm(FlaskForm):
    field1 = StringField()
```

- BooleanField
- DateField
- DateTimeField
- DecimalField
- FileField
- MultipleFileField
- FloatField
- IntegerField
- PasswordField
- RadioField
- SelectField
- SelectMultipleField
- SubmitField
- StringField
- TextAreaField

Check the documentation on the specific parameters you must pass each type of field.

WTForms Field Documentation

Create a Flask Blueprint

A Flask Blueprint is a way to modularize our routes.

In a new module, import Blueprint and create one like this:

```
# admin.py
from flask import Blueprint

admin_bp = Blueprint('admin', __name__, url_prefix='/admin')
```

Register the Flask Blueprint with the Flask application

Then import it into your main Flask app file
and register it so Flask knows about the routes contained within.

```
from admin import admin_bp

app = Flask()

app.register_blueprint(admin_bp)
```

Use the Flask Blueprint to make routes

Inside the blueprint you can add routes, like you normally would, just you use the blueprint instance instead of using `app`

```
@admin_bp.route('/', methods=('GET', 'POST'))
def admin_index():
    return
```

Configure and use sessions in Flask

You must set a `SECRET_KEY` property in your flask config for sessions to work.

You can import session from flask.

```
from flask import Flask, session
```

Then simply use `session` to store things you want to be available later

```
# To set something in the session
session['key'] = value
# To get something from the session
session.get('key')
# to remove something from the session
session.pop('key')
```

Use a Jinja template as return for a Flask route with render_template

Use the `render_template` method to render the template into a string, and then return it from your route. You can give it the HTML file and keyword arguments that will be accessible as variables inside the template.

```
@app.route('/')
def index():
    return render_template('index.html', sitename='My Sample')
```

Add variables to a Jinja template with {{ }}

Then inside our HTML we can access the key

```
<title>{{ sitename }}</title>
```

Check the Jinja2 docs for all the things you can do in Jinja2 templates.

Use include to share template content in Jinja

Just use the include directive to include another html inside a jinja template.

```
{% include 'file.html' %}
```

Psycopg Objectives

Connect to a PostgreSQL RDBMS using Psycopg

```
import psycopg2

CONNECTION_PARAMETERS = {
                          'dbname': 'psycopg_test_db',
                          'user': 'psycopg_test_user',
                          'password': 'password',
}

with psycopg2.connect(**CONNECTION_PARAMETERS) as conn:
    print(conn.get_dsn_parameters())
```

Open a "cursor" to perform data operations

Use the with keyword to clean up connections and database cursors

```
import psycopg2

CONNECTION_PARAMETERS = {
                          'dbname': 'psycopg_test_db',
                          'user': 'psycopg_test_user',
                          'password': 'password',
}

with psycopg2.connect(**CONNECTION_PARAMETERS) as conn:
    print(conn.get_dsn_parameters())
```

Use results performed from executing a SELECT statement on existing database entities

```
with psycopg2.connect(**CONNECTION_PARAMETERS) as conn:
    with conn.cursor() as curs:
        curs.execute('SELECT manu_year, make, model FROM cars;')
        cars = curs.fetchall()
        for car in cars:
            print(car) # (1993, 'Mazda', 'Rx7')
```

Use parameterized SQL statements to insert, select, update, and delete data

```
def print_all_cars():
    with psycopg2.connect(**CONNECTION_PARAMETERS) as conn:
        with conn.cursor() as curs:
            curs.execute('SELECT manu_year, make, model, owner_id FROM cars;')
            cars = curs.fetchall()
            for car in cars:
                print(car)


print_all_cars()
# Output:
# (1993, 'Mazda', 'Rx7', 1)
# ...additional cars
```

Specify what type Psycopg will convert the following PostgreSQL types into:

PostgreSQL	Python
NULL	None
bool	bool
double	float
integer	long
varchar	str
text	unicode
date	date

SQLAlchemy Objectives

Describe how to create an "engine" that you will use to connect to a PostgreSQL database instance

Note: When using Flask-SQLAlchemy you don't have to do this

```
from sqlalchemy import create_engine

engine = create_engine("postgresql://sqlalchemy_test:password@localhost/sqlalchemy_test")
```

Describe how the with engine.connect() as connection: block establishes and cleans up a connection to the database

Note: When using Flask-SQLAlchemy you don't have to do this

