Welcome to Web Programming Foundations

Welcome to **Web Programming Foundations**. In this course, we will learn some of the foundational skills necessary to build and deploy scalable Web Applications that leverage current technologies such as **Node.js**, **Express.js**, **HTML5** / **CSS3**, **EJS** and **Postgres**. This will involve studying: foundational knowledge of JavaScript (ECMAScript 6, ie: ES6), the Node.js JavaScript Runtime, the Express.js web application framework, the HTTP Protocol, HTML5 / CSS3, Database connectivity and finally, an introduction to web security.

Before we get started, please have a look around the site and familiarize yourself with the content. If you're looking to get started right away, the "Hello World" is a great place to start.

Developer Tools & Core Technologies

Throughout this course, we will be working almost exclusively in the following environments:

Visual Studio Code

“Visual Studio Code is an open-source (free) streamlined code editor with support for development operations like debugging, task running and version control. It aims to provide just the tools a developer needs for a quick code-build-debug cycle and leaves more complex workflows to fuller featured IDEs”. Visual Studio Code also runs on Mac OS X, Linux and Windows operating systems, which will provide the class with a single unified environment to work in regardless of a student’s choice of laptop or home computer. Some of the noteworthy features of Visual Studio Code Include:

Integrated Terminal

“In Visual Studio Code, you can open an integrated terminal, initially starting at the root of your workspace. This can be very convenient as you don’t have to switch windows or alter the state of an existing terminal to perform a quick command line task”.

To open the terminal:

- Use the keyboard shortcut **Ctrl +** `
- Use the **View** | **Toggle Integrated Terminal menu command.**

Smart Editing

VS Code comes with a built-in JavaScript language service so you get JavaScript code intelligence out-of-the-box. Language services provide the code understanding necessary for features like:

- IntelliSense: (suggestions)
- smart code navigation (Go to Definition, Find All References, Rename Symbol)

File & Folder Based

Since VS Code is file and folder based – you can get started immediately by simply opening a file or folder in VS Code.

“On top of this, VS Code can read and take advantage of a variety of project files defined by different frameworks and platforms. For example, if the folder you opened in VS Code contains one or more package.json (which we will be making extensive use of during the semester), project.json, tsconfig.json, or .NET Core Visual Studio solution and project files, VS Code will read these files and use them to provide additional functionality, such as rich IntelliSense in the editor”.

Version Control

Visual Studio Code has integrated Git support for some of the most common commands, making it easy to verify and commit code changes (see "Git" below).

Modern Web Browser

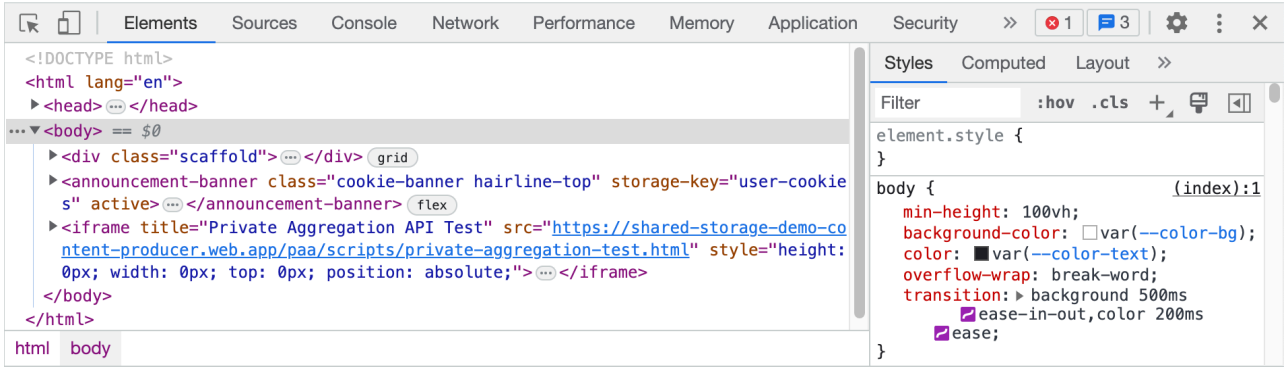
A modern web browser such as Google Chrome or Mozilla Firefox will be used regularly throughout this course. Microsoft Edge will work as well, as it supports a similar set of development tools, however due to it's lack of plugins / addons and cross-platform support it's not as highly recommended. All screenshots and development examples used throughout this course have been taken in Google Chrome.

Browser Developer Toolbar

Before starting this course, students should have at least a basic understanding of the Developer Tools built into a modern web browser. Typically, pressing the F12 Key (Windows) will open the bar, however there are alternate ways of opening it. For Google chrome:

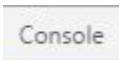
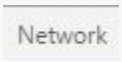
- Open the Chrome menu at the top-right of your browser window, then select Tools > Developer Tools.
- Right-click on any page element and select Inspect.

This will bring up the Chrome "Developer Toolbar", as seen below :



We will be working with many of these panels throughout the semester. A quick list of their functionality (from left to right, starting at the top left corner) is as follows:

Icon	Description
	**Element Inspector:** Select an element in the page to inspect it; this will cause the Developer Tools (Devtools) to switch to the “Elements” panel and highlight the rendered source code (HTML) responsible for displaying the item. This will also cause the “Styles” panel (on the right) to highlight all current CSS applied to the element.
	**Device Toolbar Toggle:** Toggles the “device toolbar” on and off. This allows the developer to select a device and manually enter the pixel dimensions of the screen and scale of the page. This is useful for ensuring that the page looks correct on a variety of devices.
	**Elements Panel:** Shows a view of the current page’s Document Object Model (DOM) tree as HTML. Selecting a given node (element) will highlight it in the page and show it’s

Icon	Description
	applied CSS in the “Styles” panel. Developers can also modify this element and corresponding CSS (“Styles” panel) live and see the results directly in the browser. **Important Note:** The HTML shown in this panel isn’t necessarily the source code of the page, as it will show elements and attributes that have been dynamically added after the page is loaded. Changes to the HTML/CSS/JavaScript in this mode will not save to the source file.
	**Sources Panel:** Shows a list of all items included in the page (i.e., all images, CSS, JavaScript, etc.) and their corresponding locations of origin. Developers can click on an item to show its contents in the middle (preview) panel. If the selected item is a JavaScript file, developers can (in the “debugger” panel) set breakpoints and watch variables to help identify and debug a misbehaving piece of JavaScript code.
	**Console Panel:** Shows a JavaScript console pane. JavaScript calls to `console.log()` will show the resultant text in this window. Additionally, all JavaScript errors will show up in this location in red. Developers can also write small JavaScript code snippets to be executed immediately within the context of the page.
	**Network Panel:** Is used to get additional insights into requested and downloaded resources. Developers can view a log that tracks all resources loaded including their corresponding status code, type, time (latency), size of the

Icon	Description
	resource and the initiator of the request.
Performance	**Performance Panel:** Enables a tool that allows developers to record and analyze all the activity in their applications as they run. It’s the best place to start investigating perceived performance issues. This is done by recording a timeline of every event that occurs after a page loads and analyzing the corresponding FPS, CPU, and network requests.
Memory	**Memory Panel:** Provides more detailed debugging information than the timeline by enabling developers to record detailed CPU/Memory profiles such as a “Heap Snapshot”, “Allocation instrumentation on timeline", and “Allocation sampling".
Application	**Application Panel:** Allows developers to inspect and manage client-side storage, caches, and resources. This includes: key-value pairs stored in “Local Storage”, access to IndexedDB Data (a JavaScript-based object-oriented database used to store data locally), a “Web SQL” explorer (deprecated in favor of IndexedDB), as well as access to stored cookies and cache data. This is very useful in verifying that your application is storing data correctly on the client side.
Security	**Security Panel:** Gives an overview of a page from a security standpoint including: Certificate verification (indicating whether the site has proven its identity with a TLS certificate), Transport Layer Security (TLS) connection (Note: TLS is often

Icon	Description
	referred to by the name of its predecessor, SSL), and Subresource security (indicating whether the site loads insecure HTTP subresources – i.e., “mixed content”).
	**Error Icon:** Displays the number of errors present in the “Console Pane”. To review the errors, simply switch over to the Console pane and locate the items highlighted in red.
	**Customize Icon:** Controls where the Developer Toolbar should be placed relative to the browser, as well as a collection of all related settings and preferences for the tool set.
	**Close Icon:** Closes the Developer Toolbar.

Core Technologies

Additionally, we will cover a number of topics surrounding the following technologies (in no particular order):

JavaScript (ECMAScript)

A huge focus of this course will be on JavaScript. In fact – JavaScript will be the only official programming language that we will be studying in this course. While we will be interacting with HTML5 and CSS3, neither is considered a “programming language” in the same way that C, C++ or JavaScript is. HTML5 and CSS3 are instead considered markup languages and style sheet languages respectfully – that is, they describe presentation, whereas programming languages describe function. Regardless, we will be focusing exclusively on JavaScript and how a number of very sophisticated tools and frameworks can help us create efficient and functional web applications.

ECMAScript

Back in 1996 the JavaScript language specification was taken to Ecma (European Computer Manufacturers Association) International to develop a formal standardized specification, which other browser vendors and companies could implement and expand upon. This standardized JavaScript was dubbed

“ECMAScript” and specific vendor versions of the specification were known as “dialects”, the most popular of which being “JavaScript”. When we refer to “JavaScript” we’re really referring to a dialect of ECMAScript that has been implemented in the engine / runtime environment that is running our JavaScript formatted code. For example, this includes JavaScript engines like SpiderMonkey in Firefox and v8 in Chrome.

In 2015, ECMAScript 6 was released and many important features were introduced, such as:

- Arrow Functions
- Class Definitions
- Block Scoped Variables
- Promises
- Binary & Octal literals
- Modules
- and many more…

Since then, development of ECMAScript has continued and new versions are released yearly. For a comprehensive list of which features are supported in specific browsers, environments and runtimes, see:

- ECMAScript Compatibility Table

Node.js

At it’s core, Node.js is an open-source, cross-platform JavaScript runtime

environment built on Chrome's V8 JavaScript engine. It is typically used for developing server-side and networking applications and has exploded as the go-to application framework for many real-time web applications. This is largely due to it's event-driven, non-blocking I/O model which ensures that the main thread of execution is not kept waiting for slow I/O operations (ie: stopping and waiting for a database query to complete). Some major companies using it include Paypal, eBay, GoDaddy, Microsoft, Shutterstock, Uber, Wikia just to name a few.

Node.js also has an expansive package ecosystem accessible via it's Node Package Manager (NPM) utility. We will leverage this by experimenting with a number of popular, open-source modules including:

- Express.js (http://expressjs.com)
- EJS (https://ejs.co)
- Multer (https://github.com/expressjs/multer)
- Sequelize (https://sequelize.org)
- Helmet (https://helmetjs.github.io)

Git

We will be using Git: a command-line tool which serves as a version control system used for tracking changes to your source code and making it available for collaboration with other developers (by leveraging online tools such as Github or GitLab). Additionally, there are many online services that connect to your published code to 3rd party cloud platforms such as Render or Netlify which can build your code and host your web application. For this class, we will

be using Vercel - please see the Vercel Guide for more information.

There is a ton of information online on how to get started using Git / GitHub, such as:

- An Intro to Git and GitHub for Beginners (Tutorial)
- Pro Git (eBook)
- Git and GitHub learning resources

PostgreSQL

From the PostgreSQL site, postgresql.org:

> “PostgreSQL (also known as “Postgres”) is a powerful, open source object-relational database system. It has more than 15 years of active development and a proven architecture that has earned it a strong reputation for reliability, data integrity, and correctness. It runs on all major operating systems, including Linux, UNIX (AIX, BSD, HP-UX, SGI IRIX, macOS, Solaris, Tru64), and Windows. It is fully ACID compliant, has full support for foreign keys, joins, views, triggers, and stored procedures (in multiple languages). It includes most SQL:2008 data types, including INTEGER, NUMERIC, BOOLEAN, CHAR, VARCHAR, DATE, INTERVAL, and TIMESTAMP. It also supports storage of binary large objects, including

> pictures, sounds, or video. It has native programming interfaces for C/C++, Java, .Net, Perl, Python, Ruby, Tcl, ODBC, among others, and exceptional documentation.

Hello World

To get a sense of how to write code using the tools for this course and to ensure that your development environment is set up correctly, let's start with a simple "Hello World".

1. If you haven’t already, be sure to download and install the current release of Node.js. If you’re not sure whether or not you have Node.js installed, open the **Command Prompt** and type **node -v**. If Node.js has been installed, this will output the version.

2. Make sure you have Visual Studio Code installed. This is an open-source, cross-platform development environment provided by Microsoft. While it is true that you can write your code in any text editor, Visual Studio Code works very nicely alongside Node.js and all examples going forward will assume that you are using Visual Studio Code. You can download it here

3. On your Local computer, navigate to your desktop and **create a folder** called **Ex1**

4. Open **Visual Studio Code** and select **File -> Open Folder**. Choose your newly created **“Ex1”** Folder and click **“Select Folder”**

5. You should see an “Explorer” pane open on the left side with two items: “Open Editors” and “Ex1”. Click to expand “Ex1” and locate the “New File” button (). Click this and type **“hello.js”.**

6. You should now see your newly created “Hello.js” file in the editor. Enter the following line of code:

```
console.log('Hello World!');
```

and click **File -> Save (Ctrl + S)**

7. Open the **Integrated Terminal** by selecting **View -> Integrated Terminal (Ctrl + `)** and type:

```
node hello.js
```

Hello World! This is the most basic example in Node.js – notice how we didn't need to open a web browser, scratchpad, devtools, etc? It's also important to note that the command **"node hello.js"** can be executed in any command prompt as long as the active working directory is set to wherever your **hello.js** file is located (Ex1 in this case). The Integrated Terminal is just a quick, easy way to get a command prompt running in the correct location without leaving the development environment.

Introduction to JavaScript

As you may have guessed from above, the first web technology we will learn is JavaScript. JavaScript (often shortened to JS) is a lightweight, interpreted or JIT (i.e., Just In Time) compiled language meant to be embedded in host environments, for example, web browsers or runtime environments such as "Node.js".

JavaScript looks similar to C/C++ or Java in some of its syntax, but is quite different in philosophy; it is more closely related to Scheme than C. For example, JavaScript is a dynamic scripting language supporting multiple programming styles, from object-oriented to imperative to functional.

JavaScript is one of, if not the most popular programming languages in the world, and has been for many years. Learning JavaScript well will be a

tremendous asset to any software developer, since so much of the software we use is built using JS.

> JavaScript's many versions: JavaScript is an evolving language, and you'll hear it referred to by a number of names, including: ECMAScript (or ES), ES5, ES6, ES2015, ES2017, ..., ES2021, ES2022, etc. ECMA is the European Computer Manufacturers Association, which is the standards body responsible for the JS language. As the standard evolves, the specification goes through different versions, adding or changing features and syntax. In this course we will primarily focus on ECMAScript 6 (ES6) and newer versions, which all browsers support. We will also sometimes use new features of the language, which most browsers support. Language feature support across browsers is maintained in this table.

JavaScript Resources

Throughout the coming weeks, we'll make use of a number of important online resources. They are listed here so you can make yourself aware of them, and begin to explore on your own. All programmers, no matter how experienced, have to return to these documents on a routine basis, so it's good to know about them.

- JavaScript on MDN
  - JavaScript Guide
  - JavaScript Reference
- Eloquent JavaScript
- JavaScript for impatient programmers (ES2022 edition)

JavaScript Environments

JavaScript is meant to be run within a host environment. There are many

possible environments, but we will focus on the following:

- Web Browsers, and their associated developer tools, primarily:
  - Chrome DevTools
  - Firefox Developer Tools
- Node.js, and its command line REPL (Read-Eval-Print-Loop)

If you haven't done so already, you should install all of the above.

JavaScript Engines

JavaScript is parsed, executed, and managed (i.e., memory, garbage collection, etc) by an engine written in C/C++. There are a number of JavaScript engines available, the most common of which are:

- V8, maintained an used by Google in Chrome and in node.js
- SpiderMonkey, maintained and used by Mozilla in Firefox
- ChakraCore, maintained and used by Microsoft in Edge
- JavaScriptCore, maintained and used by Apple in Safari

These engines, much like car engines, are meant to be used within a larger context. We will encounter them indirectly via web browsers and in node.js.

It's not important to understand a lot about each of these engines at this point, other than to be aware that each has its own implementation of the ECMAScript standards, its own performance characteristics (i.e., some are faster at certain things), as well as its own set of bugs.

JavaScript Syntax

Recommend Readings

We will spend the next few weeks learning JavaScript, and there is no one best

way to do it. The more you read and experiment the better. The following chapters/pages give a good overview:

- Chapter 1. Basic JavaScript of Exploring JS (ES5).
- MDN JavaScript Introduction Tutorial
- Chapter 1. Values, Types and Operators and Chapter 2. Program Structure of Eloquent JavaScript (2nd Ed.).

Important Ideas

- JavaScript is Case-Sensitive: `customerCount` is not the same thing as `CustomerCount` or `customercount`
- Name things using `camelCase` (first letter lowercase, subsequent words start with uppercase) vs. `snake_case`.
- Semicolons are optional in JavaScript, but highly recommended. We'll expect you to use them in this course, and using them will make working in C++, Java, CSS, etc. much easier, since you have to use them there.
- Comments work like C/C++, and can be single or multi-line

```
// This is a single line comment. NOTE: the space between the
// and first letter.

/*
This is a multi-line comment,
and can be as long as you need.
*/
```

- Whitespace: JavaScript will mostly ignore whitespace (spaces, tabs, newlines). In this course we will expect you to use good indentation practices, and for your code to be clean and readable. Many web

programmers use Prettier to automatically format their code, and we will too:

```
// This is poorly indented, and needs more whitespace
function add(a, b) {
  if (!b) {
    return a;
  } else {
    return a + b;
  }
}

// This is much more readable due to the use of whitespace
function add(a, b) {
  if (!b) {
    return a;
  } else {
    return a + b;
  }
}
```

- JavaScript statements: a JavaScript program typically consists of a series of statements. A statement is a single-line of instruction made up of objects, expressions, variables, and events/event handlers.

- Block statement: a block statement, or compound statement, is a group of statements that are treated as a single entity and are grouped within curly brackets `{...}`. Opening and closing braces need to work in pairs. For example, if you use the left brace `{` to indicate the start of a block, then you must use the right brace `}` to end it. The same matching pairs applies to single `'......'` and double `"......."` quotes to designate text strings.

Variables

Variables are declared using the `let` keyword. You must use the `let` keyword to precede a variable name, but you do not need to provide a type, since the initial value will set the type.

```
let year;
let seasonName = 'Fall';

// Referring to and using syntax:
year = 2023;
console.log(seasonName, year);
```

> **NOTE**: JavaScript also supports the `var` and `const` keywords for variable declaration.

Variables must start with a letter (`a-zA-Z`), underscore (`_`), or dollar sign (`$`). They cannot be a reserved (key) word. Subsequent characters can be letters, numbers, underscores. If you forget to use the `let` keyword, JavaScript will still allow you to use a variable, and simply create a *global variable*. We often refer to this as "leaking a global," and it should always be avoided:

```
let a = 6; // GOOD: a is declared with let
b = 7; // BAD: b is used without declaration, and is now a global
```

Data Types

JavaScript is a typeless language -- you don't need to specify a type for your data (it will be inferred at runtime). However, internally, the following data types are used:

- `Number` - a double-precision 64-bit floating point number. Using `Number`

you can work with both Integers and Floats. There are also some special `Number` types, `Infinity` and `NaN`.

- `BigInt` - a value that can be too large to be represented by a `Number` (larger than `Number. MAX_SAFE_INTEGER`,) can be represented by a `BigInt`. This can easily be done by appending `n` to the end of an integer value.

- `String` - a sequence of Unicode characters. JavaScript supports both single (`'...'`) and double (`"..."`) quotes when defining a `String`.

  > **NOTE:** JavaScript doesn't distinguish between a single `char` and a multi-character `String` -- everything is a `String`. You define a `String` using either single (`'...'`), double (`"..."`) quotes. Try to pick one style and stick with it within a given file/program vs. mixing them. Modern ECMAScript also allows for the use of template literals. Instead of `'` or `"`, a template literal uses ` (backticks), and you can also interpolate expressions.

- `Boolean` - a value of `true` or `false`. We'll also see how JavaScript supports so-called *truthy* and *falsy* values that are not pure `Boolean`s.

- `Object`, which includes `Function`, `Array`, `Date`, and many more. - JavaScript supports object-oriented programming, and uses objects and functions as first-class members of the language.

- `Symbol` - a primitive type in JavaScript that represents a unique and anonymous value/identifier. They can normally be used as an object's unique properties.

- `null` - a value that means "this is intentionally nothing" vs. `undefined`

- `undefined` - a special value that indicates a value has never been defined.

Declaration	Type	Value
`let s1 = "some text";`	String	`"some text"`
`let s2 = 'some text';`	String	`"some text"`
`let s3 = '172';`	String	`"172"`
`let s4 = '172' + 4;`	String	`"1724"` (concatenation vs. addition)
`let n1 = 172;`	Number	`172` (integer)
`let n2 = 172.45;`	Number	`172.45` (double-precision float)
`let n3 = 9007199254740993n;`	BigInt	`9007199254740993n` (integer)
`let b1 = true;`	Boolean	`true`
`let b2 = false;`	Boolean	`false`
`let b3 = !b2;`	Boolean	`true`
`let s = Symbol("Sym");`	symbol	`Symbol(Sym)`
`let c;`	undefined	`undefined`

Declaration	Type	Value
`let d = null;`	object	`null`

Arrays

We can create an `Array` in JavaScript using an "Array literal", using the following syntax (we will discuss the "Array Object" further in the coming weeks):

```
let arr2 = [1, 2, 3]; // array literal
```

Like arrays in C / C++, a JavaScript `Array` has a length, and items contained within it can be accessed via an index:

```
let arr = [1, 2, 3];
let len = arr.length; // len is 3
let item0 = arr[0]; // item0 is 1
```

Unlike languages such as C / C++ , a JavaScript `Array` can contain any type of data, including mixed types:

```
let list = [0, '1', 'two', true];
```

JavaScript `Array`s can also contain holes (i.e., be missing certain elements), change size dynamically at runtime, and we don't need to specify an initial size:

```
let arr = []; // empty array
arr[5] = 56; // element 5 now contains 56, and arr's length is now
6
```

> **NOTE:** a JavaScript `Array` is really a **map**, which is a data structure that associates values with unique keys (often called a key-value pair). JavaScript arrays are a special kind of map that uses numbers for the keys, which makes them look and behave very much like arrays in other languages. We will encounter this **map** structure again when we look at how to create `Object`s.

Operators

Common JavaScript Operators (there are more, but these are a good start):

Operator	Operation	Example
`+`	Addition of `Number`s	`3 + 4`
`+`	Concatenation of `String`s	`"Hello " + "World"`
`-`	Subtraction of `Number`s	`x - y`
`*`	Multiplication of `Number`s	`3 * n`
`/`	Division of	`2 / 4`

Operator	Operation	Example
	`Number`s	
`%`	Modulo	`7 % 3` (gives 1 remainder)
`++`	Post/Pre Increment	`x++`, `++x`
`--`	Post/Pre Decrement	`x--`, `--x`
`=`	Assignment	`a = 6`
`+=`	Assignment with addition	`a += 7` same as `a = a + 7`. Can be used to join `String`s too
`-=`	Assignment with subtraction	`a -= 7` same as `a = a - 7`
`*=`	Assignment with multiplication	`a *= 7` same as `a = a * 7`
`/=`	Assignment with division	`a /= 7` same as `a = a / 7`
`&&`	Logical `AND`	`if(x > 3 && x < 10)` both must be `true`
`()`	Call/Create	`()` invokes a function, `f()` means invoke/ call function stored in variable `f`

Operator	Operation	Example
`||`	Logical `OR`	`if(x === 3 || x === 10)` only one must be `true`
`|`	Bitwise `OR`	`3.1345|0` gives `3` as an integer
`!`	Logical `NOT`	`if(!(x === 2))` negates an expression
`==`	Equal	`1 == 1` but also `1 == "1"` due to type coercion
`===`	Strict Equal	`1 === 1` but `1 === "1"` is not `true` due to types. Prefer `===`
`!=`	Not Equal	`1 != 2`, with type coercion
`!==`	Strict Not Equal	`1 !== "1"`. Prefer `!==`
`>`	Greater Than	`7 > 3`
`>=`	Greater Than Or Equal	`7 >=7` and `7 >= 3`
`<`	Less Than	`3 < 10`
`<=`	Less Than Or Equal	`3 < 10` and `3 <=3`
`typeof`	Type Of	`typeof "Hello"` gives `'string'`, `typeof`

Operator	Operation	Example
		`6` gives `'number'`
`cond ? a : b`	Ternary	`status = (age >= 18) ? 'adult' : 'minor';`

> **NOTE:** JavaScript is dynamic, and variables can change value *and* type at runtime:
>
> ```
> let a; // undefined
> a = 6; // 6, Number
> a++; // 7, Number
> a--; // 6, Number
> a += 3; // 9, Number
> a = 'Value=' + a; // "Value=9", String
> a = !!a; // true, Boolean
> a = null; // null
> ```

JavaScript is also a garbage collected language. Memory automatically gets freed at runtime when variables are not longer in scope or reachable. We still need to be careful not to leak memory (i.e., hold onto data longer than necessary, or forever) and block the garbage collector from doing its job.

Expressions

A JavaScript expression is any code (e.g., literals, variables, operators, and expressions) that evaluates to a single value. The value may be a `Number`, `String`, an `Object`, or a logical value.

```
let a = 10 / 2; // arithmetic expression
let b = !(10 / 2); // logical expression evaluates to false
let c = '10 ' + '/' + ' 2'; // string, evaluates to "10 / 2"
```

Conditional Statements

Conditional statements in JavaScript allow programs to make decisions based on various conditions. These conditions can be evaluated to either true or false, and depending on the outcome, different blocks of code are executed.

Basic Syntax

The most fundamental conditional statement in JavaScript is the if statement. It checks a condition and executes a block of code if the condition is true. The syntax for an if statement is as follows:

```
if (condition) {
    // code to execute if condition is true
}
```

Example: Checking Balance

Consider a scenario where you want to check if a user has enough balance to purchase an item. Here's how you could implement this using an `if` statement:

```
// Set balance and price of item
let balance = 500;
let itemPrice = 100;

// Check if there are enough funds to purchase item
if (itemPrice <= balance) {
  console.log('You have enough money to purchase the item!');
```

In this example, the `if` statement checks if the price of the item is less than or equal to the user's balance. If the condition is true, it logs a success message; otherwise, it logs a failure message.

Else Statement

The else statement allows you to specify alternative code to run if the condition in the if statement is false. The syntax combines if and else as follows:

```
if (condition) {
    // code to execute if condition is true
} else {
    // code to execute if condition is false
}
```

Else If Statement

The else if statement in JavaScript allows you to check multiple conditions sequentially. It is used after an if statement and before an optional else statement. The else if statement checks whether a new condition is true. If it is, the code block associated with that condition is executed. If none of the conditions are true, the code block under the else statement (if present) is executed.