```
from sqlalchemy import create_engine

db_url = "postgresql://sqlalchemy_test:password@localhost/sqlalchemy_test"
engine = create_engine(db_url)

with engine.connect() as connection:
    result = connection.execute("""
        SELECT o.first_name, o.last_name, p.name
        FROM owners o
        JOIN ponies p ON (o.id = p.owner_id)
    """)
    for row in result:
        print(row["first_name"], row["last_name"], "owns", row["name"])

engine.dispose()
```

Describe how to create a database session from an engine

Note: When using Flask-SQLAlchemy you don't have to do this

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

db_url = "postgresql://sqlalchemy_test:password@localhost/sqlalchemy_test"
engine = create_engine(db_url)

SessionFactory = sessionmaker(bind=engine)

session = SessionFactory()

# Do stuff with the session

engine.dispose()
```

Create a mapping for SQLAlchemy to use to tie together a class and a table in the database

Mappings

Mappings with plain SQLAlchemy

With just SQLAlchemy we inherit from `Base` and we have to import all the schema objects and types manually.

```
# ponies.py
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.schema import Column, ForeignKey
from sqlalchemy.types import Integer, String

Base = declarative_base()

class Pony(Base):
    __tablename__ = 'ponies'

    id = Column(Integer, primary_key=True)
    name = Column(String(255))
    birth_year = Column(Integer)
    breed = Column(String(255))
    owner_id = Column(Integer, ForeignKey("owners.id"))
```

Mappings with Flask-SQLAlchemy

When using Flask-SQLAlchemy we inherit from `db.Model` instead of `Base` and we can use all the schema objects and types because Flask-SQLAlchemy attaches them to the `db` instance. So we just prefix them with `db.`

```
# owner.py

from .models import db

class Pony(db.Model):
    __tablename__ = 'ponies'

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(255))
    birth_year = db.Column(db.Integer)
    breed = db.Column(db.String(255))
    owner_id = db.Column(db.Integer, db.ForeignKey("owners.id"))
```

Relationships

One-to-Many

Just create the proper foreign key columns on the models, and then define the relationships. *(Remember Flask-SQLAlchemy will need to preface most of these objects with `db.`)*

Remember the rule of thumb. The "Many" always has the foreign key on it.

```
# The one
class Owner(db.Model):
    __tablename__ = "owners"

    id = db.Column(db.Integer, primary_key=True)
    first_name = db.Column(db.String(255))
    last_name = db.Column(db.String(255))
    email = db.Column(db.String(255))

    # ponies belong to an owner
    ponies = db.relationship("Pony", back_populates="owner")

# The Many
class Pony(db.Model):
    __tablename__ = "ponies"

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(255))
    birth_year = db.Column(Integer)
    breed = db.Column(db.String(255))
    # The pony contains an owner_id foreign key
    owner_id = db.Column(db.Integer, db.ForeignKey("owners.id"))
    # An owner has many ponies
    owner = db.relationship("Owner", back_populates="ponies")
```

Many-to-Many

Remember that a Many-to-Many relationship is really two One-to-Many relationships with a join table in the middle.

You must create a `Table()` object and not a model for your join table.

```
# We define the foreign keys on our join table, which joins the Ponies
# to thier Handlers.
pony_handlers = db.Table(
    "pony_handlers",
    db.Column("pony_id", db.ForeignKey("ponies.id"), primary_key=True),
    db.Column("handler_id", db.ForeignKey("handlers.id"), primary_key=True)
```

Then setup the relationships on each Model making sure to define a "secondary" keyword argument is set to the table we just made.

```
    # Inside the Pony class...
    handlers = db.relationship("Handler",
                           secondary=pony_handlers,
                           back_populates="ponies")

    # Inside the Handler class...
    ponies = db.relationship("Pony",
                         secondary=pony_handlers,
                         back_populates="handlers")
```

On backpopulates

If you leave out the backpopulates parameter, then when you create an object and add related data, the opposite relationship won't be populated. For instance assume we have an `Owner` instance and we add a `Pony` instance to it.

```
owner.ponies.append(pony)
```

If we do not have `backpopulates` set to the `owner` propery of the Pony class, then if you try to look at the owner of the pony like this:

```
print(pony.owner) # Returns None
```

Then it will still be None. If you set `backpopulates` to the owner, then this will get populates and stay in sync.

IMPORTANT: backpopulates just controls what happens with the objects *BEFORE* we commit them to the database.

It's always a good idea to setup your `backpopulates` properly so you aren't surprised.

Add data to the database, both single entities as well as related data

```
you = Owner(first_name="your first name",
            last_name="your last name",
            email="your email")

your_pony = Pony(name="your pony's name",
                 birth_year=2020,
                 breed="whatever you want",
                 owner=you)

# Note, id will be None until we commit
print(you.id)         # > None
print(your_pony.id)   # > None

session.add(you)      # Connects you and your_pony objects
session.commit()      # Saves data to the database

# After commiting the ids exist
print(you.id)         # > 4 (or whatever the new id is)
print(your_pony.id)   # > 4 (or whatever the new id is)
```

Using session with Flask-SQLAlchemy

We use this exactly the same as above but we get the session from the `db` instance.

```
db.session.add(you)      # Connects you and your_pony objects
db.session.commit()      # Saves data to the database
```

IMPORTANT don't confuse this session with the Flask session. This is a *database* session while flask session is the *browser* session.