Basic Syntax

The basic syntax for using else if is as follows:

```
if (condition1) {
    // code to execute if condition1 is true
```

Example: Checking User Age

Let's say you're building a website that displays content based on the user's age. Here's how you might use if, else if, and else to tailor the experience:

```
let userAge = 25;

if (userAge >= 18 && userAge < 30) {
  console.log("Welcome You're eligible for our special offer.");
} else if (userAge >= 30 && userAge < 40) {
  console.log('Hello Enjoy exploring our products.');
} else {
  console.log('Welcome We hope you enjoy browsing our site.');
}
```

In this example:

- If the user is between 18 and 29 years old, they receive a special welcome message.
- If the user is between 30 and 39 years old, they get a general welcome message.
- If the user is younger than 18 or older than 39, they receive a generic welcome message.

Important Notes

- There is no elseif syntax in JavaScript. Always use else if.
- Conditions are checked in order. Once a condition is found to be true, the corresponding block of code is executed, and the rest of the conditions are ignored.

- An else statement is optional and serves as a fallback if none of the if or else if conditions are met.

By using if, else if, and else, you can create complex logic flows in your JavaScript applications, enabling them to respond dynamically to various conditions.

Ternary Operator

For simple conditions with two outcomes, the ternary operator provides a concise way to write conditional expressions. Its syntax is:

```
condition? expressionTrue : expressionFalse;
```

Here's how you could rewrite the balance check using a ternary operator:

```
const message =
  itemPrice <= balance
    ? 'You have enough money to purchase the item!'
    : 'You do not have enough money in your account to purchase
this item.';
console.log(message);
```

This single line of code performs the same check as the previous example but uses the ternary operator for brevity.

Iteration

Iteration is a fundamental concept in JavaScript, enabling developers to efficiently execute a block of code multiple times based on specific conditions. Whether it's looping through arrays, traversing objects, or repeating actions until a certain criterion is met, iteration helps streamline processes and reduce redundancy in code.

JavaScript provides various iteration mechanisms, such as:

For Loop

One of the most common iteration structures is the "for" statement:

```
for (initialization; condition; afterthought)
  statement
```

For example:

```
for (let i = 0; i < 10; i++) {
  console.log(i);
}
```

This code iterates through a sequence of numbers from 0 to 9, printing each number to the console. Here's a step-by-step explanation of how it works:

1. **Initialization (`let i = 0`)**: The loop starts by declaring a variable `i` and initializing it to 0. This variable acts as the loop counter, which keeps track of the current iteration.

2. **Condition (`i < 10`)**: Before each iteration of the loop, the condition `i < 10` is evaluated. If the condition is true, the loop body (the code inside the loop) is executed. If the condition is false, the loop stops executing.

3. **Iteration (`i++`)**: After the loop body is executed, the `i++` expression increments the value of `i` by 1. This is known as the iteration statement, which prepares for the next loop cycle.

4. **Loop Body (`console.log(i)`)**: Inside the loop, the `console.log(i)` statement is executed, printing the current value of `i` to the console.

Putting it all together, the loop starts with `i` equal to 0 and continues to execute the loop body (printing the value of `i`) and incrementing `i` by 1 on each iteration until `i` is no longer less than 10. The output will be the numbers 0 through 9, each on a new line.

Here's what happens in each iteration:

- **1st iteration**: `i` is 0, `console.log(i)` prints 0, `i` becomes 1.
- **2nd iteration**: `i` is 1, `console.log(i)` prints 1, `i` becomes 2.
- **3rd iteration**: `i` is 2, `console.log(i)` prints 2, `i` becomes 3.
- ...
- **10th iteration**: `i` is 9, `console.log(i)` prints 9, `i` becomes 10.

After the 10th iteration, `i` becomes 10, which makes the condition `i < 10` false, so the loop terminates.

Do While Loop

Another form of iteration is the "do...while" statement:

```
do
  statement
while (condition);
```

For example:

```
let i = 0;

do {
  console.log(i);
  i++;
} while (i < 10);
```

This code once again iterates through a sequence of numbers from 0 to 9, printing each number to the console. Here's a step-by-step explanation of how it works:

1. **Initialization (`let i = 0`)**: The loop starts by declaring a variable `i` and initializing it to 0. This variable acts as the loop counter, which keeps track of the current iteration.

2. **Loop Body (`console.log(i); i++`)**: Inside the `do` block, the `console.log(i)` statement is executed, printing the current value of `i` to the console. Then, the `i++` statement increments the value of `i` by 1.

3. **Condition (`i < 10`)**: After executing the loop body, the condition `i < 10` is evaluated. If the condition is true, the loop repeats and executes the body again. If the condition is false, the loop terminates.

Unlike a `for` loop, a `do...while` loop ensures that the loop body is executed at least once, even if the condition is false initially.

Putting it all together, the loop starts with `i` equal to 0 and continues to execute the loop body (printing the value of `i` and incrementing `i` by 1) until `i` is no longer less than 10. The output will be the numbers 0 through 9, each on a new line.

Here's what happens in each iteration:

- **1st iteration**: `i` is 0, `console.log(i)` prints 0, `i` becomes 1.
- **2nd iteration**: `i` is 1, `console.log(i)` prints 1, `i` becomes 2.
- **3rd iteration**: `i` is 2, `console.log(i)` prints 2, `i` becomes 3.
- ...
- **10th iteration**: `i` is 9, `console.log(i)` prints 9, `i` becomes 10.

After the 10th iteration, `i` becomes 10, which makes the condition `i < 10` false, so the loop terminates.

While Loop

Finally, let's quickly discuss the "while" statement:

```
while (condition)
  statement
```

For example:

```
let i = 0;

while (i < 10) {
  console.log(i);
  i++;
```

You will notice that this is very similar to the `do...while` loop, however the primary distinction lies in the condition check. In a `while` loop, the condition is evaluated before executing the block of code inside the loop. This means that if the condition evaluates to false at the start, the loop body will never execute. Conversely, a `do...while` loop guarantees that the loop body will execute at least once because the condition is checked after the execution of the loop body.

Functions

A function is a *subprogram*, or a smaller portion of code that can be called (i.e., invoked) by another part of your program, another function, or by the environment in response to some user or device action (e.g., clicking a button, a network request, the page closing). Functions *can* take values (i.e., arguments) and may *return* a value.

Functions are first-class members of JavaScript, and play a critical role in developing JavaScript programs. JavaScript functions can take other functions as arguments, can return functions as values, can be bound to variables or `Object` properties, and can even have their own properties. We'll talk about more of this when we visit JavaScript's object-oriented features.

Learning to write code in terms of functions takes practice. JavaScript supports functional programming. Web applications are composed of lots of small components that need to get wired together using functions, have to share data (i.e., state), and interoperate with other code built into the browser, or in third-party frameworks, libraries, and components.

We use JavaScript functions in a number of ways. First, we encapsulate a series of statements into higher-order logic, giving a name to a set of repeatable steps we can call in different ways and places in our code. Second, we use them to define actions to be performed in response to events, whether user initiated or triggered by the browser. Third, we use them to define behaviours for objects, what is normally called a *member function* or *method*. Fourth, we use them to define *constructor* functions, which are used to create new objects. We'll look at all of these in the coming weeks.

Before we dive into that, we'll try to teach you that writing many smaller functions is often better than having a few large ones. Smaller code is easier to

test, easier to understand, and generally has fewer bugs.

User-defined Functions

JavaScript has many built-in functions, which we'll get to later on in these notes; however, it also allows you to write your own and/or use functions written by other developers (libraries, frameworks).

These user-defined functions can take a number of forms.

Function Declarations

The first is the *function declaration*, which looks like this:

```
// The most basic function, a so-called NO OPERATION function
function noop() {}

// square function accepts one parameter `n`, returns its value
squared.
function square(n) {
  return n * n;
}

// add function accepts two parameters, `a` and `b`, returns their
sum.
function add(a, b) {
  return a + b;
}
```

Here the `function` keyword initiates a *function declaration*, followed by a *name*, a *parameter list* in round parenthesis, and the function's *body* surrounded by curly braces. There is no semi-colon after the function body.

Function Expressions

The second way to create a function is using a *function expression*. Recall that expressions evaluate to a value: a function expression evaluates to a `function` Object. The resulting value is often bound (i.e., assigned) to a variable, or used as a parameter.

```
let noop = function () {};

let square = function (n) {
  return n * n;
};

let add = function add(a, b) {
  return a + b;
};
```

A few things to note:

- The function's *name* is often omitted. Instead we return an *anonymous function* and bind it to a variable. We'll access it again via the variable name. In the case of recursive functions, we sometimes include it to make it easier for functions to call themselves. You'll see it done both ways.
- We *did* use a semi-colon at the end of our function expression. We do this to signify the end of our assignment statement `let add = ... ;`.
- In general, *function declarations* are likely a better choice (when you can choose) due to subtle errors introduced with declaration order and hoisting; however, both are used widely and are useful.

Arrow Functions

Modern JavaScript also introduces a new function syntax called an Arrow Function or "Fat Arrow". These functions are more terse, using the `=>` notation (not to be confused with the `<=` and `>=` comparison operators):

```
let noop = () => {};

let square = (n) => n * n;

let add = (a, b) => a + b;
```

When you see `let add = (a, b) => a + b;` it is short-hand for `let add = function(a, b) { return a + b; }`, where `=>` replaces the `function` keyword and comes *after* the parameter list, and the `return` keyword is optional, when functions return a single value.

Arrow functions also introduce some new semantics for the `this` keyword, which we'll address later.

You should be aware of Arrow functions, since many web developers use them heavily. However, don't feel pressure to use them yet if you find their syntax confusing.

Parameters and arguments

Function definitions in both cases take parameter lists, which can be empty, single, or multiple in length. Just as with variable declaration, no type information is given:

```
function emptyParamList() {}

function singleParam(oneParameter) {}

function multipleParams(one, two, three, four) {}
```

A function can *accept* any number of arguments when it is called, including none. This would break in many other languages, but not JavaScript:

```
function log(a) {
  console.log(a);
}

log('correct'); // logs "correct"
log('also', 'correct'); // logs "also"
log(); // logs undefined
```

Because we can invoke a function with any number of arguments, we have to write our functions carefully, and test things before we make assumptions. How can we deal with a caller sending 2 vs. 10 values to our function?

One way we do this is using the built-in `arguments` Object.

Every function has an implicit `arguments` variable available to it, which is an array-like object containing all the arguments passed to the function. We can use `arguments.length` to obtain the actual number of arguments passed to the function at runtime, and use array index notation (e.g., `arguments[0]`) to access an argument:

```
function log(a) {
  console.log(arguments.length, a, arguments[0]);
}
```

We can use a loop to access all arguments, no matter the number passed:

```
function sum() {
  const count = arguments.length;
  let total = 0;
  for (let i = 0; i < count; i++) {
    total += arguments[i];
  }
  return total;
}

sum(1); // 1
sum(1, 2); // 3
sum(1, 2, 3, 4); // 10
```

You may have wondered previously how `console.log()` can work with one, two, three, or more arguments. The answer is that all JavaScript functions work this way, and you can use it to "overload" your functions with different argument patterns, making them useful in more than one scenario.

Parameters and "..."

Modern JavaScript also supports naming the "rest" of the parameters passed to a function. These Rest Parameters allow us to specify that all final arguments to a function, no matter how many, should be available to the function as a named `Array`.

There are some advantages to *not* using the implicit `arguments` keyword, which rest parameters provide.

We can convert the example above to this, naming our arbitrary list of "numbers":

```
function sum(...numbers) {
  let total = 0;
  for (let i = 0; i < numbers.length; i++) {
    total += numbers[i];
  }
  return total;
}
```

Dealing with Optional and Missing Arguments

Because we *can* change the number of arguments we pass to a function at runtime, we also have to deal with missing data, or optional parameters. Consider the case of a function to calculate a player's score in a video game. In some cases we may wish to provide an optional bonus to the score, for example:

```
function updateScore(currentScore, value, bonus) {
  return bonus ? currentScore + value * bonus : currentScore +
value;
}

updateScore(10, 3);
updateScore(10, 3, 2);
```

Here we call `updateScore` first with 2 arguments, and then once with 3. Our `updateScore` function has been written so it will work in both cases. We've used a conditional ternary operator to decide whether or not to add an extra bonus score. When we say `bonus ? ... : ...` we are checking to see if the `bonus` argument is *truthy* or *falsy*--that is, did the caller provide a value for it? If they did, we do one thing, if not, we do another.

Here's another common way you'll see code like this written, using a default

value:

```
function updateScore(currentScore, value, bonus) {
  // See if `bonus` is truthy (has a value or is undefined) and
use it, or default to 1
  bonus = bonus || 1;
  return currentScore + value * bonus;
}
```

In this case, before we use the value of `bonus`, we do an extra check to see if it actually has a value or not. If it does, we use that value as is; but if it doesn't, we instead assign it a value of `1`. Then, our calculation will always work, since multiplying the value by `1` will be the same as not using a bonus.

The idiom `bonus = bonus || 1` is very common in JavaScript. It uses the Logical Or Operator `||` to test whether `bonus` evaluates to a value or not, and prefers that value if possible to the fallback default of `1`. We could also have written it out using an `if` statements like these:

```
function updateScore(currentScore, value, bonus) {
  if (bonus) {
    return currentScore + value * bonus;
  }
  return currentScore + value;
}

function updateScore(currentScore, value, bonus) {
  if (!bonus) {
    bonus = 1;
  }
  return currentScore + value * bonus;
}
```

JavaScript programmers tend to use the `bonus = bonus || 1` pattern because it is less repetitive, using less code, and therefore less likely to introduce bugs. We could shorten it even further to this:

```
function updateScore(currentScore, value, bonus) {
  return currentScore + value * (bonus || 1);
}
```

Because this pattern is so common, modern JavaScript has added a built-in way to handle Default Parameters. Instead of using `||` notation in the body of the function, we can specify a default value for any named parameter when it is declared. This frees us from having to check for, and set default values in the function body. Using default parameters, we could convert our code above to this:

```
function updateScore(currentScore, value, bonus = 1) {
  return currentScore + value * bonus;
}
```

Now, if `bonus` has a value (i.e., is passed as a parameter), we use it; otherwise, we use `1` as a default.

Return Value

Functions always *return* a value, whether implicitly or explicitly. If the `return` keyword is used, the expression following it is returned from the function. If it is omitted, the function will return `undefined`:

```
function implicitReturnUndefined() {
  // no return keyword, the function will return `undefined`
```

Function Naming

Functions are typically named using the same rules we learned for naming any variable: `camelCase` and using the set of valid letters, numbers, etc. and avoiding language keywords.

Function declarations always give a name to the function, while function expressions often omit it, using a variable name instead:

```
// Name goes after the `function` keyword in a declaration
function validateUser() {
    ...
}

// Name is used only at the level of the bound variable, function
is anonymous
let validateUser = function() {
    ...
};

// Name is repeated, which is correct but not common. Used with
recursive functions
let validateUser = function validateUser() {
    ...
};

// Names are different, which is also correct, but not common as
it can lead to confusion
let validateUser = function validate() {
    // the validate name is only accessible here, within the
function body
    ...
};
```

Because JavaScript allows us to bind function objects (i.e., result of function expressions) to variables, it is common to create functions without names, but immediately pass them to functions as arguments. The only way to use this function is via the argument name:

```
// The parameter `fn` will be a function, and `n` a number
function execute(fn, n) {
  // Call the function referred to by the argument (i.e, variable)
`fn`, passing `n` as its argument
  return fn(n);
}

// 1. Call the `execute` function, passing an anonymous function,
which squares its argument, and the value 3
execute(function (n) {
  return n * n;
}, 3);

// 2. Same thing as above, but with different formatting
execute(function (n) {
  return n * n;
}, 3);

// 3. Same thing as above, using an Arrow Function
execute((n) => n * n, 3);

let doubleIt = function (num) {
  return num * 2;
};

// 4. Again call `execute`, but this time pass `doubleIt` as the
function argument
execute(doubleIt, 3);
```

We can also use functions declared via function declarations used this way,

and bind them to variables:

```
function greeting(greeting, name) {
  return greeting + ' ' + name;
}

var sayHi = greeting; // also bind a reference to greeting to
sayHi

// We can now call `greeting` either with `greeting()` or
`sayHi()`
console.log(greeting('Hello', 'Steven'));
console.log(sayHi('Hi', 'Kim'));
```

JavaScript treats functions like other languages treat numbers or booleans, and lets you use them as values. This is a very powerful feature, but can cause some confusion as you get started with JavaScript.

You might ask why we would ever choose to define functions using variables. One common reason is to swap function implementations at runtime, depending on the state of the program. Consider the following code for displaying the user interface depending on whether the user is logged in or not:

```
// Display partial UI for guests and non-authenticated users,
hiding some features
function showUnauthenticatedUI() {
    ...
}

// Display full UI for authenticated users
function showAuthenticatedUI() {
    ...
```

Invoking Functions, the Execution Operator

In many of the examples above, we've been invoking (i.e., calling, running, executing) functions but haven't said much about it. We invoke a function by using the `()` operator:

```
let f = function () {
  console.log('f was invoked');
};
f();
```

In the code above, `f` is a variable that is assigned the value returned by a function expression. This means `f` is a regular variable, and we can use it like any other variable. For example, we could create another variable and share its value:

```
let f = function () {
  console.log('f was invoked');
};
let f2 = f;
f(); // invokes the function
f2(); // also invokes the function
```

Both `f` and `f2` refer to the the same function object. What is the difference between saying `f` vs. `f()` in the line `let f2 = f;`? When we write `f()` we are really saying, "Get the value of `f` (the function referred to) and invoke it." However, when we write `f` (without `()`), we are saying, "Get the value of `f` (the function referred to)" so that we can do something with it (assign it to another variable, pass it to a function, etc).

The same thing is true of function declarations, which also produce `function`

Objects:

```
function f() {
  console.log('f was invoked');
}
let f2 = f;
f2(); // also invokes the function
```

The distinction between referring to a function object via its bound variable name (`f`) vs invoking that same function (`f()`) is important, because JavaScript programs treat functions as *data*, just as you would a `Number`. Consider the following:

```
function checkUserName(userName, customValidationFn) {
  // If `customValidationFn` exists, and is a function, use that
to validate `userName`
  if (customValidationFn && typeof customValidationFn ===
'function') {
    return customValidationFn(userName);
  }
  // Otherwise, use a default validation function
  return defaultValidationFn(userName);
}
```

Here the `checkUserName` function takes two arguments: the first a `String` for a username; the second an optional (i.e., may not exist) function to use when validating this username. Depending on whether or not we are passed a function for `customValidationFn`, we will either use it, or use a default validation function (defined somewhere else).

Notice the line `if(customValidationFn && typeof customValidationFn === 'function') {` where `customValidationFn` is used like any other variable

(accessing the value it refers to vs. doing an invocation), to check if it has a value, and if its value is actually a function. Only then is it save to invoke it.

It's important to remember that JavaScript functions aren't executed until they are called via the invocation operator, and may also be used as values without being called.

Example Code

You may download the sample code for this topic here:

> JavaScript-Fundamentals

Built in Objects

As we have seen, JavaScript is a flexible and widely adopted programming language. It provides a wealth of integrated objects that enable numerous capabilities across both client-side and server-side environments. These objects encompass fundamental JavaScript operations such as working with a String, Array, Map, Math, Date, Error, etc... We also have Node.js-specific objects tailored for managing file systems, processing HTTP requests, and other server-side operations.

Let's begin by discussing some of the built-in objects that come bundled with the JavaScript language, as well as some of the key functions that exist in the Node.js ecosystem.

String

Here are a few examples of how you can declare a String in JavaScript, first using a string literal (which we have already discussed), followed by a call to the `new` operator and the `String` object's constructor function:

```
/*
 * JavaScript String Literals
 */
let s = 'some text'; // single-quotes
let s1 = 'some text'; // double-quotes
let s2 = `some text`; // template literal using back-ticks
let unicode =
  '
```

If we want to convert other types to a `String`, we have a few options:

```
let x = 17;
let s = '' + x; // concatenate with a string (the empty string)
let s2 = String(x); // convert to String. Note: the `new` operator
is not used here
let s3 = x.toString(); // use a type's .toString() method
```

Whether you use a literal or the constructor function, in all cases you will be able to use the various functionality of the `String` type.

Common Properties and Methods

- `s.length` - will tell us the length of the string (UTF-16 code units)
- `s.charAt(1)` - returns the character at the given position (UTF-16 code unit). We can also use `s[1]` and use an index notation to get a particular character from the string.
- `s.concat()` - returns a new string created by concatenating the original with the given arguments.
- `s.padStart(2, '0)` - returns a new string padded with the given substring until the length meets the minimum length given. See also `s.padEnd()`.
- `s.includes("tex")` - returns `true` if the search string is found within the string, otherwise `false` if not found.
- `s.startsWith("some")` - returns `true` if the string starts with the given substring, otherwise `false`.
- `s.endsWith("text")` - returns `true` if the string ends with the given

substring, otherwise `false`.

- `s.indexOf("t")` - returns the first index position of the given substring within `s`, or `-1` if the substring is not found within `s`. See also `s.lastIndexOf()`

- `s.match(regex)` - tries to match a regular expression against the string, returning the matches.

- `s.replace(regex, "replacement")` - returns a new string with the first occurrence of a matched RegExp replaced by the replacement text. See also `s.replaceAll()`, which replaces *all* occurrences.

- `s.slice(2, 3)` - returns a new string extracted (sliced) from within the original string. A beginning index and (optional) end index mark the position of the slice.

- `s.split()` - returns an Array (see discussion below) of substrings by splitting the original string based on the given separator (`String` or `RegExp`).

- `s.toLowerCase()` - returns a new string with all characters converted to lower case.

- `s.toUpperCase()` - returns a new string with all characters converted to upper case.

- `s.trim()` - returns a new string with leading and trailing whitespace removed.

> **NOTE:** JavaScript also supports template literals, also sometimes called template *strings*. Template literals use back-ticks instead of single- or

> double-quotes, and allow you to interpolate JavaScript expressions. For example:
>
> ```
> let a = 1;
> let s = 'The value is ' + 1 * 6;
> // Use ${...} to interpolate the value of an expression into a string
> let templateVersion = `The value is ${1 * 6}`;
> ```

Array

An `Array` is also technically an `Object` with various properties and methods we can use for working with lists in JavaScript.

Declaring JavaScript Arrays

Like creating a `String`, we can create an `Array` in JavaScript using either a literal (which we have seen) or the `Array` constructor function:

```
let arr = new Array(1, 2, 3); // array constructor
let arr2 = [1, 2, 3]; // array literal
```

Accessing Elements in an Array

We can use index notation to obtain an element at a given index:

```
let numbers = [50, 12, 135];
let firstNumber = numbers[0];
let lastNumber = numbers[numbers.length - 1];
```

JavaScript also allows us to use a technique called Destructuring Assignment to unpack values in an Array into distinct variables. Consider each of the following methods, both of which accomplish the same goal:

```
// Co-ordinates for Seneca's Newnham Campus
let position = [43.796, -79.3486];

// Separate the two values into their own unique variables.

// Version 1 - index notation
let lat = position[0];
let lng = position[1];

// Version 2 - destructure
let [lat, lng] = position;
```

This technique is useful when working with structured data, where you know exactly how many elements are in an array, and need to access them:

```
let dateString = `17/02/2001`;
let [day, month, year] = dateString.split('/');
console.log(`The day is ${day}, month is ${month}, and year is
${year}`);
```

Here we `.split()` the string `'17/02/2001'` at the `'/'` character, which will produce the Array `['17', '02', '2001']`. Next, we destructure this Array's values into the variables `day`, `month`, `year`.

You can also ignore values (i.e., only unpack the one or ones you want):

```
let dateString = `17/02/2001`;
// Ignore the first index in the array, unpack only position 1 and
```

Common Properties and Methods

- `arr.length` - a property that tells us the number of elements in the array.

Methods that modify the original array

- `arr.push(element)` - a method to add one (or more) element(s) to the end of the array. Using `push()` modifies the array (increasing its size). You can also use `arr.unshift(element)` to add one (or more) element to the *start* of the array.
- `arr.pop()` - a method to remove the last element in the array and return it. Using `pop()` modifies the array (reducing its size). You can also use `arr.shift()` to remove the *first* element in the array and return it.

Methods that do not modify the original array

- `arr.concat([4, 5], 6)` - returns a new array with the original array joined together with other arrays or values provided.
- `arr.includes(element)` - returns `true` if the array includes the given element, otherwise `false`.
- `arr.indexOf(element)` - returns the index of the given element in the array, if it exists, otherwise `-1` (meaning not found).
- `arr.join("\n")` - returns a string created by joining (concatenating) all elements in the array with the given delimiter (`String`).

Methods for iterating across the elements in an Array

JavaScript's `Array` type also provides a long list of useful methods for working with list data. All of these methods work in roughly the same way:

```
// Define an Array
let list = [1, 2, 3, 4];

// Define a function that you want to call on each element of the
array
function operation(element) {
  // do something with element...
}

// Call the Array method that you want, passing your function
operation
list.arrayOperation(operation);
```

JavaScript will call the given function on every element in the array, one after the other. Using these methods, we are able to work with the elements in an Array instead of only being able to do things with the Array itself.

As a simple example, let's copy our `list` Array and add 3 to every element. We'll do it once with a for-loop, and the second time with the `forEach()` method:

```
// Create a new Array that adds 3 to every item in list, using a
for-loop
let listCopy = [];

for (let i = 0; i < list.length; i++) {
  let element = list[i];
  element += 3;
  listCopy.push(element);
}
```

Now the same code using the `Array`'s `forEach()` method:

```
let listCopy = [];

list.forEach(function (element) {
  listCopy.push(element + 3);
});
```

We've been able to get rid of all the indexing code, and with it, the chance for off-by-one errors. We also don't have to write code to get the element out of the list: we just use the variable passed to our function.

These `Array` methods are so powerful that there are often functions that do exactly what we need. For example, we could shorten our code above even further but using the `map()` method. The `map()` method takes one `Array`, and calls a function on every element, creating and returning a new `Array` with those elements:

```
let listCopy = list.map(function (element) {
  return element + 3;
});
```

Here are some of the `Array` methods you should work on learning:

- `arr.forEach()` - calls the provided function on each element in the array.
- `arr.map()` - creates and returns a new array constructed by calling the provided function on each element of the original array.
- `arr.find()` - finds and returns an element from the array which matches a condition you define. See also `arr.findLast()`, `arr.findIndex()`, and `arr.findLastIndex()`, which all work in similar ways.
- `arr.filter()` - creates and returns a new array containing only those elements that match a condition you define in your function.

- `arr.every()` - returns `true` if all of the elements in the array meet a condition you define in your function.

There are more `Array` methods you can learn as you progress with JavaScript, but these will get you started.

Iterating over String, Array, and other collections

The most familiar way to iterate over a `String` or `Array` works as you'd expect:

```
let s = 'Hello World!';
for (let i = 0; i < s.length; i++) {
  let char = s.charAt(i);
  console.log(i, char);
  // Prints:
  // 0, H
  // 1, e
  // 2, l
  // ...
}

let arr = [10, 20, 30, 40];
for (let i = 0; i < arr.length; i++) {
  let elem = arr[i];
  console.log(i, elem);
  // Prints:
  // 0, 10
  // 1, 20
  // 2, 30
  // ...
}
```

The standard `for` loop works, but is not the best we can do. Using a `for` loop is prone to various types of errors: off-by-one errors, for example. It also requires extra code to convert an index counter into an element.

An alternative approach is available in ES6, `for...of`:

```
let s = 'Hello World!';
for (let char of s) {
  console.log(char);
  // Prints:
  // H
  // e
  // l
  // ...
}

let arr = [10, 20, 30, 40];
for (let elem of arr) {
  console.log(elem);
  // Prints:
  // 10
  // 20
  // 30
  // ...
}
```

Using `for...of` we eliminate the need for a loop counter altogether, which has the added benefit that we'll never under- or over- shoot our collection's element list; we'll always loop across exactly the right number of elements within the given collection.

The `for...of` loop works with all collection types, from `String` to `Array` to `arguments` to `NodeList` (as well as newer collection types like `Map`, `Set`, etc.).

Date

The Date object represents a single moment in time and provides methods to work with dates and times.

Creating Dates

We can create a new Date object Using the new Date() Constructor

```
let now = new Date(); // Current date and time
```

If we wish to set a specific date and time, ie: "May 21st 2024 at 12:34:56" (using a 24-hour format), we have a number of options:

```
let specificDate = new Date('May 21, 2024 12:34:56'); //
DISCOURAGED: may not work in all runtimes
let specificDate = new Date('2024-05-21T12:34:56'); // This is
standardized and will work reliably
let specificDate = new Date(2024, 4, 21); // The month is
0-indexed
let specificDate = new Date(2024, 4, 21, 12, 34, 56); // Also
passing hours, minutes & seconds
let specificDate = new Date(1684639396000); // Passing epoch
timestamp
```

Common Properties and Methods

`let currentTime = new Date();` - will give us the current date / time

Methods to Set Individual Parts of the Date Object

```
let date = new Date();

date.setFullYear(2024); // Sets the year
date.setMonth(5); // Sets the month (0-based)
date.setDate(21); // Sets the day
date.setHours(12); // Sets the hour
date.setMinutes(34); // Sets the minutes
date.setSeconds(56); // Sets the seconds

// etc... (https://developer.mozilla.org/en-US/docs/Web/
JavaScript/Reference/Global_Objects/Date)
```

Methods to Get Individual Parts of the Date Object:

```
let date = new Date();

console.log(date.getFullYear()); // Gets the year
console.log(date.getMonth()); // Gets the month (0-based)
console.log(date.getDate()); // Gets the day
console.log(date.getHours()); // Gets the hour
console.log(date.getMinutes()); // Gets the minutes
console.log(date.getSeconds()); // Gets the seconds

// etc... (https://developer.mozilla.org/en-US/docs/Web/
JavaScript/Reference/Global_Objects/Date)
```

Methods to Output the Full Date String

```
let date = new Date();

console.log(date.toString()); // outputs a string representing
this date interpreted in the local timezone
console.log(date.toDateString()); // outputs a string representing
```

Comparing Dates

To compare dates, we can use our familiar "greater than" / "less than", "equals", etc, operators:

```
let date1 = new Date(2024, 5, 21);
let date2 = new Date(2024, 5, 22);

if (date1 < date2) {
  console.log('Date1 is earlier than Date2');
} else if (date1 > date2) {
  console.log('Date1 is later than Date2');
} else {
  console.log('Dates are equal');
}
```

Node.js Globals

The following section outlines some global objects, functions and variables that ship with the Node.js / JavaScript language.

console

The console object provides a simple debugging console that is similar to the JavaScript console mechanism provided by web browsers.

Some of the key methods that we will be using are:

- console.log()
- console.time() / console.timeEnd()

- console.dir()

__dirname

__dirname is used to obtain name of the directory that the currently executing script resides in.

For example: if our .js file is located in /Users/pcrawford/ex1.js:

```
console.log(__dirname);
// outputs /Users/pcrawford
```

__filename

__filename is be used to obtain file containing the code being executed as well as the directory. This is the resolved absolute path of this code file.

For example: if our .js file is located in /Users/pcrawford/ex1.js:

```
console.log(__filename);
// outputs /Users/pcrawford/ex1.js
```

setTimeout()

The setTimeout() function will execute a piece of code (function) after a certain delay. It accepts 3 parameters:

- **callback** Function: The function to call when the timer elapses.
- **delay** number: The number of milliseconds to wait before calling the callback

- **[, ...arg]** Optional arguments to pass when the callback is called.

For example:

```
// outputs "Hello after 1 second" to the console
setTimeout(function () {
  console.log('Hello after 1 second');
}, 1000);
```

setInterval()

The setInterval() function will execute a piece of code (function) after a certain delay and continue to call it repeatedly. It accepts 3 parameters (below) and returns a timeout object

- **callback** Function: The function to call when the timer elapses.
- **delay** number: The number of milliseconds to wait before calling the callback
- **[, ...arg]** Optional arguments to pass when the callback is called.

Note: Unless you want the interval to continue forever, you need to call clearInterval() with the timeout object as a parameter to halt the interval

For example:

```
let count = 1; // global counter
let maxCount = 5; // global maximum

let myCountInterval = setInterval(function () {
  console.log('Hello after ' + count++ + ' second(s)');
  checkMaximum();
}, 1000);
```

URL

The URL class is used to create a new URL object by parsing the full URL string, ie:

```
let myURL = new URL('https://myProductInventory.com/
products?sort=asc&onSale=true');
```

Once we have a new URL object, we can access / modify aspects of it via their associated properties:

```
console.log(myURL);

/*
URL {
  href: 'https://myproductinventory.com/
products?sort=asc&onSale=true',
  origin: 'https://myproductinventory.com',
  protocol: 'https:',
  username: '',
  password: '',
  host: 'myproductinventory.com',
  hostname: 'myproductinventory.com',
  port: '',
  pathname: '/products',
  search: '?sort=asc&onSale=true',
  searchParams: URLSearchParams { 'sort' => 'asc', 'onSale' =>
'true' },
  hash: ''
*/
```

To access the parsed query parameters (ie the "search" property), we can use

a "for...of" loop to iterate over key-value pairs the "searchParams": property:

```
for (const [key, value] of myURL.searchParams) {
  console.log('key: ' + key + ' value: ' + value);
}

/*
key: sort value: asc
key: onSale value: true
*/
```

Error

The Error object is used to represent an error ("exception") that occurs during the execution of our code. It provides a way to capture and handle the exception with useful information such as the error message and the stack trace.

Handling Errors

Consider the following example that generates the runtime error: `TypeError: Assignment to constant variable.`:

```
const PI = 3.14159;

console.log('trying to change PI!');

PI = 99;

console.log('Haha! PI is now: ' + PI);
```

Here, we are trying to change the value of a constant: PI. If we try to run this short program in Node.js, the program will crash before we get a chance to see

the string "Haha! PI is now: 99", or even "Haha! PI is now: 3.14159". There is no elegant recovery and we do not exit the program gracefully.

Fortunately, before our program crashes in such a way, Node.js will "throw" an "Error" object that we can intercept using the "try...catch" statement:

```
const PI = 3.14159;

console.log('trying to change PI!');

try {
  PI = 99;
} catch (ex) {
  console.log('uh oh, an error occurred: ' + ex.message);
  // outputs: uh oh, an error occurred: Assignment to constant
variable.
}

console.log('Alas, it cannot be done, PI remains: ' + PI);
```

By utilizing properties such as Error.message & Error.stack, we can gain further insight to exactly what went wrong and we can either refactor our code to remedy the error, or acknowledge that the error will happen and handle it gracefully.

"Throwing" custom Errors

We can also "throw" our own error using the "throw" keyword with a new instance of the "Error" class, for example:

```
function divide(x, y) {
  if (y == 0) {
    throw new Error('Division by Zero!');
  }
```

Custom Objects

As we have seen from ou discussion on "Built in Objects", Javascript is "Object Oriented":

> "Object-oriented programming is about modeling a system as a collection of objects, where each object represents some particular aspect of the system. Objects contain both functions (or methods) and data. An object provides a public interface to other code that wants to use it but maintains its own private, internal state; other parts of the system don't have to care about what is going on inside the object."
>
> https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_programming

Object Literal Notation

The most simple and straight-forward way to create an object in JavaScript is to use "Object Literal Notation" (sometimes referred to as "object initializer" notation). The syntax for creating an object using this notation is as follows:

```
let obj = {
  property_1: value_1,
  property_2: value_2,
  // ...,
  'property n': value_n,
}; // properties can also be defined as a string`
```

So, if we wanted to create an object with the following properties:

- **name** (string)
- **age** (number)
- **occupation** (string)

and methods...

- **setName** ("setter" to set a new value for the "name" property)
- **setAge** ("setter" to set a new value for the "age" property)
- **getName** ("getter" to get the current value of the "name" property)
- **getAge** ("getter" to get the current value of the "age" property)

using "Object Literal" notation, we would write the code:

```
let architect = {
  name: 'Joe',
  age: 34,
  occupation: 'Architect',

  setName: function (newName) {
    this.name = newName;
  },

  setAge: function (newAge) {
    this.age = newAge;
  },

  getName: function () {
    return this.name;
  },

  getAge: function () {
    return this.age;
  },
};
```

and access the data (properties) and functions (methods) using the following code, ie:

```
console.log(architect.name); // "Joe"
// or
console.log(architect.getName()); // "Joe"
```

We must use the **“this”** keyword whenever we refer to one of the properties of the object inside one of it’s methods. This is due to the fact that when a method is executed, "age" (for example) might already exist in the global scope, or within the scope of the function as a local variable. To be absolutely sure that we are referring to the correct "age" property of the current object, we must refer to the "execution context" - ie: the object that is actually making a call to this method. We know the object has an "age" property, so in order to be more specific about *which* age variable that we want to change, we leverage the keyword **this**. "this" will refer to the "execution context", ie: the object that called the function! So, **"this.age"** can be read literally as **"the age property on this object"**, which is exactly the property that we wish to edit.

However, while "this" allows us to be specific with which **properties** that we refer to in our **methods**, it can lead to some confusing scenarios. For example, what if we added a new "outputNameDelay()" method to our architect object that writes the architect's name to the console after 1 second (1000 milliseconds):

```
// ...
outputNameDelay: function(){
  setTimeout(function(){
    console.log(this.name);
  },1000);
}
```

Everything looks correct and we have made proper use of the "this", however because the setTimeout function is not executed as a method of our architect object, we end up with "undefined" as output to the console. There are a number of fixes for this issue, however the most common is by using an "arrow function" as we have seen when first discussing JavaScript. The reason that this works is because Arrow functions use a "lexical" this (ie: the "this" value of the parent scope).

```
// ...
outputNameDelay: function(){
    setTimeout(() => {
      console.log(this.name);
    },1000);
}
// ...
architect.outputNameDelay(); // outputs "Joe"
```

The "class" keyword

If we wish to create multiple objects of the same "type" (ie: that have the same properties and methods, but with different values), we can leverage the "class" and "new" keywords, ie:

```
class Architect {
  name;
  age;
  occupation = 'architect'; // default value of "architect" for
occupation

  constructor(setName = '', setAge = 0) { // handle missing
parameters with '' and 0
    this.name = setName;
```

Here, we specify the properties (with default values), a "constructor" function to take initialization parameters, as well as specify all of the methods within the "class" block.

> **NOTE:** For more information / advanced topics on objects in JavaScript, such as: "Private Methods / Properties", "Getters / Setters" and "Inheritance" see the documentation on MDN.

Example Code

You may download the sample code for this topic here:

> Objects-In-JavaScript

Callbacks

Before we begin to discuss "callbacks" and other methods for working with asynchronous logic within our programs, we should first define what "asynchronous programming" is:

> "Asynchronous programming is a technique that enables your program to start a potentially long-running task and still be able to be responsive to other events while that task runs, rather than having to wait until that task has finished. Once that task has finished, your program is presented with the result."
>
> https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing

This means that potentially long-running tasks will not cause delays within our main execution logic. However, it also means that we need to find a way to execute code when a long-running task has completed (ie: connecting to a database, reading a file, etc).

As a simple example of how JavaScript works with asynchronous code, we can refer to the global "setTimeout" function; here, we will wait 2 seconds (2000 milliseconds) and execute some code *before* and *after* the function:

```
console.log('Hello');

setTimeout(() => {
  console.log('World');
}, 2000);

console.log('!');
```

Here, we see the text output to the console is out of order, ie: "Hello" followed by "!" and (2 seconds later) we finally see the text "World". This is because the "setTimeout" function is "asynchronous" and will not cause the main flow of execution to wait (2 seconds) for it to complete. The *function* that is passed in the first parameter of "setTimeout" (which is responsible for outputting "World" to the console) is a **callback** function:

> "a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action."

Defining Functions with Callbacks

Now that we know that a callback is really just a function passed to another function to perform an action once some asynchronous logic is complete, let's try writing our own code. Here, we will be using the setTimeout() function to approximate an asynchronous action such as connecting to a database.

For our first example, let's say that we have a function called "connectToDatabase" that establishes a database connection after a random amount of time (between 1 and 2000 milliseconds). We also have a function called "queryData" that also takes a random amount of time to complete (in this case, it is between 1 and 1000 milliseconds).

```
function connectToDatabase() {
  let randomTime = Math.floor(Math.random() * 2000) + 1;

  setTimeout(() => {
    console.log('Connection Established');
  }, randomTime);
}
```

For our code to work correctly, we must first connect to the database, then query the data. To accomplish this, we would intuitively write the code to invoke the functions in order, ie:

```
connectToDatabase();
queryData();
```

However, this poses a problem as there's no way to ensure that the logic to connect to the database happens **before** the query. In fact, since it takes longer to connect to the database, it's more likely that the query logic will complete first.

One way to solve this problem is to provide the "queryData()" function as a **callback** function to "connectToDatabase()" to be executed once the connection has been established:

```
function connectToDatabase(queryFunction) {
  let randomTime = Math.floor(Math.random() * 2000) + 1;

  setTimeout(() => {
    console.log('Connection Established');
    queryFunction();
  }, randomTime);
}
```

Notice how we have added "queryFunction" as a parameter to the connectToDatabase() function. Once the connection has been established, we manually invoke the function using "()".

Now, we can ensure that the functions are executed in order, using the code:

```
connectToDatabase(queryData);
```

Adding Parameters

As our code stands now, the "queryData" function is very simple and does not take any parameters. Why don't we try making it a little more dynamic by adding parameters to it, so that a query can be provided:

```
function queryData(query) {
  let randomTime = Math.floor(Math.random() * 1000) + 1;

  setTimeout(() => {
    console.log(query);
  }, randomTime);
}
```

Now we can invoke our queryData with a given query, for example:

```
queryData('select * from Employees');
```

However, a problem occurs when we attempt to provide the "queryData" function as a **callback** to another function (in our case, the "connectToDatabase" function):

```
connectToDatabase(queryData('select * from Employees')); //
TypeError: queryFunction is not a function
```

This is because the "()" syntax after the function name causes the function to *execute* which then passes its return value ("undefined") to the

connectToDatabase function. To solve this, we must pass the parameters to the "queryData()" callback function, as parameters to the "connectToDatabase()" function:

```
function connectToDatabase(queryFunction, query) {
  let randomTime = Math.floor(Math.random() * 2000) + 1;

  setTimeout(() => {
    console.log('Connection Established');
    queryFunction(query);
  }, randomTime);
}
```

Here, you can see that we have added the "query" as a 2nd parameter to the connectToDatabase function and use it as a parameter to the "queryFunction".

Putting it all together, we get:

```
function connectToDatabase(queryFunction, query) {
  let randomTime = Math.floor(Math.random() * 2000) + 1;

  setTimeout(() => {
    console.log('Connection Established');
    queryFunction(query);
  }, randomTime);
}

function queryData(query) {
  let randomTime = Math.floor(Math.random() * 1000) + 1;

  setTimeout(() => {
    console.log(query);
  }, randomTime);
}
```

Promises & Async / Await

As we have seen in our "callbacks" discussion, JavaScript is "asynchronous" in nature. Code can be written to respond to events or wait for tasks to complete before executing. One way of handling such situations was to enclose our "follow up" logic in a function that may be passed to another function to be executed (typically, after some asynchronous logic has completed such as connecting to a database, or reading a file).

Callback Review

As a quick review of the callback logic discussed earlier, consider the following three functions:

```
// output "A" after a random time between 0 & 3 seconds
function outputA() {
  let randomTime = Math.floor(Math.random() * 3000) + 1;

  setTimeout(() => {
    console.log('A');
  }, randomTime);
}

// output "B" after a random time between 0 & 3 seconds
function outputB() {
  let randomTime = Math.floor(Math.random() * 3000) + 1;

  setTimeout(() => {
    console.log('B');
```

If we were to execute them in order, ie:

```
outputA();
outputB();
outputC();
```

we would have no idea which letter would be output to the console first ("A", "B", or "C"), since each function takes a random amount of time to complete. If however, we wanted to be absolutely sure that the output of the code is in the correct order ("A", "B", "C") regardless of how long it takes each function to execute, we must ensure that the "follow up" functions are passed as parameters to the functions with the asynchronous logic (ie: "callbacks"). This case is more complicated because we have 3 functions, however it can still be achieved using the following code:

```
// output "A" after a random time between 0 & 3 seconds
function outputA(firstCallback, secondCallback) {
  let randomTime = Math.floor(Math.random() * 3000) + 1;

  setTimeout(() => {
    console.log('A');
    firstCallback(secondCallback);
  }, randomTime);
}

// output "B" after a random time between 0 & 3 seconds
function outputB(lastCallback) {
  let randomTime = Math.floor(Math.random() * 3000) + 1;

  setTimeout(() => {
    console.log('B');
    lastCallback();
  }, randomTime);
```

In the above code, we have ensured the correct flow of execution of the three functions by passing both follow up functions to the first function as parameters. The final function is then passed to the second function as a callback, so that it may be executed in the right order.

While this does indeed work to solve the intended problem (getting the output to happen in order: "A", "B" then "C"), we have created some code which is difficult to follow, maintain and scale. For example, what happens when we add an "outputD()" function? We would need to pass it as well to the outputA() function as a parameter, only to get passed down the chain until it is executed in the correct context (for example, after outputC() has completed). As you can imagine, this creates a problem in our code and leaves us asking: "is there a better way?"

Promises

Resolve & Then

Fortunately, JavaScript has the notion of the **"Promise"** that can help us deal with this type of situation. Put simply, a promise object is used for asynchronous computations (like the situation in the example above) and represents a value which may be available now, or in the future, or never. Basically, what this means is that we can place our asynchronous code inside a promise object as a function with specific parameters ("resolve" and "reject"). When our code is complete, we invoke the "resolve" function and if our code encounters an error, we can invoke the "reject" function. We can handle both of these situations later with the .then() method or (in the case of an error that we wish to handle) the .catch() method. To see how this concept is implemented in practice, consider the following addition to the outputA() method from above:

```
// output "A" after a random time between 0 & 3 seconds
function outputA() {
  let randomTime = Math.floor(Math.random() * 3000) + 1;

  return new Promise((resolve, reject) => {
    // place our code inside a "Promise" function
    setTimeout(() => {
      console.log('A');
      resolve(); // call "resolve" because we have completed the
function successfully
    }, randomTime);
  });
}

// call the outputA function and when it is "resolved", output a
confirmation to the console

outputA().then(() => {
  console.log('outputA resolved!');
});
```

Our "outputA()" function still behaves as it did before (outputs "A" to the console after a random period of time). However, our outputA() function now additionally returns a **new Promise** object that contains all of our asynchronous logic and its status. The container function for our logic always uses the two parameters mentioned above, ie: **resolve** and **reject**. By invoking the **resolve** method we are placing the promise into the fulfilled state, meaning that the operation completed successfully and the character "A" was successfully output to the console. We can respond to this situation using the "then" function on the returned promise object to execute some code **after** the asynchronous operation is complete! This gives us a mechanism to react to asynchronous functions that have completed successfully so that we can perform additional tasks.

Adding Data

Now that we have the promise structure in place and are able to **"resolve"** the promise when it has completed it's task and **"then"** execute another function using the returned promise object (as above), we can begin to think about how to pass data from the asynchronous function to the "then" method. Fortunately, it only requires a little tweak to the above the above example to enable this functionality:

```
// output "A" after a random time between 0 & 3 seconds
function outputA() {
  let randomTime = Math.floor(Math.random() * 3000) + 1;

  return new Promise((resolve, reject) => {
    // place our code inside a "Promise" function
    setTimeout(() => {
      console.log('A');
      resolve('outputA resolved!'); // call "resolve" because we
have completed the function successfully
    }, randomTime);
  });
}

// call the outputA function and when it is "resolved", output a
confirmation to the console

outputA().then((data) => {
  console.log(data);
});
```

Notice how we are able to invoke the **resolve()** function with a single parameter that stores some data (in this case a string with the text "outputA resolved!"). This is typically where we would place our freshly returned data

from an asynchronous call to a web service / database, etc. The reason for this is that we will have access to it as the first parameter to the anonymous function declared inside the **.then** method and this is the perfect place to process the data.

Reject & Catch

It is not always safe to assume that our asynchronous calls will complete successfully. What if we're in the middle of a request and our connection is dropped or a database connection fails? To ensure that we handle this type of scenario gracefully, we can invoke the "reject" method instead of the "resolve" method and provide a reason why our asynchronous operation failed. This causes the promise to be in a "rejected" state and the ".catch" function will be invoked, where we can gracefully handle the error. The typical syntax for handling both "then" and "catch" in a promise is as follows:

```
// output "A" after a random time between 0 & 3 seconds
function outputA() {
  let randomTime = Math.floor(Math.random() * 3000) + 1;

  return new Promise((resolve, reject) => {
    // place our code inside a "Promise" function
    setTimeout(() => {
      console.log('-');
      reject('outputA rejected!'); // call "reject" because the
function encountered an error
    }, randomTime);
  });
}

// call the outputA function and when it is "resolved" or
"rejected, output a confirmation to the console
```

> **NOTE**: Calling "resolve()" or "reject()" won't immediately exit the promise and invoke the related ".then()" or ".catch()" callback - it simply puts the promise in a "resolved" or "rejected" state and code immediately following the statement will still run, ie:

```
// ...
reject();
console.log('I will still be executed');
resolve(); // This promise will not be "resolved", since the resolve() call came after reject()
// this also works the other way around.  A promise has been "settled" once reject or resolve has been called
// ...
```

> If we want to immediately exit the function and prevent further execution of the code within the promise, we can invoke the "return" statement, immediately following the "resolve()" or "reject()" call, ie:

```
// ...
reject();
return;
console.log('I will not be executed');
// ...
```

Putting it Together

Now that we know how the promise object and pattern can help us manage our asynchronous code, let's loop back to our original problem - ensuring that "A", "B" and "C" are output in the correct order when invoking the "outputA()", "outputB()" and "outputC()" functions, respectfully.

To make it more interesting, we will alter our code such that each of the

functions **resolve** with the value if the *randomTime* is *odd* and **reject** with an error if *randomTime* is *even*:

```
// output "A" after a random time between 0 & 3 seconds
function outputA() {
  let randomTime = Math.floor(Math.random() * 3000) + 1;

  return new Promise((resolve, reject) => {
    setTimeout(() => {
      randomTime % 2 ? resolve('A') : reject('Error with
outputA()');
    }, randomTime);
  });
}

// output "B" after a random time between 0 & 3 seconds
function outputB() {
  let randomTime = Math.floor(Math.random() * 3000) + 1;

  return new Promise((resolve, reject) => {
    setTimeout(() => {
      randomTime % 2 ? resolve('B') : reject('Error with
outputB()');
    }, randomTime);
  });
}

// output "C" after a random time between 0 & 3 seconds
function outputC() {
  let randomTime = Math.floor(Math.random() * 3000) + 1;

  return new Promise((resolve, reject) => {
    setTimeout(() => {
      randomTime % 2 ? resolve('C') : reject('Error with
outputC()');
    }, randomTime);
```

If we wish to use the promises correctly to output the values in order **and** correctly handles errors, our code looks like the following (this is known as promise "chaining"):

```
outputA()
  .then((data) => {
    console.log(data); // output the result of "outputA()" to the
console
    return outputB();
  })
  .then((data) => {
    console.log(data); // output the result of "outputB()" to the
console
    return outputC();
  })
  .then((data) => {
    console.log(data); // output the result of "outputC()" to the
console
  })
  .catch((err) => {
    console.log(err); // output the error to the console
  });
```

Success! We always have "A", followed by "B" and "C" in the console and the errors are correctly handled when they occur (preventing the subsequent promises from executing). We have the benefit of not having to *alter* the functions themselves at all if follow-up logic is necessary. Each function simply does its job, then reports back with the data ("resolves") if it was successful or sends the error ("rejects") it failed. This is a much more maintainable, scalable and cleaner approach to working with asynchronous code. This is why you will find that most modules mentioned in these notes are "promise-based", ie: if their logic is asynchronous, functions provided by the module will return **Promise** objects.

While functions that return promises are indeed the preferred way to work with asynchronous operations in JavaScript, as you can see from the above code, *working* with promises can sometimes be difficult. If we wish to chain promises (in the case above) We must ensure that for every "then()" callback function returns the correct follow up function and it can be difficult to visually walk through the code.

Async & Await

To help us work with promises more easily in JavaScript, ECMAScript 2016 (ES7) released async & await as an alternative to using "then()" and "catch()"

Putting it Together (again)

Knowing that there is an alternative to "then()" and "catch()", let's see how we can re-write the section of "Putting it Together" that *makes use of* the promises (we will *not* alter the functions themselves) using "async" and "await". To achieve this, we must place our logic inside a function, ie "showOutput()" - the reason for this will be described below:

```
async function showOutput() {
  try {
    let A = await outputA();
    console.log(A); // output the result of "outputA()" to the
console

    let B = await outputB();
    console.log(B); // output the result of "outputB()" to the
console

    let C = await outputC();
    console.log(C); // output the result of "outputC()" to the
```

This is *much* cleaner and easier to read. By using the "await" operator, we're essentially saying "wait for this function's returned promise to resolve". Additionally, you can see that we actually get the resolved value from the promise!