Update data in the database

```
print(your_pony.birth_year)    # > 2020

# Updating is just like setting a property
your_pony.birth_year = 2019

# The pony instance updates immediately
print(your_pony.birth_year)    # > 2019

# but the database doesn't update until we commit!
session.commit()

print(your_pony.birth_year)    # > 2019
```

Delete data from the database (including cascades!)

Know how to use and specify the "delete-orphan" cascading strategy

```
# Just passing the owner instance to delete, deletes it, but....
db.session.delete(you)
# It doesn't actually change the database until you commit!
db.session.commit()
```

```
class Owner(db.Model):
    __tablename__ = 'owners'

    id = db.Column(db.Integer, primary_key=True)
    first_name = db.Column(db.String(255))
    last_name = db.Column(db.String(255))
    email = db.Column(db.String(255))

    # This is a relationship between Ponies and Owner.
    # We have set it to cascase and delete orphans so
    # when we delete an owner all the ponies related to
    # that owner will be deleted
    ponies = db.relationship("Pony",
                         back_populates="owner",
                         cascade="all, delete-orphan")
```

Describe the purpose of a Query object

When you use SQLAlchemy's querying API, you're not actually immediately executing SQL against the database. Instead, all of the specifications that you add to the query are saved up into a single object that you then use to have SQL executed against the database. This allows you to make decisions at runtime about how you want to apply filters to the query. This will become clearer as you read about how to query and apply filters in the following sections. The important thing to note is that a Query object will not actually do anything with the database unless you explicitly tell it to do something.

Use a Session object to query the database using a model

With plain SQLAlchemy

```
pony_query = session.query(Pony)
print(pony_query)
```

```
pony_id_4_query = session.query(Pony).get(4)
```

With Flask SQLAlchemy

Flask SQLAlchemy attaches the `session.query` to the Model directly.

So you can re-write any call to `session.query` as `<Model>.query`.

```
# This plain SQLAlchemy query:
pony = session.query(Pony).get(4);

# Can be re-written as:
pony = Pony.query.get(4)
```

How to order your results

```
owner_query = Owner.query(Owner.first_name, Owner.last_name)
                    .order_by(Owner.last_name)
print(owner_query)
```

Use the filter method to find just what you want

```
pony_query = Pony.query.filter(Pony.name.like("%u%"))

pony_query = Pony.query
                .filter(Pony.name.ilike("%u%"))
                .filter(Pony.birth_year < 2015)
```

Use instance methods on the Query object to return a list or single item

- `all` - returns a list
- `first` - returns a single object
- `one` - returns a single object or raises an exception
- `one_or_none` - returns a single object or None

```
ponies = Pony.query.all()
for pony in ponies:
    print(pony.name)
```

Use the count method to … count

```
pony_query = Pony.query
print(pony_query.count())
```

Query objects with criteria on dependant objects

```
hirzai_owners = Owner.query \
                    .join(Pony)  \
                    .filter(Pony.breed == "Hirzai")

for owner in hirzai_owners:
    print(owner.first_name, owner.last_name)
```

Lazily load objects

```
for owner in Owner.query:
    print(owner.first_name, owner.last_name)
    for pony in owner.ponies:
        print(pony.name)
```

Eagerly load objects

```
owners_and_ponies = Owner.query.options(joinedload(Owner.ponies))

for owner in owners_and_ponies:
    print(owner.first_name, owner.last_name)
    for pony in owner.ponies:
        print(pony.name)
```

Install the Flask-SQLAlchemy extension to use with Flask

```
pipenv install Flask psycopg2-binary \
               SQLAlchemy Flask-SQLAlchemy
```

Configure SQLAlchemy using Flask-SQLAlchemy

Create a `SQLALCHEMYDATABASE_URI` property in your Flask app config

Then you can pass your app to SQLAlchemy for super simple apps

```
from config import Config
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config.from_object(Config)
# We are creating the DB in app.py after creating the app.
# So we can just pass our app to SQLAlchemy
db = SQLAlchemy(app)
```

However, if you've defined your db object BEFORE your app is created in another module, you must use the `init_app` method on db to configure Flask-SQLAlchemy

```
# models.py
from flask_sqlalchemy import SQLAlchemy

# notice we create the db instance without passing it app
db = SQLAlchemy()
```

```
# app.py
from flask import Flask
from .config import Configuration
# The act of importing this creates the db instance
from .models import db

# We create our app here
app = Flask(__name__)
app.config.from_object(Configuration)
# We use init_app and pass it the app
db.init_app(app)
```