Using Await

> "await" pauses the execution of its surrounding async function until the promise is settled (that is, fulfilled or rejected). When execution resumes, the value of the await expression becomes that of the fulfilled promise.
>
> If the promise is rejected, the await expression throws the rejected value.
>
> https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await

Notice how the documentation mentions the "surrounding async function". This is because to actually **use** the "await" operator, it **must** be placed within a function marked as "async". If we fail to do this and try to use await outside of an async function, we will get an error:

```
SyntaxError: await is only valid in async functions and the top
level bodies of modules
```

You will also notice how the documentation mentions that if the promise is rejected, the await expression "throws the rejected value". This is why we must place our "await" logic within a "try" / "catch" block. If we fail to do so and one of the promise-based functions is actually rejected, we will get the following error (**NOTE:** this error also occurs if a ".catch()" function is missing when using then() & catch()):

```
UnhandledPromiseRejection: This error originated either by
throwing inside of an async function without a catch block, or by
rejecting a promise which was not handled with .catch().
```

> **NOTE**: When using "async" to identify a function, you are implicitly **returning a Promise**. This is because async functions *cannot* exist within the normal flow of execution (since they contain asynchronous code). If you do return a value from an "async" function, it will be the "resolved" value of the returned promise:

```
async function adder(num1, num2) {
  return num1 + num2;
}

adder(1, 2).then((result) => console.log(result)); //3
```

Example Code

You may download the sample code for this topic here:

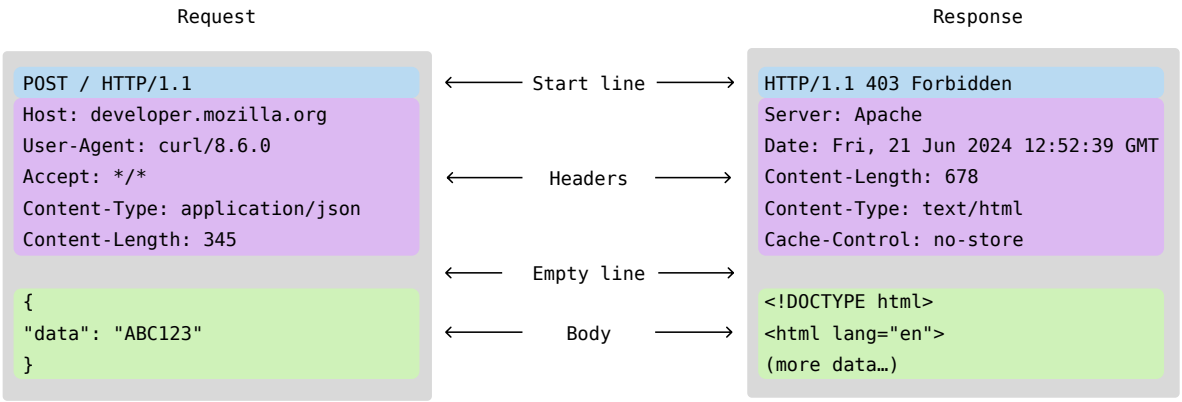
> Handling-Asynchronous-Code

HTTP Protocol Overview

The HTTP Protocol itself is an Application layer protocol – that is, it essentially sits “on top” of an underlying network-level protocol such as the Transmission Control Protocol (TCP). HTTP is human-readable and extensible, which makes the protocol extremely easy to extend and to experiment with. New functionality can be introduced simply by establishing an agreement between a client and a server and specifying new “headers” – these will enable the client and server to pass additional information along with the request or the response. The payload content (ie: raw HTML) is sent in the “message body”.

Both HTTP requests and responses share a similar structure and are composed of:

- A start-line that describes the requests to be performed, or its status that is a success or a failure. This start-line is always a single line.
- An optional set of HTTP headers specifying the request, or describing the body included in the message.
- A blank line indicating that all meta-information for the request has been sent.
- An optional body that contains data associated with the request (like the content of an HTML form), or the document associated with a response. The presence of the body and its size is defined by the start-line and the HTTP headers.



(https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages)

HTTP Requests

Start line

HTTP requests are messages sent by the client to initiate an action on the server. Their start-line contains of three elements:

1. An HTTP method that describes the action to be performed:

Method	Description
GET	The GET method is used to retrieve information from a specified URI (Universal Resource Identifier) and is assumed to be a safe, repeatable operation by browsers, caches and other HTTP aware components. This means that the operation must have no side effects and GET requests can be re-issued without worrying about the consequences.

Method	Description
POST	The POST method requests that the target resource process the representation enclosed in the request according to the resource’s own specific semantics. For example, POST is used for the following functions (among others): - Providing a block of data, such as the fields entered into an HTML form, to a data-handling process; - Posting a message to a bulletin board, newsgroup, mailing list, blog, or similar group of articles; - Creating a new resource that has yet to be identified by the origin server; - Appending data to a resource’s existing representation(s).
PUT	The PUT method is used to request that server store the content included in message body at a location specified by the given URL. For example, this might be a file that will be created or replaced.
HEAD	The HEAD method is identical to GET except that the server MUST NOT send a message body in the response (i.e., the response terminates at the end of the header section). This method can be used for obtaining metadata about the selected representation without transferring the representation data.
DELETE	The DELETE method requests that the origin server remove the association between the target resource and its current functionality. In effect, this method is similar to the rm command in UNIX: it expresses a deletion operation on the URI mapping of the origin server.

Method	Description
CONNECT	The CONNECT method requests that the recipient establish a tunnel to the destination origin server identified by the request-target and, if successful, thereafter restrict its behavior to blind forwarding of packets, in both directions, until the tunnel is closed. Tunnels are commonly used to create an end-to-end virtual connection, through one or more proxies, which can then be secured using TLS (Transport Layer Security).
OPTIONS	The OPTIONS method requests information about the communication options available for the target resource. This method allows a client to determine the options and/or requirements associated with a resource, or the capabilities of a server, without implying a resource action.
TRACE	The TRACE method requests a remote, application-level loop-back of the request message. This is typically used to echo the contents of an HTTP Request back to the requester which can be used for debugging purposes during development.

2. The request target (this can vary between the different HTTP methods) – for example, this can be:
    - An absolute path, optionally followed by a '?' and a query string. This is the most common form, called origin form, and is used with GET, POST, HEAD, and OPTIONS methods, for example:
        - `POST / HTTP 1.1`
        - `GET /background.png HTTP/1.0`
        - `HEAD /test.html?query=alibaba HTTP/1.1`

        - `OPTIONS /anypage.html HTTP/1.0`
    - A complete URL, the absolute form, mostly used with GET when connected to a proxy, for example:
        - `GET http://developer.mozilla.org/en-US/docs/Web/HTTP/Messages HTTP/1.1`
    - The authority component of an URL, that is the domain name and optionally the port (prefixed by a ‘:’), called the authority form. It is only used with CONNECT when setting up an HTTP tunnel, for example:
        - `CONNECT developer.mozilla.org:80 HTTP/1.1`
    - The asterisk form, a simple asterisk (‘*’) used with OPTIONS and representing the server as a whole, for example:
        - `OPTIONS * HTTP/1.1`
3. The HTTP version, that defines the structure of the rest of the message, and acts as an indicator of the version to use for the response.

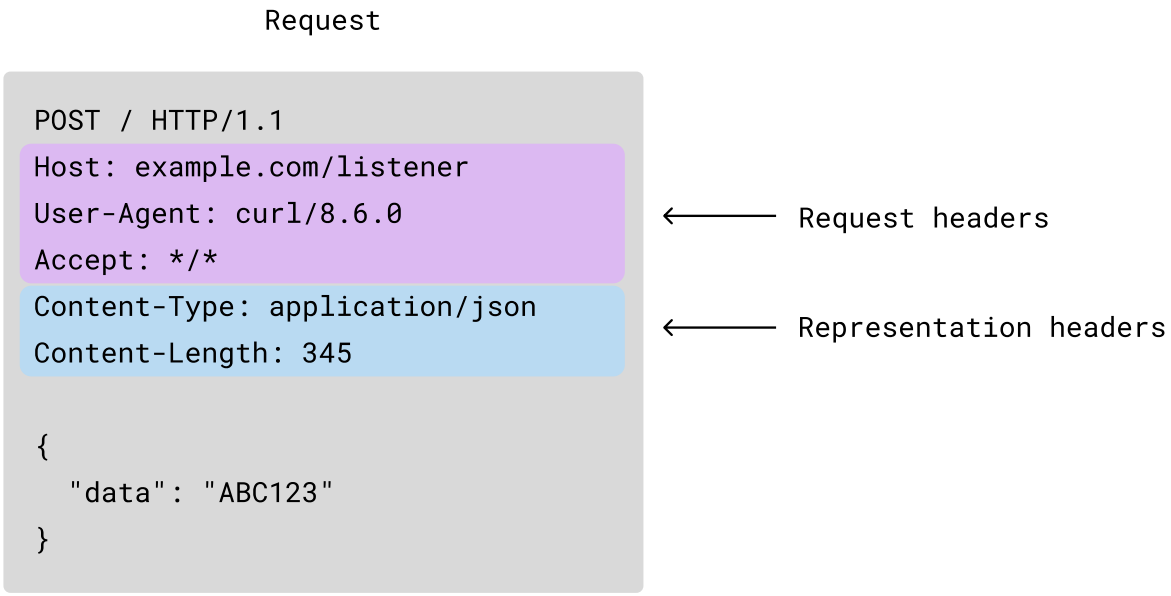
Headers

HTTP headers in a request follow the basic structure of any HTTP header: a case-insensitive string followed by a colon (‘:’) and a value whose structure depends upon the header. The whole header, including the value, consists of one single line, that can be quite long.

There are numerous request headers available. In a request, the headers can be divided into two groups:

- **Request headers:** Provide additional context to a request or add extra logic to how it should be treated by a server (e.g., conditional requests).

- **Representation headers:** Sent in a request if the message has a body, and they describe the original form of the message data and any encoding applied. This allows the recipient to understand how to reconstruct the resource as it was before it was transmitted over the network.



(https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages)

Body

The last part of a request is its body. Not all requests have one: for example, requests fetching resources (like GET or HEAD) usually don’t need any. Similarly, DELETE or OPTIONS also do not require a body.

Other requests send data in the body to the server in order to update it: this is often the case of POST requests (that can have HTML form data).

HTTP Responses

Status line

The start line of an HTTP response, called the status line, contains the following information:

1. The protocol version, usually **HTTP/1.1**
2. A status code beginning with 1, 2, 3, 4 or 5 that provides information such as the success or failure of the request:

Range	Description
1xx	**Informational:** Request received, continuing process. For example, Microsoft IIS (Internet Information Services) initially replies with 100 (Continue) when it receives a POST request and then with 200 (OK) once it has been processed.
2xx	**Success:** The action was successfully received, understood, and accepted. For example, the 200 (Ok) status code indicates that the request has succeeded. The meaning of “success” varies depending on the HTTP method, for example: **GET:** The resource has been fetched and is transmitted in the message body, **HEAD:** The entity headers are in the message body, **POST:** The resource describing the result of the action is transmitted in the message body, and **TRACE:** The message body contains the request message as received by the server
3xx	**Redirection:** Further action must be taken in order to complete

Range	Description
	the request. For example, the 302 (Found) status code indicates that the requested resource has been temporarily moved and the browser should issue a request to the URL supplied in the Location response header.
4xx	**Client Error:** The request contains bad syntax or cannot be fulfilled. For example, the famous 404 (Not Found) status code indicates that the server can not find requested resource, or is not willing to disclose that one exists.
5xx	**Server Error:** The server failed to fulfill an apparently valid request. For example, the 500 (Internal Server Error) status code indicates that the server encountered an unexpected error / condition that prevented it from fulfilling the request.

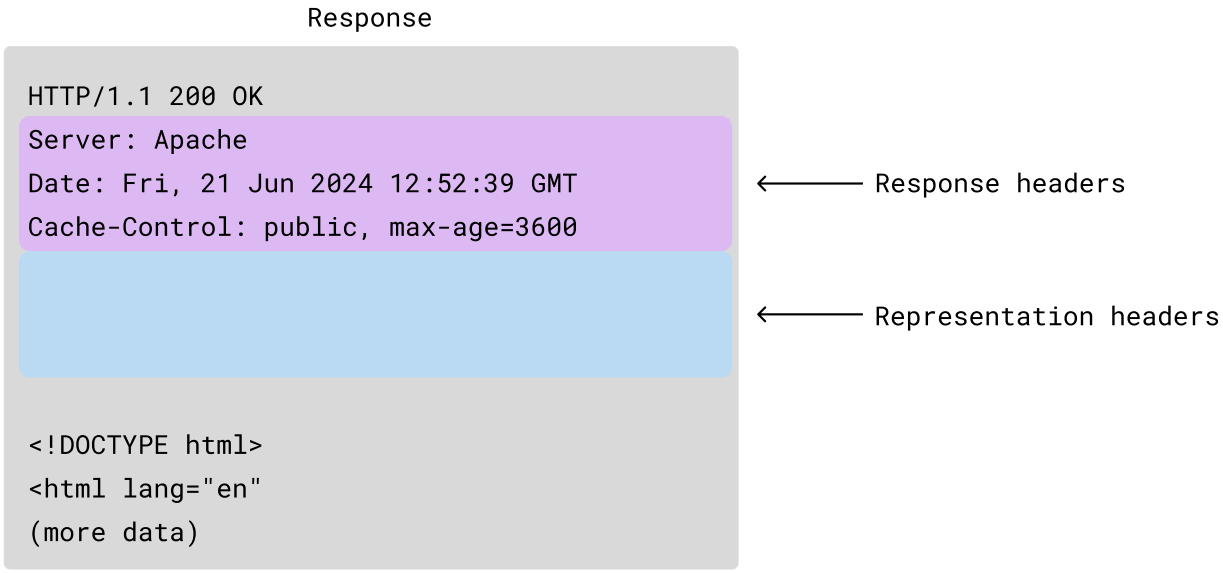
3. A status text, purely informational, that is a textual short description of the status code. This helps HTTP messages be more human-readable, for example:
    - `HTTP/1.1 404 Not Found`

Headers

The HTTP header format for responses follow the same basic structure (a case-insensitive string followed by a colon (‘:’) and a value whose structure depends upon the type of the header. The whole header, including the value, stands in one single line)

There are numerous response headers available. In a response, the headers can be divided into two groups:

- **Response headers:** Give additional context about the message or add extra logic to how the client should make subsequent requests. For example, headers like Server include information about the server software, while Date includes when the response was generated. There is also information about the resource being returned, such as its content type (Content-Type), or how it should be cached (Cache-Control).
- **Representation headers:** Describe the form of the message data and any encoding applied (if the message has a body). For example, the same resource might be formatted in a particular media type such as XML or JSON, localized to a particular written language or geographical region, and/or compressed or otherwise encoded for transmission. This allows a recipient to understand how to reconstruct the resource as it was before it was transmitted over the network.



(https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages)

Body

The last part of a response is the body. This is typically a single file of known length (defined by the two headers: “Content-Type” and “Content-Length”) or a

single file of unknown length (encoded in chunks with the “Transfer-Encoding” header set to “chunked”. However, not all responses have a body, for example: responses with status code like 201 (Created) or 204 (No Content).

Modules & Node Package Manager

Modules in JavaScript serve as fundamental building blocks for organizing and structuring code, enabling code reuse and modularization. They contain self-contained pieces of code that perform specific functionalities, allowing developers to break down complex applications into manageable components. This modular approach not only improves code readability, maintainability, and scalability but also simplifies testing as individual modules can be tested independently.

In Node.js, modules can be either built-in, local (custom modules created within the application), or third-party (external modules downloaded from npm).

Let's begin with discussing some of the common "Built-In" modules and how we can use them.

Built-In Modules / 'require()'

While referencing the official Node.js Documentation, you may have noticed that some of the examples include a mandatory 'require()' statement. For example, if we try to execute this *simplified* 'EventEmitter' sample:

```
const myEmitter = new EventEmitter();

myEmitter.on('event', function () {
  console.log('an event occurred!');
});
```

we run into an error: `ReferenceError: EventEmitter is not defined`. As you will have guessed, this is because our running script does not know about the "EventEmitter" class, as it is not global. To remedy this, we can include the required class by "requiring" it, with the following syntax:

```
const EventEmitter = require('events');

const myEmitter = new EventEmitter();

myEmitter.on('event', function () {
  console.log('an event occurred!');
});

myEmitter.emit('event');
```

By using the global 'require' function, we have loaded a code "module" which contains code and logic that we can use in our own solutions. We will discuss modules in detail below, however for now we should be aware of the following "Built-In" modules:

fs

The 'fs' module is used to work directly with the file system (ie: read / write files, list the contents of a directory, etc). For example, if we had a CSV file with names, (ie: *names.csv*):

```
Jacob,Alexandra,Jessie,Ranya,Felix
```

We could read the contents of the file and convert the list into an array:

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

Similarly, if we had a directory of images, ie: "img", we could list the files using:

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

fs.readdir('img', function (err, filesArray) {
  if (err) console.log(err);
  else {
    console.log(filesArray);
  }
});
```

path

The 'path' module provides utilities for working with file and directory paths. This will be useful when working with reading template files or writing uploaded files. For example, it can easily be used to safely concatenate two directories / paths together:

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

console.log('Absolute path to about.html');

console.log(path.join(__dirname, '/about.html')); // with leading
slash
console.log(path.join(__dirname, '//about.html')); // with
multiple leading slashes
console.log(path.join(__dirname, 'about.html')); // without
leading slash
console.log(path.join(__dirname, '\about.html')); // with
incorrect leading slash
```

Writing Modules

We can also create our own modules that work the same way, by making use of a global “module” object – which isn’t truly “global” in the same sense as “console”, but instead global to each of your modules, which are located in separate .js files. For example, consider the two following files (modEx1.js: the main file that Node will execute, and message.js: the file containing the module):

file ./modEx1.js

```
let message = require('./modules/message');

message.writeMessage('Hello World!');

message.readMessage();
```

file: ./modules/message.js

```
// NOTE: Node.js wraps the contents of this file in a function:
// (function (exports, require, module, __filename, __dirname) {
... });
// so that we have access to the working file/directory names as
well
// as creating an isolated scope for the module, so that our
// variables are not global.

let localFunction = () => {
  // a function local to this module
};
```

Executing the code in modEx1.js (ie: **node modEx1.js**) should output:

“Hello World” from …

where … is the absolute location of the message.js file in your system, for example: **/Users/pat/Desktop/Seneca/modules/message.js**

Notice how our “message” module uses the exports property of the “module” object to store functions and data that we want to be accessible in the object returned from the require(“./modules/message”); function call from modEx1.js. Generally speaking, if you want to add anything to the object returned by “require” for your module, it’s added to the module.exports object from within your module. In this case, we only added two functions (readMessage() and writeMessage()).

Using this methodology, we can safely create reusable code in an isolated way that can easily be added (plugged in) to another .js file.

NPM – Node Package Manager

The Node Package Manager is a core piece of the module based Node ecosystem. The package manager allows us to install and manage 3rd party modules, available from https://www.npmjs.com within our own applications.

From the npm documentation:

> npm is the world's largest software registry. Open source developers from every continent use npm to share and borrow packages, and many organizations use npm to manage private development as well.
>
> npm consists of three distinct components:
>
> - the website

> - the Command Line Interface (CLI)
> - the registry
>
> Use the website to discover packages, set up profiles, and manage other aspects of your npm experience. For example, you can set up organizations to manage access to public or private packages.
>
> The CLI runs from a terminal, and is how most developers interact with npm.
>
> The registry is a large public database of JavaScript software and the meta-information surrounding it.

The CLI is installed by default when you install Node. From the command line you can run ‘npm’ with various commands to download and remove packages for use with your Node applications. When you have installed a package from npm you use it in the same way as using your own modules like above, with the require() function.

All npm packages that you install locally for your application will be installed in a node_modules folder in your project folder.

While there are over 60 "npm" commands available, the ones that we will most commonly use in this course are as follows:

Command	Description
npm install [Module Name]	install is used to install a package from the npm repository so that you can use it with your application. ie: `let express = require("express");`
npm	uninstall does exactly what you would think, it uninstalls a

Command	Description
uninstall [module name]	module from the node_modules folder and your application will no longer be able to require() it.
npm init	create a new package.json file for a fresh application. More on this part later.
npm prune	The prune command will look through your package.json file and remove any npm modules that are installed that are not required for your project. More on this part later.
npm list	Show a list of all packages installed for use by this application.

Globally installing packages

Every so often, you will want to install a package globally. Installing a package globally means you will install it like an application on your computer which you can run from the command line, not use it in your application code. For example, some npm packages are tools that are used as part of your development process on your application:

One example is the migrate package which allows you to write migration scripts for your application that can migrate your data in your database and keep track of which files have been run.

Another example is grunt-cli so that you can run grunt commands from the command line to do things like setup tasks for running unit tests or checking for formatting errors in code before pushing up new code to a repository.

A third example is bower. Bower is a package manager similar to npm but typically used for client side package management. To install a package globally you just add the -g switch to your npm install command. For example:

```
npm install bower -g
```

Globally installed packages do not get installed in your node_modules folder and instead are installed in a folder in your user directory. The folder uses for global packages varies for Windows, Mac, and Linux. See the documentation if you need to find globally installed packages on your machine.

package.json explained

The Node Package Manager is great. It provides an easy way to download reusable packages or publish your own for other developers to use. However, there are a few problems with sharing modules and using other modules, once you want to work on an application with someone else. For example:

- How are you going to make sure everyone working on your project has all the packages the application requires?
- How are you going to make sure everyone has the **same version** of all those packages?
- Finally, how are you going to handle updating a package and making sure everyone else on your project updates as well?

This is where the package.json file comes in.

The package.json file is a listing of all the packages your application requires and also which versions are required. It provides a simple way for newcomers to your project to get started easily and stay up to date when packages get

updated.

The npm documentation for the package.json file has all the information you will need as you begin building applications in node.js

Let’s look at how we can generate a package.json file using the npm **init** command from within your project's folder (in this case: "/Users/pat/Desktop/Seneca/"):

```
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible
defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (seneca)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to /Users/pat/Desktop/Seneca/package.json:

{
  "name": "seneca",
  "version": "1.0.0",
```

If you try running this command yourself, you will see that the process is *interactive*, ie: you will be prompted to enter everything from the "package name" to the "license". Any values that you see in brackets "()" are *default* values and will be accepted if you press "Enter".

Once this process is done, you will see that you have a new file created in your project called **package.json**. In the above case, it will look like this:

```
{
  "name": "seneca",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Once generated, you can edit it if you decide to change the name or version (for example). Once you decide to add packages to your app you can simply install the package with **npm install**. This will save the package and version into the package.json file for you so that when others want to work on your app, they will have the package.json file and can use **npm install** to install all the required dependencies with the right version. Think of package.json as a checklist for your application for all of its dependencies.

Simple Web Server using Express.js

A major focus of these notes going forward will be creating modern web applications using Node.js. While there are many ways of accomplishing this task, including using the built-in 'http' module, we will be using the extremely popular "Express" web framework, available on NPM.

Project Structure

To get started working with Node.js and Express, we should create a new folder for our application (ie: "MyServer", as used in the below example). Once this is completed, open it in Visual Studio Code and create the following directory structure by adding "public" and "views" folders as well as a "server.js" file:

```
/MyServer
  ↳ /public
  ↳ /views
  ↳ server.js
```

Next, we must open the integrated terminal and create the all-important "package.json" file at the root of our "MyServer" folder, using the command **"npm init"**.

> **NOTE:** You will be using all of the *default* options when creating your package.json file

Once this is complete, you should have a new package.json file in your MyServer folder that looks like the following:

```
{
  "name": "myserver",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node server.js"
  },
  "author": "",
  "license": "ISC"
}
```

Express.js

Express.js is described as:

> "a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications."

Essentially, it is a Node module that takes a lot of the leg work out of creating a framework to build a website. It is extremely popular in the node.js community with a multitude of developers using it to build websites. It is a proven way to build flexible web applications quickly and easily.

To use it in our project we need to use "npm" to install it. From the integrated terminal in Visual Studio code, enter the command:

```
npm i express
```

(where "i" is shorthand for the "install" command).

Once this is complete, you should see that your "package.json" file has a new entry that looks like the following (**NOTE:** Your version may differ from the below):

```
"dependencies": {
    "express": "^4.18.2"
  }
```

You will also notice that a 2nd file was created called "package-lock.json":

> The purpose of package-lock.json is to ensure that the same dependencies are installed consistently across different environments, such as development and production environments. It also helps to prevent issues with installing different package versions, which can lead to conflicts and errors.
>
> https://www.atatus.com/blog/package-json-vs-package-lock-json/#package-lock-json

Finally, we also now have the aforementioned "node_modules" folder, which not only contains an "express" folder, but also folders for all of the other modules that "express" depends upon, such as "cookie", "encodeurl", "http-errors", etc.

To begin using Express.js, we must first "require" it in our server.js file and execute the code to start our server. As a starting point, you may use the following *boilerplate* code:

File: server.js

```
const express = require('express'); // "require" the Express module
```

The above code will be used in nearly every server written using "Express" in these notes. As mentioned above, it "requires" the Express module, which is then invoked as a function to get an "app" object, which is used to start our server on a given HTTP Port. The reason that the HTTP_PORT constant is defined as `process.env.PORT || 8080` is because when we move our server online, it will be assigned a different port, using a "PORT" environment variable.

If we now want to start our server, we can simply execute the "server.js" file using node:

```
node --watch server.js
```

> **NOTE:** the "--watch" flag will cause Node to run in "watch" mode, which will restart the process when a change is detected

If you open a browser to: `http://localhost:8080`, you should see the following message:

```
Cannot GET /
```

Congratulations! Your web server is up and running! Unfortunately, we don't have any "routes" (ie: paths to pages / resources) defined yet, so the Express framework automatically generated a **404** error for the path that we tried to access (ie: GET /)

> **NOTE:** To stop the server from running, you may use the `Ctrl+C` command from the integrated terminal in Visual Studio Code

Simple 'GET' Routes

As you have seen from running our server, not much is happening. Even if we try to navigate around to other paths such as "http://localhost:8080/about" (thereby making a "GET" request to the "/about" path (route)), we will keep getting the same 404 error: "Cannot GET". This is because we have not defined any "GET" routes within our server.

To fix this, we must write code in our server.js file to correctly *respond* to these types of requests. This can be accomplished using the "app" object, that was used to start our server. If we wish to respond to a "GET" request, we must invoke a "GET" function and provide the target path as well as a "callback" function to handle the request. For example, if we wish to respond to a "GET" request on the "/" route, we would write the following code *before* the call to app.listen();

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

Here, we have specified a callback function to be executed when our server encounters a "GET" request for the "/" route. It will be invoked with the following parameters:

- "req": The "request" object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on.
- "res": "The "response" object represents the HTTP response that an Express app sends when it gets an HTTP request

In the above case, we use the "res" object's "send" method to send a response back to the client.

If we wish to have a second route, all we have to do is add another call to "app.get()" with the new path. This is how we will define any path "route" that we wish our server to respond to, when it encounters a "GET" request from a web client (ie: web browser):

```
app.get('/about', (req, res) => {
  res.send('About the Company');
});
```

Now, we should be able to navigate to both: `http://localhost:8080` and `http://localhost:8080/about` and see the text sent by our server.

Returning .html Files

Returning plain text is fine to test if our routes are configured properly, however if we want to start making web applications, we should be returning valid HTML documents. To get started, we will create two simple .html files within the "views" folder:

File: /MyServer/views/home.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-
scale=1.0" />
    <title>Home</title>
  </head>
  <body>
    <h1>Welcome Home</h1>
    <p>...</p>
```

File: /MyServer/views/about.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-
scale=1.0" />
    <title>About</title>
  </head>
  <body>
    <h1>About the Company</h1>
    <p>...</p>
    <p><a href="/">Back Home</a></p>
  </body>
</html>
```

Next, we must update our route definitions to return these documents *instead* of the simple messages: "Hello World!" and "About the Company". To achieve this, we will be using the "sendFile()" method of the "res" object, *instead* of "send()".

For "sendFile()" to function correctly, we must provide an **absolute** path to the file we wish to send as a parameter to the function. As you know, we cannot hard-code this path into our server.js, as this path will differ depending on which machine is executing the code - for example: the service the app is deployed on, vs. your local computer.

This is where knowledge of the built-in "path" module and the __dirname global come into play.

At the top of your server file, we will *require* "path";

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

Next, we can update our routes to use "sendFile()" as follows:

```
res.sendFile(path.join(__dirname, '/views/someFile.html'));
```

where **"someFile.html"** would be any file that you wish to send back to the client, from your "views" folder, ie: "home.html" or "about.html". We use `path.join()` to safely join the "__dirname" path with the local path to the file. Together, this results in an absolute path that is *not* tied to a specific machine.

CSS & Images

Now that we know how to send complete HTML files back to the client, the next step is including "static" resources, ie: images, CSS, etc. So far, if we wish to respond to a request from a client we must have an explicit "route" configured. For example, the "/about" route only works because we have defined the corresponding `app.get("/about", ...)` function call. What happens when a request for a static resources is requested? Do we have to have a specific root configured for every resource? Thankfully, the answer is *no*.

Using Express, we can identify a specific folder as "static" and any valid requests for resources contained within that folder are automatically sent back to the client with a 200 status code.

Using our existing project structure, we can use the "public" folder as our static folder and place any static resources in there. For example, if we want a custom CSS file, we could place it in:

```
/MyServer
```

We could then link to it in our HTML documents the code:

```
<link rel="stylesheet" href="/css/site.css" />
```

> **NOTE:** The same pattern would work for images as well, ie:
>
> ```
> /MyServer
>   ↳ /public
>     ↳ /img
>       ↳ banner.jpg
> ```
>
> ```
> <img src="/img/banner.jpg" alt="site banner" />
> ```

Notice how we do not include "/public" in the `href` (or `src`) properties. This is because we will mark "/public" as the official "static" folder and all requests must made to resources *within* the folder. To accomplish this in our server.js file, we can add the following code *above* the other `app.get()` function calls:

```
app.use(express.static('public'));
```

Here, we have used "express.static()" - a built-in middleware function (explained later in these notes) to mark the "public" directory as static. With this code in place, whenever a request is sent to our server, Express will first check to see if the requested resource exists in the "public" folder, before checking our other routes.

Public Hosting (Vercel)

As a final exercise, review the documentation on "Getting Started with Vercel" and see if you can get the server running online!

Example Code

You may download the sample code for this topic here:

> Web-Server-Introduction

Application, Request & Response Objects

Express.js makes it very straightforward to get a web server running on a given port and responding to simple "get" requests (ie: `GET / HTTP/1.1`):

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

const HTTP_PORT = process.env.PORT || 8080;

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

app.listen(HTTP_PORT, () => console.log(`server listening on:
${HTTP_PORT}`));
```

From the above code, it is clear that there are three important objects that are used to configure the server: `app`, `req` and `res`. Let's discuss these objects in detail and how we can use them to handle more complicated scenarios, such as route / query parameters, cookies and custom errors.

The Application object

The "app" object in the example above represents the express main application object. It contains several methods for tasks, such as processing route requests, setting up middleware, and managing html views or view engines.

In the above example, we set a route on the host to handle HTTP GET requests to “/”. This means any "GET" requests to localhost:8080/ will be sent to this function. A typical route handler in express (like the one above) is created by invoking a function on the app object using the HTTP method (verb) that matches the type of request and passing it two parameters: a string representing the route, and a callback function to invoke when the route is matched. In this case, we wish to handle GET requests for the default route "/" (typically requests from the browser to load the page initially).

Here are some of the commonly used application properties and methods that we will use throughout these notes.

app.all()

This method is used to register a single callback for a route that matches *any* HTTP Method IE: GET, PUT, POST, DELETE, etc.

```
app.all('/http-testing', (req, res) => {
  res.send('test complete');
});
```

HTTP Verb Methods

We can also respond to a request a callback for a route using a *single* HTTP Method (ie: `app.get()` from our Simple Web Server using Express.js example):

```
app.get('/get-test', (req, res) => {
  res.send('GET Test Complete');
});

app.put('/put-test', (req, res) => {
```

app.locals

The "locals" property allows you to attach local variables to the application, which persist throughout the life of the app. You can access local variables in templates rendered within the application (discussed in "Template Engines").

```
app.locals.title = 'My App';
```

app.listen()

As we have seen, this function is used to start the HTTP server listening for connections on a specific port, ie:

```
const HTTP_PORT = process.env.PORT || 8080;

// (route handlers / middleware) ...

app.listen(HTTP_PORT, () => {
  console.log('server listening on: ' + HTTP_PORT);
});
```

app.set()

The "set" method assigns a value to a specific "setting". According to the documentation, you may store any value that you want in your own custom "setting", however certain settings can be used to configure the behavior of the server. For example, we will be setting the value of the "view engine" setting when configuring our template engine.

app.use()

The ***use*** method is used to add middleware to your application. Middleware consists of functions (typically placed before the route handlers) that automatically execute either when a specified path is matched or globally before every request. This is very useful when you want to do something with every request like add properties to the request object or check if a user is logged in.

This is discussed further in the next section: "Middleware"

The Request object

The "req" object represents the object that contains all the information and metadata for the request sent *to* the server. When you see examples of the request object in use, it will typically be referred to as ‘req’ (short for request object).

Some of the commonly used request properties and methods used throughout these notes are:

req.body

The req.body property contains the data submitted as part of request. It requires that you use a "body parsing" middleware (discussed in: "Middleware") which will attach data (properties) to req.body. If you post data in your request, this is how you access that data.

```
app.post('/urlencoded-test', (req, res) => {
```

req.cookies

If we wish to read the value specific "cookie" value, ie:

> "a small piece of data that a server sends to a user's web browser. The browser may store the cookie and send it back to the same server with later requests."
>
> https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies

we can reference it using the corresponding property on the "req.cookies" object:

```
// Cookie: name=tj
console.log(req.cookies.name); // "tj"
```

However, like "req.body" above, we must use a ("cookie parsing") middleware function to populate "req.cookies" with data from the cookie

req.params

The "params" property is used when we wish to read the values of "Route Parameters" defined in our route handlers:

> "Route parameters are named URL segments used to capture values at specific positions in the URL. The named segments are prefixed with a colon and then the name (E.g., /:your_parameter_name/)."
>
> https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes#route_parameters

For example, if we wish to match all GET requests for the route

"/employee/**employeeNum**", where **employeeNum** can be *any* value, ie: "123", "abc456", etc, we can use the following code:

```
app.get('/employee/:employeeNum', (req, res) => {
  res.send(`Employee Number: ${req.params.employeeNum}`);
});
```

req.query

The "query" property is needed when we wish to read the values of the "query string" in the url:

> A query string is a part of a uniform resource locator (URL) that assigns values to specified parameters. A query string commonly includes fields added to a base URL by a Web browser or other client application, for example as part of an HTML document, choosing the appearance of a page, or jumping to positions in multimedia content
>
> A typical URL containing a query string is as follows:
> `https://example.com/over/there?name=ferret`
>
> https://en.wikipedia.org/wiki/Query_string

For example, if we wanted to match a GET request for the route "/products" that *also* supports the optional query string value "onSale", ie: "/products?onSale=true", we could use the code:

```
app.get('/products', (req, res) => {
  let result = 'all Products';

  // NOTE: query parameter values are always strings
```

When designing route handlers that can accept query string values, we do not include them in the "route" (ie: "/products"). Additionally, since the route will match *without* the "onSale" query sting value, it is important to return a value if it's missing (ie: "all Products" or an error if the query parameter *must* be present)

> **NOTE:** Multiple query parameters may also be used, and are separated by an ampersand, "&": `https://example.com/path/to/page?name=ferret&color=purple`

req.get()

req.get() is necessary for checking the values of specific HTTP headers sent with the request. For example:

```
app.get('/hello', (req, res) => {
  res.send(`Hello ${req.get('user-agent')}`);
});
```

Here, when a user requests the "/hello" route, they should see the text "Hello" followed by the content of the "user-agent" header sent with the request.

The Response object

The "res" object represents the object that contains all the information and metadata for a response sent *from* the server. When you see examples of the response object in use it will typically be referred to as ‘res’ (short for response object). The data you send back from the server can be one of several different formats - the most common of which are HTML, JSON, CSS, JS and plain files (.pdf, .txt, .jpg, .png, etc).

Some of the commonly used response properties and methods used throughout these notes are:

res.cookie()

This allows you to send a cookie with the response, specified using a name = value key pair. You can set the value to a string / object using JSON notation and it will be included in the "Set-Cookie" header of the response. For example:

```
app.get('/cookie-test', (req, res) => {
  res.cookie('message', 'Hello World!');
  res.send('Cookie Sent!');
});
```

res.set()

res.set() enables you to set the values of specific / custom HTTP headers sent with the request. For example:

```
app.get('/custom', (req, res) => {
  res.set('Custom-Header', 'MyValue');
  res.send(`Custom-Header Sent`);
});
```

res.end()

res.end() is used you want to end a response immediately and send nothing back. For example, we may wish to send a "204 - No Content" status code, which indicates that "a request has succeeded, but that the client doesn't need to navigate away from its current page". For example:

```
app.put('/update', (req, res) => {
  // ... (update logic)
  res.status(204).end();
});
```

res.redirect()

The res.redirect() method is used to perform a redirect to another page on your site, go back to the previous page, or redirect to another domain. For example:

```
app.get('/to-google', (req, res) => {
  res.redirect('https://www.google.ca/');
});
```

res.send()

This is the primary response method to send a response to the client. You can send a String, Object, Array, or even a Buffer object back to the client. The send() method will automatically set the Content-Type header for you based on the type of data sent. For example:

```
app.get('/json-test', (req, res) => {
  res.send({ message: 'Hello World!' }); // Content-Type:
application/json; charset=utf-8
});

app.get('/plain-text-test', (req, res) => {
  res.send('Hello World!'); // Content-Type: text/html;
charset=utf-8
});
```

> **NOTE:** When sending a JavaScript object back (as in the example above), the "send()" method will internally convert it to a JSON-formatted string

res.sendFile()

As we have seen, this function is used when we wish to send a file (typically .html) back to the client. We use path.join() to safely join __dirname with the path of the file to be sent. This function also correctly sets the Content-Type response HTTP header based on the file extension. For example:

```
app.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, '/views/home.html'));
});
```

res.status()

res.status() is used to set a specific status code for the response (as seen above in the res.end() example). This will be useful when handling client / server errors and setting 4xx / 5xx series error codes. More detail is discussed in the following "Middleware" section.

Middleware

Middleware in Express refers to functions that can execute in the ‘middle’ of a request/response cycle typically before a matching route handler function is executed.

> Middleware functions are functions that have access to the request object (req), the response object (res), and the next() function in the application’s request-response cycle. The next() function is a function in the Express router which, when invoked, executes the middleware succeeding the current middleware.
>
> http://expressjs.com/en/guide/writing-middleware.html

By implementing middleware, we can perform tasks such as:

- Directly modify the "req" (request) or "res" (response) objects *before* processing the route (ie: `app.get('/', (req, res) => { ... });`)
- Redirect the user or respond to requests before other routes are processed
- Block clients from accessing specific routes
- Log requests / handle logic before processing routes
- Respond to requests for routes that *do not exist* (ie: generate "404" errors)
- Handle exceptions that occur during the processing of a route handler (ie: generate "500" series errors)

Getting Started

To implement middleware in our servers, we will begin by writing a simple middleware function that logs every request to the console. This function will be placed **before** any of our route handlers, ensuring that it gets executed for

every request:

```
app.use((req, res, next) => {
  console.log(`Request from: ${req.get('user-agent')} [${new
Date()}]`);
  next();
});
```

Notice how we make use of the aforementioned app.use() method to implement our middleware function. It looks very similar to a regular route handler, except it accepts a third parameter: ***next*** and (in this case) does not return anything to the client. It is because this function does not return anything to the client (ie: generate a "response"), that we must use the "next()" function - it simply calls the next middleware function, such as a route handler, ie:

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

> **NOTE**: If we fail to invoke the **next()** function or return a response, our server will hang and the client request will timeout.

Updating "req"

Let's continue the example by updating the "req" object in our middleware example to include a "log" property that simply stores the output of the log entry as a string. We can use this value in a subsequent route handler and send it back to the client, ie:

```
app.use((req, res, next) => {
```

Restricting Route Access

Another common use for middleware is to **restrict** route access for a specific route. This can be be accomplished by placing your middleware function as a *parameter* to the route handling function that requires restricted access. For example:

```
function randomDeny(req, res, next) {
  let allowed = Math.floor(Math.random() * 2); // 0 or 1

  if (allowed) {
    next();
  } else {
    res.status(403).send('Access Denied');
  }
}

app.get('/secure', randomDeny, (req, res) => {
  res.send('Welcome!');
});
```

Here, we have implemented our middleware function as "randomDeny", which randomly generates either a 0 or 1. If a 1 is generated, the "next()" function is invoked, allowing the route to be processed as normal. However, if a 0 is generated, a response, including the 403 - Forbidden error code is generated, informing the user that they do not have access (we could also redirect them to a "login" or "register" page, etc).

To ensure that this middleware function only affects the "/secure" route, we place it as the second parameter *before* the callback function.

404 Errors

As a final example of how to implement middleware in our server.js code - let's create a custom "404" error to send to the client if it has requested an unknown route (ie: a route that we have not created a handler for):

```
// Other route handlers, middleware, etc ...

app.use((req, res, next) => {
  res.status(404).send("404 - We're unable to find what you're looking for.");
});

// app.listen()
```

Here, we have created a middleware function using the familiar "use()" function. However, the main difference is where it is placed, ie: *below* all of our other middleware functions / route handlers. By placing it in this way, we can ensure that it *only* gets executed if none of the other route handlers return a response to the client.

Types of Middleware

Now that we have seen how middleware is typically implemented within an Express application, let's quickly review the 5 types of middleware available:

Application-Level Middleware

Application-level middleware is bound to your entire application and can run when every request comes in or only when it matches a specified route.

In the examples above, we have implemented "Application-level middleware".

Router-Level Middleware

Router-level middleware works the same way as application middleware but is attached to a separate router instance. Essentially, instead of "app.use()", a separate `express.Router()` instance is created and the middleware is applied to it, ie:

```
const userRouter = express.Router();

userRouter.use((req, res, next) => {
  console.log('userRouter Middleware!');
  next();
});
```

> For more information on express.Router(), see the official documentation in the official Express Routing documentation.

Error-Handling Middleware

Error-handling middleware is defined with 4 parameters in the callback function, ie: (err, req, res, next). We must specify all 4 parameters so that express can differentiate it from a regular middleware function. Error handling middleware is invoked either when a regular middleware function calls next(err) instead of next(), or when exceptions occur in your route handlers. Like our "404" example above, error handling middleware should be placed *below* your route handlers. For example:

```
app.get('/error-test', (req, res) => {
  throw new Error('Error Test');
```

Built-In Middleware

There are three types of built-in middleware functions available for us to use:

express.static()

This is what we used when sending "static" files (ie: "css" files, images, etc) in the "CSS & Images" section of the "Simple Web Server using Express.js" notes, ie:

```
app.use(express.static('public'));
```

express.json()

This is used to parse "JSON" formatted payloads, and make the result available on the "req" object. For example:

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

app.post('/json-test', (req, res) => {
  res.send(req.body);
});
```

express.urlencoded()

This is nearly identical to "express.json", except this is used to parse data from a web form using the default "enctype", (ie: "application/x-www-form-urlencoded").

> **NOTE:** The “extended” option utilizes the "qs" library which enables rich objects and arrays to be encoded into the URL-encoded format, allowing

> for a JSON-like experience with URL-encoded. For more information, please see the qs library.

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

app.post('/urlencoded-test', (req, res) => {
  res.send(req.body);
});
```

Third-Party Middleware

Since Express 4.x, previously included middleware that did common things such as handle cookies, or handle file uploads, have been moved to individual third-party middleware packages.

For example, parsing cookies requires the installation of cookie-parser:

```
$ npm install cookie-parser
```

```
const cookieParser = require('cookie-parser');

// load the cookie-parsing middleware
app.use(cookieParser());
```

> For a list of supported, third party middleware, refer to the official documentation.

Example Code

You may download the sample code for this topic here:

> Advanced-Routing-Middleware

Document Structure

HTML is the HyperText Markup Language. It allows us to write *content* in a document, just as we would in a file created by a word processor. Unlike a regular text file, it also includes structural and layout information about this content. We literally *mark up* the text of our document with extra information.

Terminology

When talking about HTML's markup, we'll often refer to the following terms:

- content: any text content you want to include can usually be written as-is.
- tag: separated from regular content, tags are special text (names) wrapped in `<` and `>` characters, for example the paragraph tab `<p>` or the image tag `<img>`.
- element: everything from the beginning of an opening tag to the closing tag, for example: `<h1>Chapter 1</h1>`. Here an element is made up of an `<h1>` tag (i.e., opening Heading 1 tag), the text content `Chapter 1`, and a closing `</h1>` tag. These three components taken together create an `h1` element in the document.
- attribute: optional characteristics of an element defined using the style `name` or `name="value"`, for example `<p id="error-message" hidden>There was an error downloading the file</p>`. Here two attributes are included with the `p` element: an `id` with value `"error-message"` (in quotes), and the `hidden` attribute (note: not all attributes need to have a value). Full list of common attributes.

- entity: special text that should not be confused for HTML markup. Entities begin with `&` and end with `;`. For example, if you need to use the `<` character in your document, you need to use `<` instead, since `<` would be interpreted as part of an HTML tag. ` ` is a single whitespace and `&` is the `&` symbol. Full list of named entities.

HTML Documents

The first HTML page ever created was built by Tim Berners-Lee on August 6, 1991.

Since then, the web has gone through many versions:

- HTML - created in 1990 and standardized in 1997 as HTML 4
- xHTML - a rewrite of HTML using XML in 2000
- HTML5 - the current standard.

Basic HTML5 Document

Here's a basic HTML5 web page:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>My Web Page</title>
  </head>

  <body>
```

Let's break this down and look at what's happening.

1. `<!doctype html>` tells the browser what kind of document this is (HTML5), and how to interpret/render it

2. `<html>` the root element of our document: all other elements will be included within `<html>...</html>`.

3. `<head>` provides various information *about* the document as opposed to providing its content. This is metadata that describes the document to search engines, web browsers, and other tools.

4. `<meta>` an example of metadata, in this case defining the character set used in the document: utf-8

5. `<title>` an example of a specific (named) metadata element: the document's title, shown in the browser's title bar. There are a number of specific named metadata elements like this.

6. `<body>` the content of the document is contained within `<body>...</body>`.

7. `<!-- ... -->` a comment, similar to using `/* ... */` in C or JavaScript

8. `<h1>` a heading element (there are headings 1 through 6), which is a title or sub-title in a document.

On the Server

During the last couple of classes we learned how to create a simple web server using Node.js with the Express.js module. We will be using this code once again to create a local web server, which will be responsible for returning requests for our HTML document on the default route, ie "/".

> **NOTE:** Try to familiarize yourself with these steps, as we will be doing this over and over in class, every time we wish to create a new web server. A more detailed guide can be found as a part of the Simple Web Server using Express.js documentation.

1. Make a directory on your computer called `test-server`
2. Open Visual Studio Code and choose **File > Open** to open your newly created `test-server` folder
3. Create a new file called `server.js`
4. Open the Integrated terminal useing the keyboard shortcut (ctrl + `` ` ``) or select "View" -> "integrated terminal" from the top menu.
5. Run the **npm init** command to generate your `package.json` file (you can choose all of the defaults)
6. Run the command `npm install express`
7. Enter the following code for your `server.js` file

```
const express = require("express");
const app = express();
const HTTP_PORT = process.env.PORT || 8080;

const path = require("path");

// setup a 'route' to listen on the default url path
app.get("/", (req, res) => {
    res.sendFile(path.join(__dirname,"/views/hello.html"));
});

// setup http server to listen on HTTP_PORT
```

8. Create a new folder in `test-server` named `views`
9. Within the `views` folder, create a file called `hello.html`
10. Open the newly created `hello.html` file and paste the html code from above and save the file
11. Go back to the Integrated terminal and type the command `node server.js`. You should see the message: "server listening on: 8080"
12. Open your web browser (Chrome, Firefox, etc) and enter `http://localhost:8080` in the URL bar
13. Make sure you can see a new page with "Hello World!" in black text.

Updating your Document

1. Go back to your editor and change the `hello.html` file so that instead of "Hello World!" you have "This is my web page."
2. Save your `hello.html` file.
3. Go back to your browser and hit the **Refresh** button.
4. Make sure your web page now says "This is my web page."

Every time we update anything in our web page, we have to refresh the web page in our browser. The web server will serve the most recent version of the file on disk when it is requested.

HTML Elements

As we have seen from our discussion on basic HTML "Document Structure", HTML consists primarily of "Elements" such as `<head>`, `<body>`, `<title>`, `<h1>`, etc. Elements serve as the foundation of HTML documents, providing structure and meaning to the content. In the following sections, we will delve deeper into a subset of these elements to better understand their roles and uses.

> **NOTE:** While there are more than 100 HTML elements, our current focus is on introducing common, widely used elements to enable you to quickly create a functional, standards-compliant user interface. Elements such as "form", etc will be discussed later on in the "Working with Forms" section.
>
> For a comprehensive list of all HTML elements, see the "HTML Element Reference" available on MDN.

Metadata

Information *about* the document itself (not the document's actual content) goes in various metadata elements. These types of elements *must* be included within the `<head>` element:

- `<link>` - links from this document to external resources, such as CSS stylesheets
- `<meta>` - metadata that can't be included via other elements
- `<title>` - the document's title

Element Types (Block vs. Inline)

Before we discuss some of the main visual HTML elements, we must introduce two important distinctions:

1. Block-level elements: create a "block" of content in a page, with an empty line before and after them. Block elements expand to fill the width of their parent element. Block elements can contain other block elements, inline elements, or text.

2. Inline elements: creates "inline" content, which is part of the containing block. Inline elements can contain other inline elements or text.

Consider the following HTML content:

```
<body>
  <p>The <em>cow</em> jumped over the <strong>moon</strong>.</p>
</body>
```

Here we have a `<p>` paragraph element. Because it is a block-level element, this paragraph will fill its container (in this case the `<body>` element). It will also have empty space (margins) added above and below it.

Within this block, we also encounter a number of other inline elements. First, we have simple text. However, we also see the `<em>` and `<strong>` elements being used. These will affect their content, but not create a new block; rather, they will continue to flow inline in their container (the `<p>` element).

Content Sections (Block)

The following is a short list of some of the more commonly used "block-level" elements. Again, these are used to provide structure and meaning to the content:

- `<h1> - <h6>` - renders six levels of section headings in "bold" text. `<h1>` is the highest section level (rendered by default as the largest header size) and `<h6>` is the lowest (rendered by default as the smallest header size).
- `<p>` - represents a paragraph. Paragraphs are usually represented in visual media as blocks of text separated from adjacent blocks by blank lines and/or first-line indentation, but HTML paragraphs can be any structural grouping of related content, such as images or form fields.
- `<div>` - serves as a "generic" container for content. Unlike the previous block level elements, it does not have any style applied to it (ie: no margins, font size changes, etc).
- `<header>`, `<nav>`, `<main>`, `<article>`, `<section>`, `<footer>` - these elements are similar to the above `<div>` element, however they describe the purpose of the element and the type of content it contains. These types of elements are known as "Semantic Elements". For example, the `<nav>` element specifically indicates a section of a webpage that contains navigation links, guiding browsers and assistive technologies to treat this part differently from other content.

Text (Inline)

The following are some examples of common "inline" elements. Unlike "block-level" elements, these only occupy as much width on the page as required for their content (ie: they do not expand to fill the width of their parent element):

- `<em>` - used to "emphasize" a section of text. By default, this will render the text using an *italic* font style.
- `<strong>` - this element is also used to emphasize a section of text, however in this case the text will be rendered using a **bold** font style.
- `<code>` - if your text contains a fragment of computer code (like the html snippets on this page), a common way to indicate this is with the `<code>` element. By default it will render the text using the client's default monospace font.
- `<span>` - like the `<div>` element, this serves as a "generic" container for content (it does not have any style applied to it). However, unlike `<div>`, it only occupies as much width as required for the content.
- `<a>` - this element, also known as the anchor element, is used to create hyperlinks, which are clickable links that navigate to other resources. These resources can be webpages, files, email addresses, or any URL. Common attributes include:
  - href: Specifies the URL of the page the link goes to. This can be an absolute URL, a relative URL, or a local link. For local links within the same page, use the "#" symbol followed by the id of the target element. For example, href="#section1" will link to an element with id="section1" on the same page.
  - target: Specifies where to open the linked document. For example, "_blank" opens the document in a new tab or window.
  - rel: Specifies the relationship between the current document and the linked document. Common values include "noopener" and "noreferrer".

```
<a href="https://www.google.ca/" target="_blank" rel="noreferrer">Visit Google.ca</a>
```

Void (Empty) Elements

Many of the elements we've seen so far begin with an opening tag, and end with a closing tag: `<body></body>`. However, not all elements need to be closed. Some elements have no *content*, and therefore don't need to have a closing tag. We call these void elements.

An example is the `<br>` line break element. We use a `<br>` when we want to tell the browser to insert a newline (similar to using `\n` in C):

```
<p>Knock, Knock<br />Who's there?</p>
```

Other examples of void elements include `<hr>` (for a horizontal line), `<meta>` for including metadata in the `<head>`, and a dozen others.

Lists

In HTML, lists are used to organize content in a structured and easily readable format, similar to how lists are used in Microsoft Word. To achieve this, we use the following elements:

- `<ol>` - the top level ("root") element of the list - it is this element that contains the "list items". `<ol>` signifies that this is an "Ordered List" and will render the containing "list items" using numbers.
- `<ul>` - this element functions exactly the same way as the above `<ol>` element, except that it indicates an "Unordered List". This will cause the containing "list items" to render using bullet points by default instead of numbers

We define list items themselves using the following:

- `<li>` - this is the element that is used within either an "ordered" or "unordered" list to indicate the elements of the list. Each element of the list will be contained within one `<li>` or "list item".

The following is an example of an "unordered" list of items, contained within an "ordered" list (specifically the first item)

```
<ol>
  <li>
    First Item
    <ul>
      <li>Subitem 1</li>
      <li>Subitem 2</li>
      <li>Subitem 3</li>
    </ul>
  </li>
  <li>Second Item</li>
  <li>Third Item</li>
</ol>
```

Tables

Sometimes our data is tabular in nature, and we need to present it in a grid. A number of elements are used to create them:

- `<table>` - the root of a table in HTML
- `<caption>` - the optional title (or caption) of the table
- `<thead>` - row(s) at the top of the table (header row or rows)
- `<tbody>` - rows that form the main body of the table (the table's content rows)
- `<tfoot>` - row(s) at the bottom of the table (footer row or rows)

We define rows and columns of data within the above using the following:

- `<tr>` - a single row in a table
- `<td>` - a single cell (row/column intersection) that contains table data
- `<th>` - a header (e.g., a title for a column)

We can use the `rowspan` and `colspan` attributes to extend table elements beyond their usual bounds, for example: have an element span three columns (`colspan="3"`) or have a heading span 3 rows (`rowspan="3"`).