Use the convenience functions and objects Flask-SQLAlchemy provides you to use in your code

Flask-SQLAlchemy adds the query object to every instance of a Model.

```
Pony.query.get(4)
```

It has some Flask specific things such as `get_or_404`, which just throws a
404 error if there's no Pony coming back from the database. There is also a similar `first_or_404` method.

```
Pony.query.get_or_404(4)
```

Flask-SQLAlchemy also adds the `session` object to the `db` instance.

```
db.session.add(owner)
db.session.commit()
```

Alembic Learning Objectives

Install Alembic into your project

```
pipenv install alembic
pipenv run alembic init <directory-name>
```

Configure Alembic to talk to your database and not have silly migration names

Add environment variable to `env.py`

Import the `os` module

```
import os
```

before `run_migrations_offline` add this line

```
config.set_main_option("sqlalchemy.url", os.environ.get("DATABASE_URL"))
```

Making better migration file names

You can set this in `alembic.ini` so your migration files will have dates in the names.

```
file_template = %%(year)d%%(month).2d%%(day).2d_%%(hour).2d%%(minute).2d%%(second).2d_%%(slug)s
```

Control Alembic's ability to migrate your database

Generating a migration (revision)

```
pipenv run alembic revision -m "create the owners table"
```

Running a migration (upgrading to a revision)

```
pipenv run alembic upgrade head
```

Rolling back a migration (downgrading to a revision)

```
pipenv run alembic downgrade <revision hash>
```

Rolling back all migrations (downgrading to base)

```
pipenv run alembic downgrade base
```

Viewing your migration history (revision history)

```
pipenv run alembic history
```

Reason about the way Alembic orders your migrations; and,

Alembic treats migrations like a linked list. It does not use the dates in the
filenames to decide which migrations to run and which order they get run.

Instead each revision has a revision hash, and each revision has a 'down_revision'
property that points at the previous revision. (except for the first revision which of course will have it's down_revision set to None)

```
revision = 'ddbf30c38165'
down_revision = 'e363377eb6d7'
```

Handle branching and merging concerns

If two teammates both commit new revisions, then you will end up with a conflict
in the down_revisions. Your revision linked list might look like this:

```
                    -- ae1027a6acf (Team A's most recent)
                   /
<-- 1975ea83b712 <--
                   \
                    -- 27c6a30d7c24 (Team B's most recent)
```

and you'll get an error like this:

```
FAILED: Multiple head revisions are present for given argument 'head';
please specify a specific target revision, '<branchname>@head' to
narrow to a specific head, or 'heads' for all heads
```

you can solve this with a merge specifying the two revisions

```
pipenv run alembic merge -m "merge contracts and devices" ae1027 27c6a
```

Configuring a Flask application to use Alembic;

```
pipenv install alembic Flask-Migrate
```

```
# app/__init__.py
from app.models import db
from flask import Flask
from config import Config
# We have to import flask_migrate
from flask_migrate import Migrate
import os

app = Flask(__name__)
# Load our config, make sure you set DATABASE_URL as flask migrate
# uses it as well
app.config.from_object(Config)
db.init_app(app)
# And we have to do this to configure Flask Migrate. It needs to know about
# both our app and our db object
Migrate(app, db)
```

Run commands to manage your database through the flask command; and,

When we use Flask-Migrate we run the commands through the flask command.

Instead of alembic init…

```
pipenv run flask db init
```

Check the help for the rest of the commands, which are the same as Alembic

```
pipenv run flask db --help

Usage: flask db [OPTIONS] COMMAND [ARGS]...

  Perform database migrations.

Options:
  --help  Show this message and exit.

Commands:
  branches   Show current branch points
  current    Display the current revision for each database.
  downgrade  Revert to a previous version
  edit       Edit a revision file
  heads      Show current available heads in the script directory
  history    List changeset scripts in chronological order.
  init       Creates a new migration repository.
  merge      Merge two revisions together, creating a new revision file
  migrate    Autogenerate a new revision file (Alias for 'revision...
  revision   Create a new revision file.
  show       Show the revision denoted by the given symbol.
  stamp      'stamp' the revision table with the given revision; don't run...
  upgrade    Upgrade to a later version
```

Autogenerate migrations from your models!

Instead of alembic migrate…

```
pipenv run flask db migrate -m "create owners table"
```

flask db migrate does **magic** now, it reads your models and tries to
autogenerate the migration files based on the model.

IMPORTANT always check the autogenerated migration though, as there's only
so much flask migrate can do and it might not get everything perfectly correct, but it is a time saver!