```
<table>
  <caption>
    Order Information
  </caption>

  <thead>
    <tr>
      <th></th>
      <th>Quantity</th>
      <th>Colour</th>
      <th>Price (CAD)</th>
    </tr>
  </thead>

  <tbody>
    <tr>
      <th rowspan="3">Products</th>
      <td>1</td>
      <td>Red</td>
      <td>$5.60</td>
    </tr>
    <tr>
      <td>3</td>
      <td>Blue</td>
      <td>$3.00</td>
    </tr>
    <tr>
      <td>8</td>
      <td>Blue</td>
      <td>$1.50</td>
    </tr>
  </tbody>

  <tfoot>
    <tr>
      <th colspan="3">Total</th>
      <th>$26.60</th>
    </tr>
  </tfoot>
</table>
```

Multimedia

HTML5 provides built-in support for incorporating images, videos, and audio directly into pages, enhancing multimedia experiences. The following are examples of how to use three of the most commonly used elements: `<img>`, `<video>`, and `<audio>`.

Image

To begin, we will discuss the `<img>` element - used for embedding images into the content of your page. The attributes available are:

- **src** - this is required, and contains the path to the image you want to embed.
- **alt** - used to specify alternate text for an image - providing alternative information if the image cannot be displayed due to reasons such as slow internet connections, errors in the src attribute, or when the user relies on a screen reader. This attribute is crucial for accessibility, as it allows screen readers to convey the meaning of the image.

```
<!-- External image URL, use full width of browser window -->
<img
  src="https://images.unsplash.com/
photo-1502720433255-614171a1835e?ixlib=rb-0.3.5&ixid=eyJhcHBfaWQiOjEyMDd9&s=344dfca9dc8cb137a4b1c2c711752bc5"
  alt="Toronto Sign (text)"
/>

<!-- Local file cat.jpg, limit to 400 pixels wide -->
<img src="cat.jpg" alt="Picture of a cat" width="400" />
```

HTML5 has also recently added the `<picture>` element , to allow for an optimal image type to be chosen from amongst a list of several options.

Audio

Now that we have covered the `<img>` element, let's move on to another important multimedia element in HTML: the `<audio>` element. This element is used to embed sound content, such as music or audio clips, into your page.

Here are some of the key attributes available for the `<audio>` element:

- **src** - this attribute specifies the path to the audio file you want to embed. Similar to the `<img>` element, the src attribute is required if you are not using nested `<source>` elements to define multiple audio formats.
- **controls** - by adding this attribute, you enable built-in controls for the audio player, such as play, pause, and volume. This attribute ensures that users can interact with the audio easily.
- **autoplay** - when this attribute is present, the audio will begin playing automatically as soon as it is ready. However, you should use this feature sparingly, as autoplaying audio can be intrusive / unpleasant for users.
- **loop** - this attribute causes the audio to restart from the beginning once it has finished playing, creating a continuous loop.
- **muted** - this attribute mutes the audio - useful if you want the audio to be initially silent.
- **preload** - This attribute provides a hint to the browser about whether the audio file should be preloaded. The possible values are "auto" (default), "metadata" (only preload metadata), and "none" (do not preload the audio).

```
<!-- Audio with controls showing, only MP3 source provided -->
<audio src="https://samplelib.com/lib/preview/mp3/sample-15s.mp3" controls></audio>

<!-- Audio with controls showing, multiple formats available -->
<audio controls>
  <source src="song.mp3" type="audio/mp3" />
```

Video

Finally, let's discuss the `<video>` element. Similar to the `<audio>` element, this element allows us to embed media content directly into our pages, providing an integrated player for video content.

Key attributes of the `<video>` element include:

- **src** - specifies the URL of the video file. Like the `<audio>` element, the src attribute is required if you are not using nested `<source>` elements to define multiple video formats.
- **controls** - when present, this attribute enables the default video controls such as play, pause, and volume.
- **autoplay** - if set, the video will start playing as soon as it is ready, without waiting for user interaction.
- **loop** - makes the video start over again, every time it is finished.
- **muted** - mutes the audio of the video.
- **poster** - specifies an image to be shown while the video is downloading or until the user hits the play button.
- **width** and **height** - define the dimensions of the video player.

```
<!-- External Video File, MP4 file format, show controls -->
<video
  src="http://commondatastorage.googleapis.com/gtv-videos-bucket/sample/BigBuckBunny.mp4"
  controls
></video>

<!-- Local video file in various formats, show with controls -->
<video width="320" height="240" controls>
  <source src="video.mp4" type="video/mp4" />
  <source src="video.webm" type="video/webm" />
  <p>Sorry, your browser doesn't support HTML5 video</p>
</video>
```

> **NOTE:** the `<audio>` and `<video>` elements must use source URLs that point to actual audio or video files and not to a YouTube URL or some other source that is actually an HTML page.

Validating HTML

It's clear that learning to write proper and correct HTML is going to take practice. There are lots of elements to get used to, and learn to use in conjunction. Also each has various attributes that have to be taken into account.

Browsers are fairly liberal in what they will accept in the way of HTML. Even if an HTML file isn't 100% perfect, a browser can often still render something. That said, it's best if we do our best to provide valid HTML.

In order to make sure that your HTML is valid, you can use an HTML Validator. There are a few available online:

- https://html5.validator.nu/
- https://validator.w3.org/

Both allow you to enter a URL to an existing page, or enter HTML directly in a text field. They will then attempt to parse your HTML and report back on any errors or warnings, for example: an element missing a closing tag.

Example Code

You may download the sample code for this topic here:

> HTML5-Introduction

Syntax / Selectors

In HTML5 we don't include markup related to how our page should look; instead we focus on its structure, layout, and organization. We put all this information in style sheets: text files that define CSS *selectors* and *rules* for how to style our HTML elements.

CSS allows us to specify styles, layout, positioning, and other "style" properties for HTML elements. CSS makes it possible for a page's style information to be separated from its structure and content

CSS Syntax

CSS syntax is made up of *rules*, which are broken into two parts:

1. a *selector*, specifying the element(s) that should have the rules applied
2. one or more *declarations*, which are *key/value* pairs surrounded by `{...}` braces

```
h1 {
  color: blue;
  font-size: 12px;
}
```

In this example, the *selector* is `h1`, which indicates that we want the following rules to be applied to level-1 heading elements (i.e., all `<h1></h1>` elements in the document). Next comes a list of two definitions, each ending with a `;`. These declarations follow the usual key/value syntax, with a *property* name coming before the `:`, and a *value* coming after:

- `color: blue;` says we want to use the colour (note the spelling) blue
- `font-size: 12px;` says we want the font to be 12px.

Here's another example:

```
p {
  color: red;
  text-align: center;
  text-decoration: underline;
}
```

This indicates we want all `<p></p>` elements in the document to have red, centered, underlined text.

Where to Put CSS

CSS can come from a number of sources in an HTML page:

1. Inline
2. Internal Embedded
3. External File(s)
4. The browser itself (e.g., default styles, or extra styles injected by a browser extension)

Browsers apply styles to elements using a priority order that matches the list above. If more than one style rule is specified for an element, the browser will prefer whatever is defined in Inline styles over Internal Embedded, Internal Embedded over External files, etc.

Inline Example

CSS rules can be placed directly on an element via the `style` attribute:

```
<div style="background-color: green">...</div>
```

Internal Embedded

If we want to apply the same CSS rules to more than one element, it makes more sense to *not* duplicate them on every element's `style` attribute. One solution is to use an internal embedded `<style>` element in the `<head>` or `<body>`, similar to how embedded `<script>` elements work:

```
<style>
  p {
    color: red;
  }

  div {
    background-color: blue;
    text-align: center;
  }
</style>
```

External File(s)

Putting large amounts of CSS in `<style>` elements makes our HTML harder to read and maintain (CSS is about separating style from structure), and also causes our page to perform worse in terms of load times (i.e., the styles can't be cached by the browser). To overcome this, we often include external `.css`

files via the `<link>` element within the document's `<head>`:

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="styles.css" type="text/css" />
  </head>
</html>
```

We can include many stylesheets in this way (i.e., everything doesn't have to go in one file), and we can include `.css` files on the same origin, or a remote origin:

```
<!DOCTYPE html>
<html>
  <head>
    <link
      rel="stylesheet"
      href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/
css/bootstrap.min.css"
    />
    <link rel="stylesheet" href="styles.css" type="text/css" />
  </head>
</html>
```

In the example above, the page uses the popular Bootstrap CSS styles along with some locally (i.e., local to the web server) styles in `styles.css`.

A `.css` file included in this way can also `@import` to have even more `.css` files get loaded at runtime:

```
/* Import Font Awesome */
```

In this example, the popular Font Awesome CSS library for font icons has been imported via a `.css` file.

CSS Selectors

As we have seen from the above section on "CSS Syntax", CSS selectors are patterns used to select elements within HTML documents for styling. They target elements based on attributes such as id, class, type, and their hierarchical relationships.

Tag / Type Selectors

The name of an HTML element can be used to specify the styles associated with all elements of the given type. For example, to indent all `<p>` text in our document, we could do this:

```
p {
  text-indent: 20px;
}
```

Class Selectors

Often we want to apply styles to *some* but not *all* elements of a certain kind. Perhaps we only want some of our page's `<p>` elements to have a particular look. To achieve this, we define a *class*, and then put that class on the elements that require it:

```
<style>
  .demo {
    text-decoration: underline red;
```

A class can be applied to elements that aren't of the same type:

```
<style>
    .invisible {
        display: none;
    }
</style>

<h1 class="invisible">Title</p>
<p class="invisible">This is a paragraph.</p>
```

I this example, both the `<h1>` element, and the `<p>` element will have the `display: none` style applied, hiding them so they don't appear in the page.

If we want to be more specific, and only apply styles to elements of a given type which also have a given class, we can do this:

```
<style>
  p.note {
    font-weight: bold;
  }
</style>

<p class="note">This is a paragraph that also uses the note
class.</p>
<div class="note">
  This div uses the note class too, but because we said p.note, no
styles are used.
</div>
```

An element can also have multiple classes applied, each one adding different styling:

```
<style>
  .invisible {
    display: none;
  }

  .example {
    color: green;
    background-color: red;
  }
</style>

<p class="invisible example">This is a paragraph that uses two
classes at once.</p>
```

ID Selectors

In many cases, we have only a single element that should use styles. Using a type or class selector would be overly broad, and so we tend to use an `id` instead. Recall that only one HTML element in a document can have a given `id` attribute: it must be unique.

```
<style>
  #summary {
    background-color: skyblue;
  }
</style>

<div id="summary"></div>
```

When we use the `id` as a selector, we prefix it with the `#` symbol. Notice that the HTML does *not* use the `#` symbol though.

Contextual Selectors

Another common way to write selectors is to use the position of elements in the DOM. The *context selector* indicates the context, or placement/nesting (i.e., determined by the parent node) of the element.

For example, if we want to apply styles to `<p>` elements that are children of `<div>` elements, we could do this:

```
<style>
  div p {
    font-size: 16px;
  }
</style>

<p>This paragraph will not receive the styling</p>

<div>
  <p>This paragraph will receive the styling.</p>
  <p>This paragraph will receive the styling also.</p>
</div>
```

Grouping Selectors

As our CSS grows, it's common that we'll notice that we're repeating the same things multiple times. Instead of doing this, we can group a number of selectors together into a comma-separated list:

```
html,
body {
  height: 100%;
```

Here we've used grouping twice to cut-down on the number of times we have to repeat things. In the first case, we defined a height of `100%` (full height of the window) for the `<html>` and `<body>` elements (they don't have a height by default, and will only be as tall as the content within them). We've also declared some font and color information for all the headings we want to use.

Containers for Styling

We have discussed `<div>` and `<span>` when discussing HTML, but their purpose may not have been clear. Why bother wrapping other elements in `<div>...</div>` or `<span>...</span>` when they don't change their appearance (ie: no default style applied to them)?

With CSS we can now start to take advantage of these elements as "containers", which can be used for grouping elements together for styling.

Recall that a `<div>` is a block level element, and `<span>` an inline element. Depending on how we want to group the elements and how we should apply CSS, we can use one or both. Consider the following:

```
<style>
  .info-box {
    border: solid green;
  }

  .info-box p {
    font-family: Serif;
  }

  .info-box span {
    font-weight: bold;
  }
```

Units & Properties

Many CSS values require units to be specified, for example, font sizes, widths, heights, etc. At first you might think that we should specify things in pixels; however, browsers need to work on such a wide variety of hardware and render to so many different displays (watches to billboards), we need more options. It's also important to be able to specify sizes using relative units vs. fixed, for layouts that need to adapt to changing conditions and still retain the correct proportions.

There is one exception, and that is for `0` (i.e., zero), which never needs a unit (i.e., `0px` is the same as `0%`, etc).

CSS Units

The most common units we use in CSS are:

```
1em = 12pt = 16px = 100%
```

Let's look at each of these in turn:

- `em` (the width of the capital letter `M`) - a scalable unit that is used in web media, and is equal to the current `font-size`. If the `font-size` is `12pt`, `1em` is the same as `12pt`. If the `font-size` is changed, `1em` changes to match. We can also use multiples: `2em` is twice the `font-size`, and `.5em` is half. Using `em` for sizes is popular on the web, since things have to scale on mobile vs. desktop (i.e., fixed unit sizes don't work as the screen shrinks/expands).

- `pt` - a fixed-size *Point* unit that comes from print media, where `1pt` equals `1/72` of an inch.
- `px` - pixels are fixed size units for web media (screens), and `1px` is equal to one dot on a computer display. We use `px` on the web when we need "pixel perfect" sizing (e.g., image sizes).
- `%` - the percent unit is similar to `em` in that it scales with the size of the display. `100%` is the same as the current `font-size`.
- `vw`, `vh` - the viewport width and height units are percentages of the visible space in the viewport (the part of the page you can see, the window's width and height). `1vw` is the same as `1%` of the width of the viewport, and `80vh` is the same as `80%` of the visible height.

You will also sometimes encounter other ways of measurement that use full words: `xx-small, x-small, small, medium, large, x-large, xx-large, smaller, larger, thin, medium, thick`

Here's an example that uses a number of the units mentioned above:

```
<style>
  html,
  body {
    height: 100vh;
  }

  .box {
    margin: 10px;
    font-size: 2em;
    height: 150px;
    border: medium solid black;
  }
```

CSS Colours (`color`)

CSS allows us to define colour values for many declarations. We do so by specifying a colour using one of the following notations:

- Hexadecimal Red, Green, Blue: written using 3 double-digit hex numbers, and starting with a `#` sign. Each of the 3 pairs represents a value between 0 and 255 for Red, Green, and Blue: `#000000` is pure Black and `#ffffff` is pure White, and `#ffd700` is Gold.

- RGB or RGBA notation: here the red, green, blue, and sometimes alpha (i.e., opacity) are defined in decimal notation: `#ffffff` is the same as `rgb(255, 255, 255)` and `#ffd700` is the same as `rgb(255, 215, 0)`. If we want to define how see-through the colour is (by default you can't see through a colour), we add an alpha value: `rgba(0, 191, 0, 0.5)` means that the colour will be 50% see through.

- Named colours: some colours are so common that they have their own name defined in the CSS standard. For example: `white`, `black`, `green`, `red`, but also `chocolate`, `darkorange`, `peru`, etc.

The easiest way to understand this is using a Colour Picker tool, which lets you visually see the difference in changing values.

CSS Properties and Values

A *property* is assigned to a selector in order to manipulate its style. The CSS properties are defined as part of the CSS standard. When you want to know how one of them works, or which values you can assign, you can look at the

documentation on MDN. For example:

- `text-indent`
- `color`
- `background-color`
- `border`

There are hundreds of properties we can tweak as web developers, and it's a good idea to explore what's available, and to look at how other web sites use them via the developer tools.

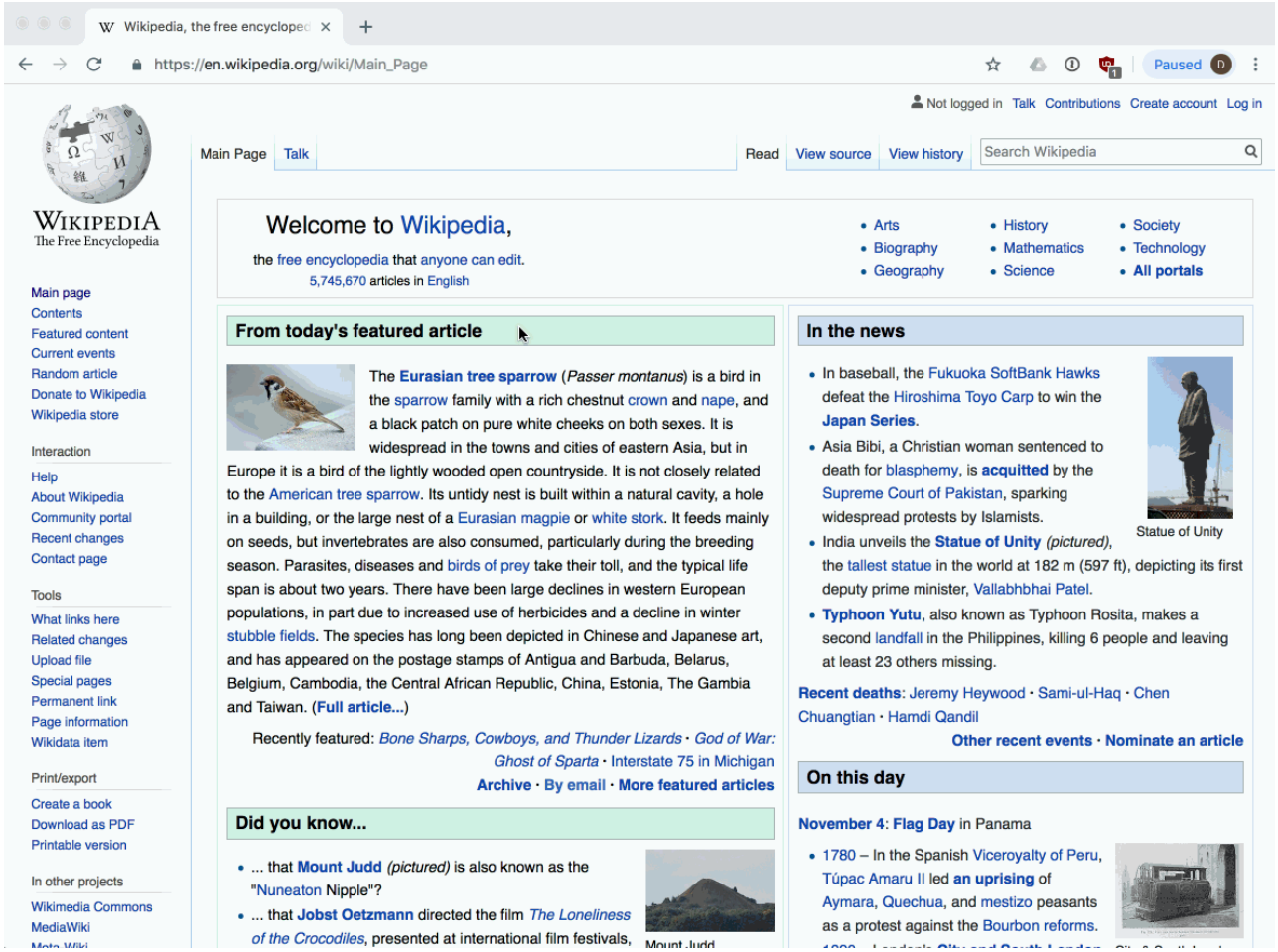
A property can have one or more values. A the possible values a property can have also comes from the standard. For example:

```
p {
  text-decoration: underline;
}

.spelling-error {
  text-decoration: red wavy underline;
}
```

The `text-decoration` property is defined to take one of a number of values, each of which is also defined in the standard.

Exploring CSS Properties and Values in the Dev Tools

By far the best way to learn about CSS is to look at how other sites use it. When you find something on the web that you think looks interesting, open your browser's dev tools and inspect the CSS Styles:



You can look at the specific properties specified for an element, or see all the *computed* styles (i.e., everything, including all default values). You can also try toggling these on and off, or double-click the values to enter your own.

CSS `text` Properties

There are dozens of properties that affect how text is rendered. These include things like the color, spacing, margins, font characteristics, etc.

```
h2 {
  color: red;
```

`font` Properties

We can use the `font-family` property to specify a font, or list of fonts, for the browser to apply to an element. The font must be available on the user's computer, otherwise the next font in the list will be tried until one is found that is installed, or a default font will be used.

In general it is safe to assume that the following fonts are available:

- `Helvetica, Arial, Verdana, sans-serif` - sans-serif fonts
- `"Courier New", Courier, monospace` - monospace fonts
- `Georgia, "Times New Roman", Times, serif` - serif fonts

You can see a list of the fonts, and OS support here.

```
h3 {
  font-family: Arial;
}

h4 {
  font-family: 'Times New Roman', Times, serif;
}

h5 {
  font-size: 18pt;
  font-style: italic;
  font-weight: 500;
}
```

Web Fonts - `@font-face`

Modern browsers also allow custom fonts to be included as external files, and

downloaded as needed by the web site. This is often the preferred method for designers, who don't want to be limited to the set of fonts available on *all* operating systems.

A font is a file that describes the curves and lines needed to generate characters at different scales. There are various formats, from `OTF` (OpenType Format) to `TTF` (TrueType Format) to `WOFF` (Web Open Font Format), etc. In order for the browser to use a new font, it has to be downloadable via one or more URLs. We then tell the browser which font files to download in our CSS via the `@font-face` property:

```
@font-face {
    font-family: "FontName"
    src: url(font.woff2) format('woff2'),
         url(font.ttf) format('truetype');
}

body {
    font-family: "FontName";
}
```

Many fonts have to be purchased, but there are some good sources of high quality, freely available fonts for your sites:

- Font Squirrel
- dafont.com
- Google Fonts

For example, we can use the popular `"Lobster"` font from Google by doing the following in our CSS:

```
@import url(https://fonts.googleapis.com/css?family=Lobster);
```

`font-size` property

Using the `font-size` property, font sizes can be given in fixed or relative units, depending on how we want our text to scale on different devices:

```
h1 {
  font-size: 250%; /* scaled to 250% of regular font size */
}

p {
  font-size: 20pt; /* size in points -- 20/72 of an inch */
}

.quote {
  font-size: smaller; /* smaller than normal size */
}

.bigger {
  font-size: 1.5em; /* 1.5 times larger than the 'M' in normal
font size */
}
```

Text Effects

There are numerous effects that can be added to text (or any element), many beyond the scope of this initial exploration of CSS. Here are a few simple examples to give you an idea

`text-shadow` allows a shadow to be added to text, giving it a 3-D style appearance. The value includes a colour, `x` and `y` offsets that determine the distance of the shadow from the text. Finally, we can also add a `blur-radius`, indicating how much to blur the shadow.

```
.shadow-text {
  text-shadow: 1px 1px 2px pink;
}
```

`text-overflow` can be used to determine what the browser should do when the amount of text exceeds the available space in a container (e.g. in a `<div>` or `<p>` that isn't wide enough). For example, we can specify that we want to `clip` the contents and not show any more, or we can automatically display `...`, the `ellipsis`.

```
<style>
  .movie-title {
    overflow: hidden;
    white-space: nowrap;
    text-overflow: ellipsis;
  }
</style>
<p class="movie-title">Pirates of the Caribbean: The Curse of the
Black Pearl</p>
```

`background` Properties

Every element has a background that we can modify. We might, for example, want to specify that the background be a certain colour; or we might want to use an image, or even tile an image multiple times (like wallpaper to create a pattern); or we might want to create a gradient, from one colour to another. All of these options and more are possible using the `background` property.

```
div.error {
  background: red;
```

Styling Links

We can control the way that links (i.e., `<a>`) appear in our document. By default they will have a solid blue underline, and when visited, a purple solid underline. If you want to remove the underline, or change it's colour to match the theme of a page, we can do that using CSS `pseudo-classes`.

With *pseudo-classes* we can specify certain states for the elements in our selector, for example:

- `a:link` - a normal, unvisited link (normally blue underline)
- `a:visited` - a link the user has visited previously (normally purple underline)
- `a:hover` - a link when hovered with the mouse
- `a:active` - a link when it is clicked (i.e., while the mouse button is pressed)

> **NOTE:** pseudo-classes can be used with any element, but we mention them here in relation to styling links, since we often need them to deal with different states for a link.

Let's alter our links so that they use blue text, with no underline. However, when hovered, add back the underline:

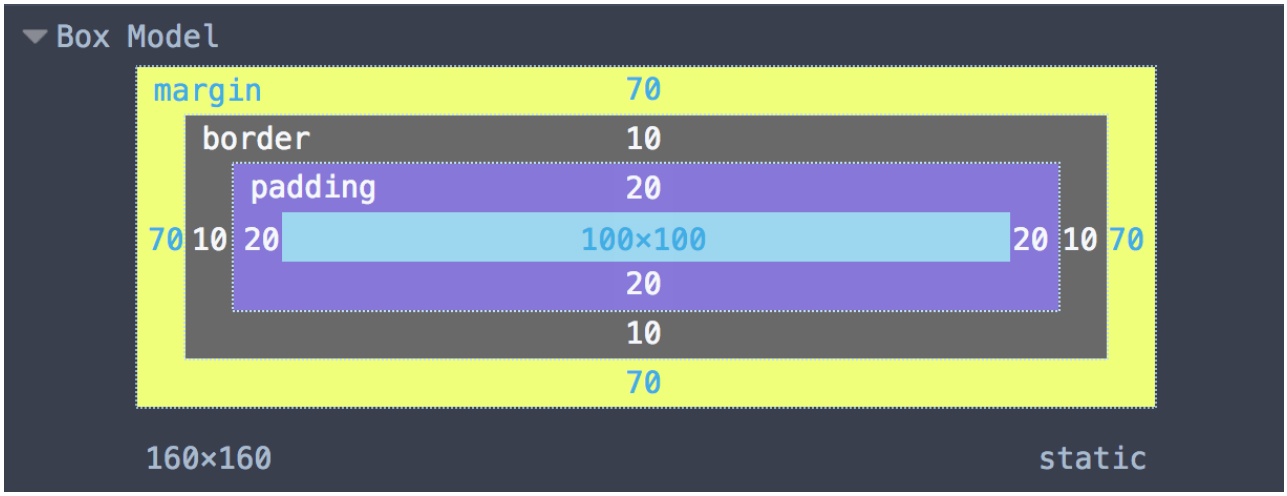
```
a:link,
a:visited {
  text-decoration: none;
}

a:hover,
a:active {
```

Box Model

All elements can be considered to be a box. The Box Model is a specification for how all the various attributes of an element's sizing relate to each other. A "box" is made up of four distinct parts:

- margin - area (whitespace) between this element and other surrounding elements
- border - a line (or lines) surrounding this element
- padding - area (whitespace) between the border and the inner content of the element
- content - the actual content of the element (e.g., text)

The easiest way to visual this is using your browser's dev tools, which have tools for viewing and altering each of these parts.



CSS "Box" Properties

The sizes (and style) of each of these can be controlled through CSS

properties:

- `margin` / (`margin-top`, `margin-right`, `margin-bottom` and `margin-left`)
- `border`
  - `border-style` / (`border-top-style`, `border-right-style`, `border-bottom-style` and `border-left-style`)
  - `border-width` / `border-top-width`, `border-right-width`, `border-bottom-width` and `border-left-width`
  - `border-color` / `border-top-color`, `border-right-color`, `border-bottom-color` and `border-left-color`
- `padding` / `padding-top`, `padding-right`, `padding-bottom` and `padding-left`

Each of these is a shorthand property that lets you specify multiple CSS properties at the same time. For example, the following are equivalent:

```
/* Use separate properties for all aspects */
.example1 {
  border-width: 1px;
  border-style: solid;
  border-color: #000;

  margin-top: 5px;
  margin-right: 10px;
  margin-bottom: 15px;
  margin-left: 20px;
}

/* Use shorthand properties to do everything at once */
```

In the code above for `margin`, notice how the the different portions of the `margin` get translated into a single line. The order we use follows the same order as a clockface, the numbers begin at the `top` and go *clockwise* around:

```
.example2 {
  /*      top right bottom left */
  margin: 5px 10px 15px 20px;
}
```

We often shorten our lists when multiple properties share the same values:

```
.example3 {
  /* Everything is different, specify them all */
  margin: 10px 5px 15px 20px;
}

.example4 {
  /* Top and bottom are both 10px, right and left are both 5px */
  margin: 10px 5px;
}

.example5 {
  /* Top, bottom, left, and right are all 5px */
  margin: 5px;
}
```

When two elements that specify a `margin` at the top and bottom are stacked, the browser will *collapse* (i.e., combine) the two into a single margin, whose size is the largest of the two. Consider the following CSS:

```
<style>
  h1 {
```

Here the stylesheet calls for a `<p>` element to have `20px` of whitespace above it. However, since the `<h1>` has `25px` of whitespace below it, when the two are placed one after the other, the distance between them will be `25px` vs. `45px` (i.e., the browser won't apply both margins, but just make sure that both margins are honoured).

"Displaying" an Element

CSS lets us control how an element gets displayed. This is a large topic, and we'll give an overview of some of the most common display types. Further study is required to fully appreciate the subtleties of each layout method.

Perhaps the easiest way to get started understanding display types is to look at what `display: none;` does:

```
<style>
  .hidden {
    display: none;
  }

  .error-msg {
    /* styles for error message UI */
  }
</style>

<div class="hidden error-msg">
  <h1>Error!</h1>
  <p>There was an error completing your request.</p>
</div>
```

When an element uses a display type of `none`, nothing will be painted to the screen. This includes the element itself, but also any of its children.

If elements don't have a display type of `none`, they get included in the render tree and eventually painted to the screen. If we don't specify a display type, the default is `inline` for inline elements (like `<a>` and `<span>`) and `block` for block-level elements (like `<p>` and `<div>`).

With `inline`, boxes are laid out horizontally (typically left to right, unless we are doing rtl), starting at the top corner of the parent.

We can also specify that an element should be `display: block;`, which will layout blocks in a vertical way, using `margin` to determine the space between them. To understand the difference, try this using this snippet of code an HTML page, and change the `display` from `block` to `inline`:

```
<style>
  h1 {
    display: block; /* try changing to `inline` */
  }
</style>
<h1>One</h1>
<h1>Two</h1>
<h1>Three3</h1>
```

We can also control the way that elements are laid out *within* an element (i.e., its children). Some of the display types for inside layout options include:

- `table` - make elements behave as though they were part of a `<table>`
- `flex` - lays out the contents according to the flexbox model
- `grid` - lays out the contents according to the grid model

A great way to learn a bit about the latter two is to work through the following online CSS learning games:

- Flexbox Froggy

- Flexbox Defense
- Grid Garden

"Positioning" an Element

Many web interface designs require more sophisticated element positioning than simply allowing everything to flow. Sometimes we need very precise control over where things end up, and how the page reacts to scrolling or movement.

To accomplish this kind of positioning we can use the CSS `position` property to override the defaults provided by the browser.

- `static` - the default, where elements are positioned according to the normal flow of the document
- `relative` - elements are positioned according to the normal flow, but with extra offsets (`top, bottom, left, right`), allowing content to overlap
- `absolute` - elements are positioned separate from normal flow, in their own "layer" relative to their ancestor element, and don't affect other elements. Useful for things like popups, dialog boxes, etc.
- `fixed` - elements are positioned separate from normal flow, and get positioned relative to the viewport.
- `sticky` - a hybrid of `relative` and `fixed`, allowing an element to be positioned relatively, but then "stick" when scrolling or resizing the viewport. This is often used for headings, which can be scrolled up, but then stay in place as you continue down into the document.

`z-index` Property

In addition to controlling how elements are positioned in the X and Y planes, we can also *stack* elements on top of each other in different layers. We achieve this through the use of the `z-index` property.

The `z-index` is a value positive or negative integer, indicting which stack level the element should be placed within. The default stack level is `0`, so using a `z-index` higher than `0` will place the content on top of anything below it.

The `z-index` is often used with `position` to place content in arbitrary positions overtop of other content. For example, a modal window that appears as a dialog box, over the site's content.

`overflow` Property

When the contents on an element are too large to be displayed, we have options as to how the browser will display the overflowing content. To do this, we work with the `overflow`, `overflow-x`, `overflow-y` properties

- `visible` - default. No scroll bars provided, content is not clipped.
- `scroll` - always include scroll bars, content is clipped and and scroll if required
- `auto` - only include scroll bars when necessary, content is clipped and and scroll if required
- `hidden` - content is clipped, no scroll bars provided.

Validating CSS

If you recall from last week, we introduced an online validator to check your HTML code for errors. There are similar tools for CSS, for example:

- https://jigsaw.w3.org/css-validator/

The above allows you to enter a URL to an existing web page, or enter CSS directly in a text field. It will then attempt to parse your CSS and report back on any errors or warnings.

Example Code

You may download the sample code for this topic here:

> CSS3-Introduction

HTML Form Elements Overview

Before we begin to implement form submission logic in our server code, let's first do a quick review of the main form elements, including:

Form

The form element serves as the primary container for housing your form, including user inputs and the submit button. It has several attributes that control its behavior, with the most common ones being **'enctype'**, **'method'**, and **'action'**.

The enctype is the encoding type. If you are working with forms that have file uploads that accompany the form data, this value should be set to: `multipart/form-data`, otherwise the default is `application/x-www-form-urlencoded`. The 'method' specifies which HTTP verb to use when making the submission request (ie: "GET: or "POST"). Finally, the 'action' attribute is the URL / route that the form will send the request to once it has been *submitted*.

```
<form method="post" enctype="application/x-www-form-urlencoded"
action="https://httpbin.org/post">
  <!-- ... -->
</form>
```

> **NOTE:** in the above example, "enctype" may be **omitted** since "application/x-www-form-urlencoded" is the default value for "enctype"

Input

The input element creates a single-line text box by default (ie: the default value for the **'type'** attribute is `text`):

```
<input type="text" name="fullName" />
```

> **NOTE:** We *must* ensure that every form control includes a "name" field, which will be used to identify the form value, when submitted.

There are also a multitude of additional *interactive* input 'types' that may be used, such as:

- **color:** Elements of type="color" provide a user interface element that lets a user specify a color, either by using a visual color picker interface or by entering the color into a text field in #rrggbb hexadecimal format.
- **date:** Elements of type="date" create input fields that let the user enter a date, either with a textbox that validates the input or a special date picker interface.
- **time:** Elements of type="time" create input fields designed to let the user easily enter a time (hours and minutes, and optionally seconds).
- **email:** Elements of type="email" are used to let the user enter and edit an email address, or, if the multiple attribute is specified, a list of email addresses.
- **number:** Elements of type="number" are used to let the user enter a number. They include built-in validation to reject non-numerical entries.

- **range:** Elements of type="range" let the user specify a numeric value which must be no less than a given value, and no more than another given value. The precise value, however, is not considered important. This is typically represented using a slider or dial control rather than a text entry box like the number input type.

- **file:** Elements of type="file" let the user choose one or more files from their device storage. Once chosen, the files can be uploaded to a server using form submission, or manipulated using JavaScript code and the File API.

Textarea

The textarea element is much like an `<input type='text'>` input, except it allows multiple lines of text. Essentially, it is a text box that has space to add a larger quantity of text, instead of just a single line of text. The textarea is useful for capturing user input that would typically be long and detailed or several sentences long.

```
<textarea name="blogEntry"></textarea>
```

Select

The select element serves as a "dropdown list" of option elements for users to choose from. Used without any attributes, it behaves exactly like a dropdown list and only permits the user to select 1 (one) "option". With the addition of the "multiple" attribute, we can allow the user to select more than one option. We can also specify a "size" attribute, to show more than a single option at a time - this will work for both `<select>` and `<select multiple>` elements.

When submitted, the value is the text in the "value" attribute for the selected option. When multiple options are selected, an array of "value" attributes are submitted, ie: `["car", "bus"]`.

```
<select name="pet">
  <option value="">-- Please choose an option --</option>
  <option value="dog">Dog</option>
  <option value="cat">Cat</option>
  <option value="hamster">Hamster</option>
  <option value="parrot">Parrot</option>
</select>

<select multiple name="transportation">
  <option value="car">Car</option>
  <option value="motorcycle">Motorcycle</option>
  <option value="bus">Bus</option>
  <option value="jet">Private Jet</option>
</select>
```

Checkbox

The checkbox is actually another "type" of input element. These are rendered as boxes that when clicked, become marked as "checked" and are rendered with a check mark. When submitted, the values are either "on" (for checked), or *undefined* if left unchecked.

```
<input type="checkbox" name="active" /> Active
```

Radio Button

The radio button is similar to "checkbox" in that it is also a "type" of input. However, radio buttons are used when you wish to present a list of options for the user. When grouped together by using the same "name" attribute, they are "mutually-exclusive" (ie: checking one radio button in the group, will automatically deselect the previously checked radio button). When submitted, the value sent is the text in the "value" attribute for the checked radio button.

```
<input type="radio" name="fastFood" value="hamburger" /> Hamburger
<br />
<input type="radio" name="fastFood" value="pizza" /> Pizza <br />
<input type="radio" name="fastFood" value="sandwich" /> Sandwich
<br />
```

Label

The label element is used to provide a label for a form control. You can use the label's 'for' attribute to make the label clickable to focus its associated input (identified by a unique "id"). Alternatively, you can wrap the label text and form control inside a parent "label" element. This adds a nice touch of usability to forms and can make it easier to focus on / interact with areas associated with a label.

```
<label for="fullName">Full Name</label><br />
<input type="text" name="fullName" id="fullName" />

<label><input type="checkbox" name="active" /> Active</label>
```

Hidden

The hidden input type is used to include data that cannot be *seen* or *modified* by users when a form is submitted. For example, "the ID of the content that is currently being ordered or edited, or a unique security token".

```
<input type="hidden" name="productID" value="193774" />
```

Submit

Every form element should contain a "submit" button that will start the process of submitting the form. This typically includes generating an HTTP request using the method identified in the "method" attribute, and sending it to the destination in the "action" attribute. The encoding of the data in the request is controlled by the "enctype" attribute.

A submit button can be created by either using a input element with type="submit" or a button with `type="submit"`.

```
<input type="submit" value="Submit" />
<!-- or -->
<button type="submit">Submit</button>
```

Processing URL Encoded Form Data

Once we have completed the HTML to correctly **render** our <form> element, we can concentrate on handling the form submission within our server logic.

Let's begin by first creating a Simple Web Server using Express.js that returns a valid HTML document for the "/" route. Somewhere in the <body> of this document, create a simple form using controls from our HTML Form Elements Review, for example:

```
<form method="post" action="/addEntry">
  Full Name<br />
  <input type="text" name="fullName" /><br /><br />

  Blog Entry<br />
  <textarea name="blogEntry"></textarea><br /><br />

  Pet<br />
  <select name="pet">
    <option value="">-- Please choose an option --</option>
    <option value="dog">Dog</option>
    <option value="cat">Cat</option>
    <option value="hamster">Hamster</option>
    <option value="parrot">Parrot</option></select
  ><br /><br />

  Transportation<br />
  <select multiple name="transportation">
    <option value="car">Car</option>
    <option value="motorcycle">Motorcycle</option>
    <option value="bus">Bus</option>
    <option value="jet">Private Jet</option></select
  ><br /><br />

  Fast Food<br />
  <label><input type="radio" name="fastFood" value="hamburger" /> Hamburger </label><br />
  <label><input type="radio" name="fastFood" value="pizza" /> Pizza </label><br />
  <label><input type="radio" name="fastFood" value="sandwich" /> Sandwich </label><br /><br />

  <label><input type="checkbox" name="active" /> Active </label><br /><br />

  <input type="hidden" name="productID" value="193774" />

  <button type="submit">Submit</button>
</form>
```

Notice how the form element has *omitted* the "enctype" attribute on the "form" element, as well as updated the action to `"/addEntry"` (instead of `"https://httpbin.org/post"`). This is because we will be using the **default** enctype ("application/x-www-form-urlencoded") and we wish to process the form on our own server, *instead* of using "httpbin".

Body Parsing Middleware

As mentioned in the Middleware / Built-In Middleware discussion, we require some "preprocessing" on the "req" object, before we can access the form data in our routes. For example, if we were to submit the form now, the **data** sent would look something like this:

```
fullName=John+Smith&blogEntry=Cool+Blog&pet=cat&transportation=car&transportation=bus&fastFood=pizza&active=on&productID=193774
```

While it does contain the data from the form, it is very difficult to work with and requires manual parsing of the string. Instead, we would prefer an object in memory:

```
{
  fullName: "John Smith",
  blogEntry: "Cool Blog",
  pet: "cat",
  transportation: [ "car", "bus" ],
  fastFood: "pizza",
  active: "on",
  productID: "193774"
}
```

This is where "Middleware" comes in, ie: perform some processing on the HTTP Request "body" data, before sending it to our route handlers in the "req" object.

To achieve this, we can use the built-in middleware: express.urlencoded():

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

Writing The Route Handler

In the example above, the form "action" attribute is set to "/addEntry". If the form were to be submitted now, Express would return a "404" error with a response containing the text "Cannot POST /addEntry".

To remedy this, create a "POST" route for "/addEntry":

```
app.post('/addEntry', (req, res) => {
  res.send(req.body);
});
```

once the "/addEntry" route is in place (beneath our "express.urlencoded()" middleware), we can try submitting the form again. This time, we should see the form data rendered as JSON in the browser.

Special Consideration ("checkbox")

As previously mentioned in the "checkbox" section of the "HTML Form Elements Review", checkboxes submit the string "on" when checked and *undefined* when unchecked. Instead, we would prefer that the value be *true* or *false*. As a simple fix for this, we can add the following code:

```
req.body.active = req.body.active ? true : false;
```

Here, we see if the "active" value is truthy (ie: not false, 0, -0, 0n, "", null, undefined, or NaN) and if it is, set it explicitly to "true". If the value is "falsy" (ie: *undefined*), then set it explicitly to "false".

```
app.post('/addEntry', (req, res) => {
  req.body.active = req.body.active ? true : false;
  res.send(req.body);
});
```

Processing Multipart Form Data

If an HTML <form> element requires **file uploads** as well as regular form data, then we can no longer use the default "enctype" value `application/x-www-form-urlencoded`. Instead, we must use the aforementioned `multipart/form-data`. For example, consider the following form using input type="file" as well as a simple text input:

```
<form method="post" action="/uploadEntry" enctype="multipart/form-data">
  <label>
    File Description<br />
    <input type="text" name="fileDescription" />
  </label>
  <br /><br />

  <label>
    Avatar Image<br />
    <input type="file" name="avatar" />
  </label>
  <br /><br />

  <button type="submit">Upload Image</button>
</form>
```

In the above code, we have modified the "action" to submit to a new route "/uploadEntry" as well as modified the enctype to use "multipart/form-data".

Processing the Data with Middleware

Recall, when working with url-encoded data, we had to use "Middleware" (specifically the built-in middleware: express.urlencoded()) to process the data and deliver it in a format that we can process. This is also the case for "multipart/form-data", however there are no available built-in middleware functions that we can use. Instead, we will use the popular third-party middleware: "Multer"

> Multer is a node.js middleware for handling multipart/form-data, which is primarily used for uploading files. It is written on top of busboy for maximum efficiency.
>
> **NOTE:** Multer will not process any form which is not multipart (multipart/form-data).

To get started using Multer, we will need to install it:

```
npm install multer
```

Next, we must *require* the module and configure the middleware, ie:

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

Default (Simple) configuration

To begin, we will use the default configuration for Multer. All that is required is a "dest" property that defines where the files will go once uploaded. In this

case, we will use the folder "uploads/":

```
const upload = multer({ dest: 'uploads/' });
```

Writing The Route Handler

With our middleware in place, we can now write our route handler for the route defined in our "action" attribute: "/uploadEntry". When using Multer, we not only have access to the "req.body" property to get the data submitted in the form, but also a "req.file" property to get information about the uploaded file:

```
app.post('/uploadEntry', upload.single('avatar'), (req, res) => {
  res.send({ body: req.body, file: req.file });
});
```

Notice how we apply the middleware on the specific route, rather than using "app.use()". Additionally, since we're uploading a single image, we invoke the "single" method, passing the "name" attribute for our `<input type="file">` (ie: "avatar").

If we try submitting the form again, we should see a result in the browser with both the form and file upload information (ie: "req.body" & "req.file").

While this does indeed work and the file is uploaded to the correct destination (the "uploads" folder, as specified), we do not have any control over how the file is *named*. Additionally, we lose the file extension associated with the file. To gain more control over the file upload, we will need to perform some additional configuration.

Additional Configuration (diskStorage)

In order to customize the filename of the upload, we will need to use the "diskStorage" option when we configure our "upload" middleware. Here, instead of creating "upload" using `multer({ dest: 'uploads/' });`, we will use the following "diskStorage" configuration:

```
const storage = multer.diskStorage({
  destination: 'uploads/',
  filename: function (req, file, cb) {
    cb(null, Date.now() + path.extname(file.originalname));
  },
});

const upload = multer({ storage: storage });
```

Here, we specify the filename to be a current date, using "Date.now()", ie:

> The number of milliseconds elapsed since the epoch, which is defined as the midnight at the beginning of January 1, 1970, UTC.

We also retain the current extension using path.extname() from the **"path"** module: `const path = require("path");`

Ephemeral / Read-Only File Systems

As a final note, it's important to consider that many cloud-based hosting

providers either have an "ephemeral" file system (ie: data is not persisted across deploys and restarts) or the file system is read-only. In this case, if we wish to persist file uploads, we could use a library like "streamifier" to create a readable stream of the file data, rather than store it. We could then pass the data to a free service like "Cloudinary" to host the file.

> For more information, see the Cloudinary documentation on Uploading assets / Upload data stream

Example Code

You may download the sample code for this topic here:

> Working-With-Forms

Introduction

Express.js is a powerful library for helping us create web servers in Node.js. In very few lines of code we can send / receive data in a way that is very straightforward and easy to understand. Recall our first example, where we were able to create two routes: "/" and "/about", each corresponding to a specific response from our server:

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

const HTTP_PORT = process.env.PORT || 8080;

app.get('/', (req, res) => {
  res.send("Hello World<br /><a href='/about'>Go to the about
page</a>");
});

app.get('/about', (req, res) => {
  res.sendFile(path.join(__dirname, '/views/about.html'));
});

app.listen(HTTP_PORT, () => {
  console.log(`server listening on: ${HTTP_PORT}`);
});
```

In the above example, we make use of the **get** method of the app object to define a route and a callback function that's executed when the route is encountered. We can leverage the 2nd parameter "res" to send either an HTML formatted string (for route "/"), or a static html page (for route "/about").

If we wanted to send (JSON formatted) data only, we can use the following

route (/getData):

```
app.get('/getData', function (req, res) {
  let someData = {
    name: 'John',
    age: 23,
    occupation: 'developer',
    company: 'Scotiabank',
  };

  res.send(someData);
});
```

This will return the JSON-formatted string:

```
{ "name": "John", "age": 23, "occupation": "developer",
"company": "Scotiabank" }
```

The important thing to notice here is that our server can return HTML formatted strings, static HTML (.html) files, and JSON data.

Returning HTML & Data

If we want to return a valid HTML5 page to the client that actually references data stored on the server, one solution would be to build a string that contains both **HTML code** and **data**, ie:

```
app.get('/viewData', function (req, res) {
  let someData = {
    name: 'John',
    age: 23,
```

While this will work to send a valid HTML5 page containing our data back to the client, it's clearly not the best way to approach this problem. What if we had a complex page that contains data in different places throughout the layout? We would be building out an enormous string containing normal, static html and in a few places, inserting a reference to our data (someData object). Wouldn't it be better if we could just write a normal HTML document that **references** the data, instead of having to build one huge string for the whole page?

Template Engines

Fortunately, we can leverage "template engines" with Express to solve this exact problem. From the express.js documentation:

> A template engine enables you to use static template files in your application. At runtime, the template engine replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client. This approach makes it easier to design an HTML page.

This sounds like exactly what we need and there are a number of popular options that we can choose from, such as:

- "Pug"
- "Express Handlebars"
- "EJS"

In the next section, we will take a look at "EJS":

> A simple templating language that lets you generate HTML markup with plain JavaScript. No religiousness about how to organize things. No reinvention of iteration and control-flow. It's just plain JavaScript.

EJS (Embedded JavaScript Templates)

EJS is described as "a simple templating language that lets you generate HTML markup with plain JavaScript. No religiousness about how to organize things. No reinvention of iteration and control-flow. It's just plain JavaScript."

It contains features that will help us generate HTML that renders complex data. For example, consider the problem with our “/viewData” route from the introduction; we can leverage the EJS template engine to write a simple (separate) HTML5 document that references the data using special delimiters, ie: `<%=` and `%>`, rather than returning a long, complex string from our route handler.

Getting Started

To begin, create the following file in your “views” directory and name it “viewData.ejs”:

```
<!DOCTYPE html>
<html>
  <head>
    <title>View Data</title>
  </head>

  <body>
    <table border="1">
      <tr>
        <th>Name</th>
```

This is a much cleaner approach. We no longer have to generate the full page as a string within our “/viewData” route and most importantly, all of the **view** logic (HTML) is separate from our **control** logic (routing).

In order to set this up correctly and get express to understand the file above, we need to modify our server code slightly:

1. The first thing that we need to do is download / install the EJS package using NPM. Open a terminal in Visual Studio Code (ctrl + ` or View -> Integrated Terminal) and make sure that your working directory is somewhere within your project and run the command

   ```
   npm install ejs
   ```

   This will install the "ejs" package in the same way that we installed the "express" package and update the dependencies in our package.json file:

   ```
   "dependencies": {
     "ejs": ...,
     "express": ..."
   }
   ```

2. Next, our server needs to know how to handle HTML files that are formatted using ejs, so near the top of our code (after we define our "app"), add the line:

   ```
   app.set('view engine', 'ejs');
   ```

   This will tell our server that any file with the ".ejs" extension (instead of ".html") will use the EJS "engine" (template engine).

3. The final step involves updating our "/viewData" route to "render" our EJS file with the data:

```
app.get('/viewData', function (req, res) {
  let someData = {
    name: 'John',
    age: 23,
    occupation: 'developer',
    company: 'Scotiabank',
  };

  res.render('viewData', {
    data: someData,
  });
});
```

Now, the route no longer returns a string consisting of our HTML + data using res.send(), but instead invokes the render method on the response object (res). We pass the name of our new file without the extension (ie: "viewData" instead of "viewData.ejs"), and a "data" object to hold all of our data (someData).

EJS Syntax

Before we begin to discuss the more advanced features of EJS, we must first become familiar with the syntax. For example, we have seen that `<%= ... %>` is used to render a specific value within our template. However, we should understand that this delimiter ("tag"), *also* escapes any HTML contained in the value (ie: "
" will be rendered as "
" so that it appears as text, *instead* of a new line).

The `<%= ... %>` is not the only delimiter available to us. EJS also provides a

number of *opening* and *closing* delimiters ("tags") that control how a value is rendered within the template.

Delimiters (Tags)

- `<%= ... %>` (HTML Escaped)

  As we have seen, this tag outputs the value into the template (HTML escaped). For example: `<br />` will be rendered as: `<br />`, when using the tag:

  ```
  <%= someValue %>
  ```

- `<%- ... %>` (Unescaped)

  This tag works exactly as the above `<%=` tag, except the value is *not* HTML escaped.For example: `<br />` will be rendered as: `<br />`, when using the tag:

  ```
  <%- someValue %>
  ```

- `<%# ... %>` (Comment)

  This tag is used when we wish add a comment to our templates that will **not** be output in the final HTML, ie:

  ```
  <%# This is a comment that will not be rendered %>
  ```

- `<% ... %>` (Scriptlet)

This is the tag that will enable us to insert **logic** into our templates (discussed further down). For example, if our "data" object contained an array of colors, ie: `['red','green','blue]`, we could use the following "scriptlet" tags to render the contents using a "forEach" loop:

```
<% data.colors.forEach((color) => { %>
  <%= color %>
<% }) %>
```

> **NOTE:** Delimiters that output a value (ie "HTML escaped" / "unescaped") are also capable of executing JavaScript expressions. For example, if "someValue" is a string, we could use the following code:
>
> `<%= someValue.toUpperCase() %>`

Includes / "Partials"

When using EJS, it is also possible to place reusable blocks of our user interface in separate files, such as a common header or an in-page modal window / dialog box. To achieve this, EJS uses an "include" function that may be used in one of the output tags (ie: "HTML escaped" or "unescaped", however since these included .ejs files typically use HTML, the "unescaped" delimiter is more commonly used).

To see how this works in practice, we will create a "partials" folder within the "views" folder (this will help us separate the reusable templates, from the "page" templates)

Next, (within the "partials" folder) create a file called **"header.ejs"**:

```
<h1>EJS Practice - <%= page %></h1>
<hr />
<a href="/">Home</a> | <a href="/about">About</a> | <a
href="/viewData">View Data</a>
<hr />
<br />
```

Notice how our partial template includes a block of reusable HTML as well as an "HTML escaped" tag to render a variable called "page". To render this template *inside* another template, we can use the aforementioned "include" function:

File: viewData.ejs

```
<%- include('partials/header', {page: '/viewData'}) %>
```

Here, we have used the "unescaped" delimiter to ensure that the HTML within the "partial" is correctly rendered. Additionally, the second parameter contains an object that we an pass to our partial (in this case, the value of the "page" variable)

> **NOTE:** Partial views have access to the data in the template in which they are placed. For example, if the "header" partial (above) was placed in the viewData template, it would have access to the "data" object and could render "data.name", for example

Logic

Using the "Scriptlet" delimiter (ie: `<% ... %>`), we can easily insert JavaScript code into our templates. This is one of the key benefits of using EJS:

> "We love JavaScript. It's a totally friendly language. All templating languages grow to be Turing-complete. Just cut out the middle-man, and use JS!
>
> https://ejs.co

if / else

To conditionally render portions of our template (HTML), we can use a simple if / else statement. To get this to work correctly, each "line" of JavaScript code should be placed inside a scriptlet delimiter. For example, say we wish to conditionally show our developer "John":

```
let someData = {
  name: 'John',
  age: 23,
  occupation: 'developer',
  company: 'Scotiabank',
  visible: true,
};
```

Notice, we have added a "visible" property that we can reference before we render "someData" in our view. Using a simple if / else statement, we can easily hide or show rows in the table:

File: viewData.ejs

```
<% if (data.visible) { %>
  <tr>
    <td><%= data.name %></td>
    <td><%= data.age %></td>
    <td><%= data.occupation %></td>
```

Iterating over Collections

In addition to conditionally rendering portions of our templates, we may also need to display the content of an array / collection. This may be done using the usual constructs, ie "for", "for...of", "while", "forEach()", etc. For example, if our someData object contained an **array** of objects:

```
let someData = [
  {
    name: 'John',
    age: 23,
    occupation: 'developer',
    company: 'Scotiabank',
  },
  {
    name: 'Sarah',
    age: 32,
    occupation: 'manager',
    company: 'TD',
  },
];
```

we could use the "forEach()" method to display each object in our table:

File: viewData.ejs

```
<% data.forEach(user=>{ %>
  <tr>
    <td><%= user.name %></td>
    <td><%= user.age %></td>
    <td><%= user.occupation %></td>
    <td><%= user.company %></td>
```

Please note that we are not limited to the forEach() loop when iterating over data. As mentioned above, we could also use another construct, such as the "for...of" loop:

```
<% for (const user of data){ %>
  <tr>
    <td><%= user.name %></td>
    <td><%= user.age %></td>
    <td><%= user.occupation %></td>
    <td><%= user.company %></td>
  </tr>
<% } %>
```

"Nesting" Logic

The "scriptlet" tag is extremely powerful - it let's us inject JavaScript into our views to control how our data is displayed. In the above examples, we have only used single pieces of logic at a time (ie: "if/else", "forEach()", etc), but it is also possible that this logic may be "nested".

For example, maybe each of our "users" in the "someData" array has a "visible" property as well. We would like to render each of the elements in the array, but **also** hide a user if their visible property is set to *false*

```
let someData = [
  {
    name: 'John',
    age: 23,
    occupation: 'developer',
    company: 'Scotiabank',
    visibility: false,
  },
  {
```

File: viewData.ejs

```
<% for (const user of data){ %>
  <% if(user.visible){ %>
    <tr>
      <td><%= user.name %></td>
      <td><%= user.age %></td>
      <td><%= user.occupation %></td>
      <td><%= user.company %></td>
    </tr>
  <% }else{ %>
    <tr>
      <td colspan="4">User: '<%= user.name %>' has hidden their
information</td>
    </tr>
  <% } %>
<% } %>
```

Layouts

EJS does not natively support "layouts". Typically, the structure of an application using EJS as its template engine will feature a common "header, "footer", "sidebar", etc with every page, ie:

```
<body>
  <%- include('header') %>

  <%# Page Content / Data Here %>

  <%- include('footer') %>
</body>
```

If you wish to customize the 'header' or 'footer' based on the current page,

data can be sent to each of the partials separately. For example, one common task is for a navigation bar within the 'header' to highlight the link for the current page. For example, if the user is currently viewing the "/about" route, then "About" should be highlighted:

File: header.ejs

```
<h1>EJS Practice - <%= page %></h1>
<hr />
<a href="/">Home</a> | <a
href="/about"><strong>About</strong></a> | <a
href="/viewData">View Data</a>
<hr />
<br />
```

To achieve this, we can pass the current route to the partial view. Currently, we are passing this value as "page":

File: viewData.ejs

```
<%- include('partials/header', {page: '/viewData'}) %>
```

Therefore, we can leverage the "unescaped" tag to conditionally highlight each of the options using the "ternary" operator, by checking the "href" attribute against the "page" value:

```
<h1>EJS Practice - <%= page %></h1>
<hr />
  <a href="/"><%- (page=="/") ? '<strong>Home</strong>' : 'Home'
%></a> |
  <a href="/about"><%- (page=="/about") ? '<strong>About</strong>'
: 'About' %></a> |
```

> **NOTE:** If you wish to use EJS with full layout support, consider the NPM package: express-ejs-layouts

Example Code

You may download the sample code for this topic here:

> Template-Engines

Introduction to Postgres

"Data Persistence" (the ability to "persist" or "save" new, updated or deleted information) is a vital part of any web application project. For example, this could be registering new users, deleting users, updating profile information or payment data for users, viewing saved files or uploaded images, etc. etc. To truly create an "application" we must be able to work with (and persist) data.

Fortunately, there are many different database systems that we can leverage to accomplish this notion of "data persistence". These range from powerful "relational" database systems, including: Microsoft SQL Server, Oracle, MySQL, PostgreSQL, and many others as well as "NoSQL" database systems such as Amazon's DynamoDB, Azure Cosmos DB and MongoDB.

We will be focusing specifically on PostgreSQL and MongoDB - today, we will look at how we can work with a PostgreSQL database in a node.js environment.

PostgreSQL (Postgres)

From the PostgreSQL site, postgresql.org:

> "PostgreSQL (also known as "Postgres") is a powerful, open source object-relational database system. It has more than 15 years of active development and a proven architecture that has earned it a strong reputation for reliability, data integrity, and correctness. It runs on all major operating systems, including Linux, UNIX (AIX, BSD, HP-UX, SGI IRIX, macOS, Solaris, Tru64), and Windows. It is fully ACID compliant, has full support for foreign keys, joins, views, triggers, and stored procedures (in

> multiple languages). It includes most SQL:2008 data types, including INTEGER, NUMERIC, BOOLEAN, CHAR, VARCHAR, DATE, INTERVAL, and TIMESTAMP. It also supports storage of binary large objects, including pictures, sounds, or video. It has native programming interfaces for C/C++, Java, .Net, Perl, Python, Ruby, Tcl, ODBC, among others, and exceptional documentation.

This is a great choice for us for multiple reasons; it is open source, highly available, standards compliant and most importantly, works nicely with node.js.

To get started, proceed to https://neon.tech and click on the **"Log in"** link at the top and log in with your GitHub account. Once you're logged in, follow the below steps to set up the database:

1. in the "Get started with Neon for Free" page, enter a value for project Name, ie: **Seneca** and Database Name, ie: **SenecaDB** (We can add more databases later)

2. Leave "region" as the default value and Click the **Create Project** Button.

3. At the next screen, you should see a dropdown with "Connection String" selected. Click this and choose **Parameters only**.

4. Next, click the **eye icon** to reveal your password (**NOTE** Also consider checking the "Pooled connection" checkbox if this app will be deployed in a serverless environment, such as Vercel)

5. Copy the **PGHOST**, **PGDATABASE**, **PGUSER** and **PGPASSWORD** values

pgAdmin

Now that we have our brand new Postgres database created in Neon.tech, why

don't we try to connect to it using the most popular GUI tool for Postgres; pgAdmin. If you're following along from the lab room, it should already be installed. However, if you're configuring your home machine, you will need to download pgAdmin:

- https://www.pgadmin.org/download/

Once it is installed and you have opened the app, we need to configure it to connect to our database:

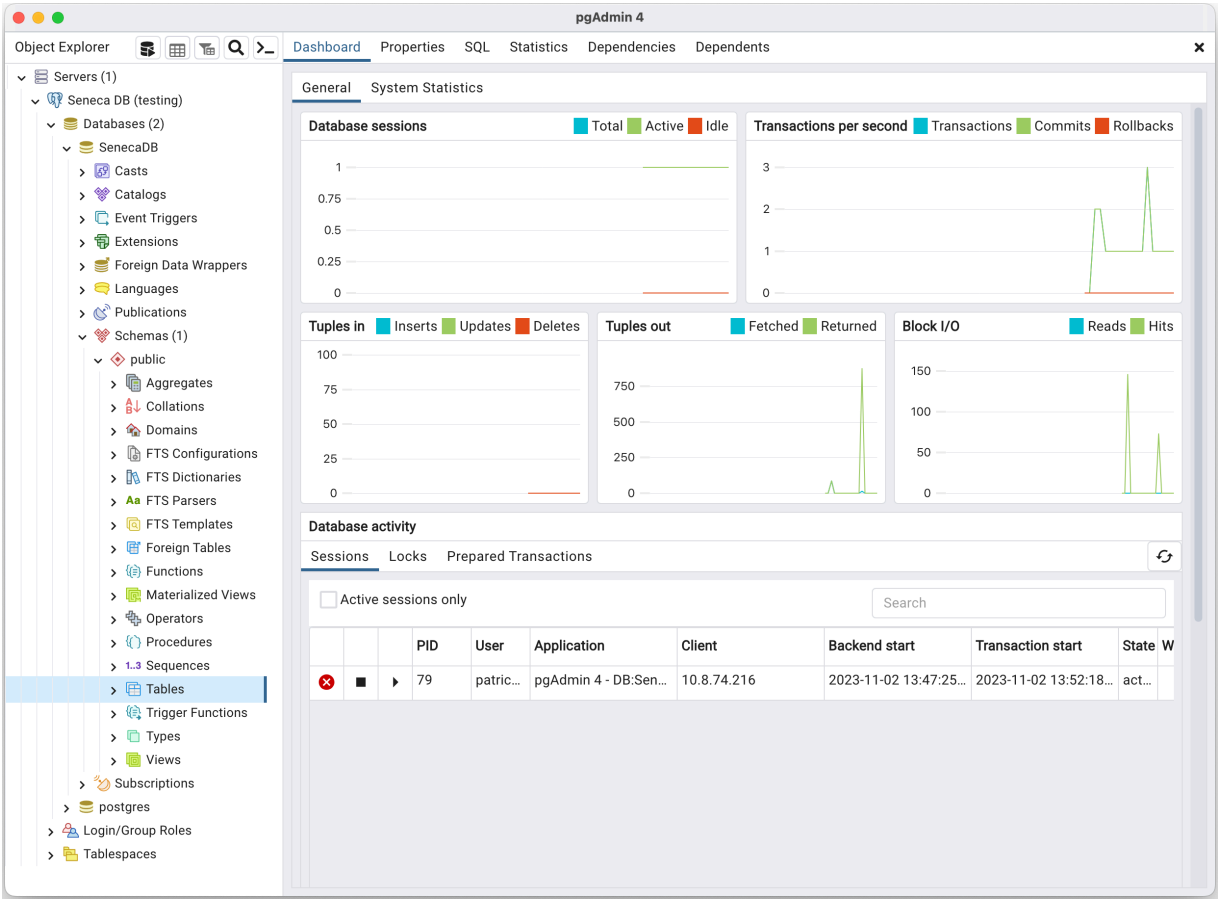
1. Right Click on the **"Servers"** icon in the left pane (Under "Browser") and select **Create > Server**

2. This will open the "Create - Server" Dialog window. Proceed to enter the following information about your Postgres Database on Neon.tech.

Field	Value
Name	This can be anything you like, ie "Test Connection"
(Connection Tab) Host	This is the server for your Neon.tech Postgres DB ("PGHOST" value), ie: **ab-cd-12345.us-east-2.aws.neon.tech**
(Connection Tab) Port	This is the port for your Neon.tech Postgres DB - it should be the same as what's already there, ie: **5432**
(Connection Tab)	Enter your "PGDATABASE" value here

Field	Value
Maintenance database	
(Connection Tab) Username	Enter your "PGUSER" value here
(Connection Tab) Password	Enter your "PGPASSWORD" value here

Once you have entered all of your information, hit the "Save" button and click "Servers" in the left pane to expand your server connections. If you entered valid information for the above fields, you should see your Neon.tech Postgres DB Connection. Expand this item and the following **"Databases (2)"** item, and you should see your database. Expand this item, as well as the nested **"Schemas (1)"** item, followed by the **"public"** item, and you should be presented with something that looks like this:



Success! We will be keeping an eye on our data using this tool, so it is wise to have it running during development.

Sequelize ORM with Postgres

Sequelize is an "ORM" tool, which stands for "Object-Relational Mapper". Using an Object-Relational Mapper enables us to interact with a relational database using object-oriented programming techniques, which abstracts away the need to write specific SQL statements. Instead, we work with regular JavaScript to interact with the data using familiar object-oriented & asynchronous programming techniques.

> Using an ORM has two benefits:
>
> - You can replace the underlying database without necessarily needing to change the code that uses it. This allows developers to optimize for the characteristics of different databases based on their usage.
> - Basic validation of data can be implemented within the framework. This makes it easier and safer to check that data is stored in the correct type of database field, has the correct format (e.g. an email address), and isn't malicious in any way (hackers can use certain patterns of code to do bad things such as deleting database records).
>
> MDN: Abstract and simplify database access

Getting Started

Fortunately, "Sequelize" is packaged as a module on NPM (see: "sequelize"). Therefore to get started, we will need to "install" it as a dependency within our project. With your application folder open in Visual Studio Code, open the

integrated terminal and enter the command

```
npm install sequelize pg pg-hstore
```

This will add both the **sequelize** and the **pg / pg-hstore** modules to our node_modules folder, as well as add their names & version numbers to our package.json file under "dependencies".

Next, we need to update our **server.js** file to use the new modules so that we can test our connection to the database. If you're working with an existing application, comment out any existing Express app code (routes, listen, etc.) that you have in server.js (for the time being) and add the following code:

```
const Sequelize = require('sequelize');

// set up sequelize to point to our postgres database
const sequelize = new Sequelize('database', 'user', 'password', {
  host: 'host',
  dialect: 'postgres',
  port: 5432,
  dialectOptions: {
    ssl: { rejectUnauthorized: false },
  },
});

sequelize
  .authenticate()
  .then(() => {
    console.log('Connection has been established successfully.');
  })
  .catch((err) => {
    console.log('Unable to connect to the database:', err);
  });
```

Where **database** is your "PGDATABASE" value, **user** is your "PGUSER" value, **password** is your "PGPASSWORD" and lastly, **host** will be your "PGHOST" url (ie: "ab-cd-12345.us-east-2.aws.neon.tech").

Once you have updated your app to use the **Sequelize** module, try running it using our usual "**node server.js**" command. If everything was entered correctly, you should see the following message in the console:

```
Executing (default): SELECT 1+1 AS result
Connection has been established successfully.
```

Finally, If you see any other errors at this point, go back and check that you have entered all of your credentials correctly when creating the sequelize object. Recall: You can use `ctrl + c` to stop a node.js application from running.

Models (Tables) Introduction

Now that we have successfully tested the connection to our Postgres database from our node.js application, we must discuss what the **Sequelize** module does and how we will be using it to manage data persistence within our Postgres Database.

As we know, sequelize is technically an Object-Relational Mapping ("ORM") framework. It maps our JavaScript objects ("models") to tables and rows within our database and will automatically execute relevant SQL commands on the database whenever data using our "models" (JavaScript objects) is updated. This saves us the trouble of manually writing complex SQL statements whenever we wish to update the back-end database to reflect changes made by the user.

To see this in action, update your server.js file to use the following code:

```
const Sequelize = require('sequelize');

// set up sequelize to point to our postgres database
const sequelize = new Sequelize('database', 'user', 'password', {
  host: 'host',
  dialect: 'postgres',
  port: 5432,
  dialectOptions: {
    ssl: { rejectUnauthorized: false },
  },
});

// Define a "Project" model

const Project = sequelize.define('Project', {
  title: Sequelize.STRING,
  description: Sequelize.TEXT,
});

// synchronize the Database with our models and automatically add
the
// table if it does not exist

sequelize.sync().then(() => {
  // create a new "Project" and add it to the database
  Project.create({
    title: 'Project1',
    description: 'First Project',
  })
    .then((project) => {
      // you can now access the newly created Project via the
variable project
      console.log('success!');
    })
```

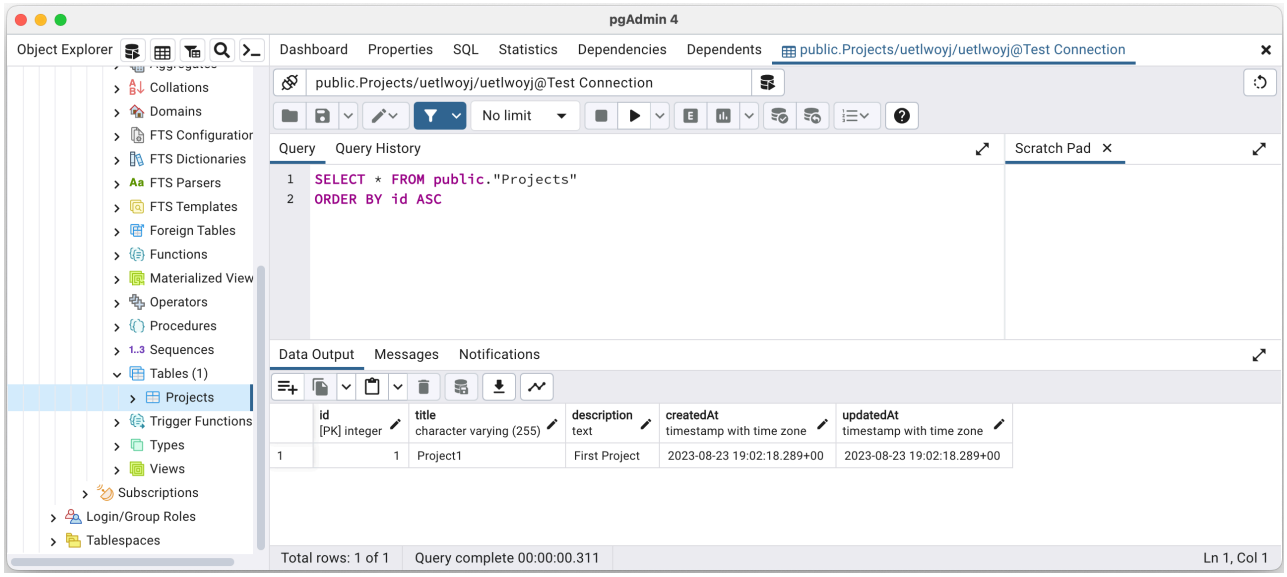
Once again, **database** is your randomly generated “User & Default database” value, **user** is also your randomly generated “User & Default database” value, **password** is your password and lastly, **host** will be your server host url.

There is a lot going on in the above code - but before we walk through what everything is doing, try updating the above code with your database credentials and run it once again with **node server.js**. You should see the something very similar to the following output:

```
Executing (default): INSERT INTO "Projects"
("id","title","description","createdAt","updatedAt") VALUES
(DEFAULT,'Project1','First Project','2017-02-28 22:45:25.163
+00:00','2017-02-28 22:45:25.163 +00:00') RETURNING \*;
success!
```

It appears that Sequelize has done some of the heavy lifting for us. To confirm that the create operation was successful and that we have indeed persisted "Project1" in a new "Projects" table, go back to your **pgAdmin** application, right-click on "**Tables**" and choose "Refresh". You should now see our new "Projects" table in the list.

To view the contents of the table, **right-click** on the **"Projects"** table and select **View / Edit Data > All Rows**. This will open a new window with a grid view that you can use to explore the data in the table:



You will notice that there are some columns in the "Project" table that we didn't define in our "Project" Model; specifically: **id**, **createdAt** and **updatedAt**; recall:

```
// Define a "Project" model

const Project = sequelize.define('Project', {
  title: Sequelize.STRING,
  description: Sequelize.TEXT,
});
```

It follows that the **title** and **description** columns are there, but where did the others come from? The addition of the extra columns are actually added by default by the **sequelize** module. Whenever we "define" a new model, we automatically get **id**, **createdAt** and **updatedAt** and when we save data using this model, our data is automatically updated to include correct values for those fields. This is extremely handy, as we didn't actually create our primary-key for the table (sequelize went ahead and made "id" our primary key). Also, the **createdAt** and **updatedAt** fields are both widely used. However, if we decide that we want to specify our own auto-incrementing

primary key and remove the createdAt and updatedAt fields, we can define our model using the following code instead:

```
// Define a "Project" model

const Project = sequelize.define(
  'Project',
  {
    project_id: {
      type: Sequelize.INTEGER,
      primaryKey: true, // use "project_id" as a primary key
      autoIncrement: true, // automatically increment the value
    },
    title: Sequelize.STRING,
    description: Sequelize.TEXT,
  },
  {
    createdAt: false, // disable createdAt
    updatedAt: false, // disable updatedAt
  }
);
```

Now that we have defined our **Project** model (either with or without the "createdAt" and "updatedAt" timestamps) we can look at the rest of the code, ie the **sync()** operation and creating **Project1** - recall:

```
// synchronize the Database with our models and automatically add
the
// table if it does not exist

sequelize.sync().then(() => {
  // create a new "Project" and add it to the database
  Project.create({
    title: 'Project1',
```

The sequelize.sync() operation needs to be completed before we can do anything else. This ensures that all of our models are represented in the database as tables. If we have defined a model in our code that doesn't correspond to a table in the database, **sequelize.sync()** will automatically create it (as we have seen).

> **NOTE:** We **do not** have to sync() the database before every operation. This is only required when the server starts to ensure that the models are correctly represented as tables within the database.

Once our models have been successfully sync()'d with the database, we can start working with the data. You will notice that we use the familiar **then()** and **catch()** functions; this is because both sync() and create() return a **promise** and as we stated above, we must work with the data **after** the sync() operation has successfully completed.

If sync() resolves successfully, we then wish to create a new record in the "Project" table, so we use **Project.create()** method and pass it some data (**title** and **description**). If the operation completed successfully, we see the message "success!" in the console - otherwise we catch the error and output "something went wrong!"

Defining Models

One of the most important things we must do when working with Sequelize is to correctly set up our models. Once the models are set up successfully, working with the data is simple. Since each model technically corresponds to a table within our database, what we are really doing is defining tables. Each column of a table within our database stores a specific **type** of data. In our previous example, we define the column **title** as a **STRING** and the column **description** as **TEXT** within a table called **Project**.

Sequelize provides definitions for a full list of types, and each column is given a type. The following is a list of the most common types:

- **Sequelize.STRING** - A variable length string. Default length 255
- **Sequelize.TEXT** - An unlimited length text column.
- **Sequelize.INTEGER** - A 32 bit integer.
- **Sequelize.FLOAT** - Floating point number (4-byte precision).
- **Sequelize.DOUBLE** - Floating point number (8-byte precision)
- **Sequelize.DATE** - A datetime column
- **Sequelize.TIME** - A time column
- **Sequelize.BOOLEAN** - A boolean column

So, if we want to define a model (table) that stores blog entries, we could use the following code:

```
// Define a "BlogEntry" model

const BlogEntry = sequelize.define('BlogEntry', {
  title: Sequelize.STRING, // entry title
  author: Sequelize.STRING, // author of the entry
  entry: Sequelize.TEXT, // main text for the entry
  views: Sequelize.INTEGER, // number of views
  postDate: Sequelize.DATE, // Date the entry was posted
});
```

> **NOTE:** It is also possible to introduce **data validation** when we define our models. For a full list of available rules and how they're implemented, see: Validators in the official documentation.

Model Relationships / Associations

In a relational database system, tables (models) can be **related** using foreign

key relationships / associations. For example, say we have a table of **Users** and a table of **Tasks**, where each User could have **1 or more** Tasks. To enforce this relationship, we would add an additional column on the Tasks table as a foreign-key to the Users table, since 1 or more Tasks could belong to a specific user. For example, "Task 1", "Task 2" and "Task 3" could all belong to "User 1", whereas "Task 4" and "Task 5" may belong to "User 2".

Using Sequelize models, we can easily define this relationship using the hasMany() / belongsTo() methods (since "User has many Task(s)"), for example:

```
// Define our "User" and "Task" models

const User = sequelize.define('User', {
  fullName: Sequelize.STRING, // the user's full name (ie: "Jason Bourne")
  title: Sequelize.STRING, // the user's title within the project (ie, developer)
});

const Task = sequelize.define('Task', {
  title: Sequelize.STRING, // title of the task
  description: Sequelize.TEXT, // main text for the task
});

// Associate Tasks with user & automatically create a foreign key
// relationship on "Task" via an automatically generated "UserId" field

User.hasMany(Task);
Task.belongsTo(User);
```

If we wish to create a User and then assign him some tasks, we can "create" the tasks immediately after the user is created, ie:

```
sequelize.sync().then(() => {
  // Create user "Jason Bourne"
  User.create({
    fullName: 'Jason Bourne',
    title: 'developer',
  }).then((user) => {
    console.log('user created');

    // Create "Task 1" for the new user
    Task.create({
      title: 'Task 1',
      description: 'Task 1 description',
      UserId: user.id, // set the correct Userid foreign key
    }).then(() => {
      console.log('Task 1 created');
    });

    // Create "Task 2" for the new user
    Task.create({
      title: 'Task 2',
      description: 'Task 2 description',
      UserId: user.id, // set the correct Userid foreign key
    }).then(() => {
      console.log('Task 2 created');
    });
  });
});
```

Next, try running this code and take a look at your database in pgAdmin. You should see that two new tables, **"Users"** and **"Tasks"** have been created, with **"Jason Bourne"** inside the "User" table and **"Task 1"** and **"Task 2"** inside the "Task" table. The two new tasks will both have a **UserId** matching "Jason Bourne"'s id. We have achieved the one-to-many relationship between this user and his tasks.

> **NOTE:** Sequelize also supports other types of relationships using:
>
> - hasOne()
> - belongsToMany()
>
> For more information, refer to "Associations" in the official documentation.

Operations (CRUD) Reference

The four major operations that are typically performed on data are **C**reate, **R**ead, **U**pdate and **D**elete (**CRUD**). Using these four operations, we can effectively work with the data in our database. Assume we have a simple **Name** model defined:

```
// Define a "Name" model

const Name = sequelize.define('Name', {
  fName: Sequelize.STRING, // first Name
  lName: Sequelize.STRING, // Last Name
});
```

We can use the following code to **C**reate new names, **R**ead a list of names, **U**pdate a specific name and lastly **D**elete a name from the "Name" table in our database

Create

To **create** new names in our **Name** table, we can use the following code:

```
sequelize.sync().then(() => {
  Name.create({
    fName: 'Kyler',
    lName: 'Odin',
  }).then(() => {
```

In the above code we create three new objects following the fields defined in our "Name" model. Since our "Name" model is synchronized with the database, this adds three new records - each with their own unique "id" value, as well as "createdAt" and "updatedAt" values for the implicit primary key and timestamp columns. The create function automatically persists the new object to the database and since it also returns a promise, we can execute code after the operation is complete. In this case we simply output the name to the console.

Read

To **read** entries from our **Name** table, we can use the following code:

```
sequelize.sync().then(() => {
  // return all first names only
  Name.findAll({
    attributes: ['fName'],
  }).then((data) => {
    console.log('All first names');
    for (let i = 0; i < data.length; i++) {
      console.log(data[i].fName);
    }
  });

  // return all first names where id == 2
  Name.findAll({
    attributes: ['fName'],
    where: {
      id: 2,
    },
  }).then((data) => {
    console.log('All first names where id == 2');
    for (let i = 0; i < data.length; i++) {
      console.log(data[i].fName);
```

Here, we are once again using a reference to our "Name" model. This time we are using it to fetch data from the "Name" table using the findAll() method. This method takes a number of configuration options in it's object parameter, such as **attributes**, which allows you to limit the columns that are returned (in this case we only want 'fName') and a **where** parameter that enables us to specify conditions that the data must meet to be returned. In the above example, **id** must have a value of **2**.

> **NOTE:** It is important to note that trying to log a model instance directly to console.log (ie: `console.log(data[i])`) will produce a lot of clutter, since Sequelize instances have a lot of things attached to them. Instead, you can use the **.toJSON()** method (which automatically guarantees the instances to be JSON.stringify-ed well). See sequelize.org - logging instances for more information.

We can also specify an order that the returned data should be in, ie:

```
sequelize.sync().then(() => {
  // return all first names only
  Name.findAll({ order: ['fName'] }).then((data) => {
    console.log('All data');
    for (let i = 0; i < data.length; i++) {
      console.log(data[i].fName);
    }
  });
});
```

> **NOTE**: See the documentation for advanced queries and fetching associated elements with the "include" option when using Model Relationships / Associations (ie: `Task.findAll({include:[User]})`)

Update

To **update** existing names in our **Name** table, we can use the following code:

```
sequelize.sync().then(() => {
  // update User 2's last name to "James"
  // NOTE: this also updates the "updatedAt field"
  Name.update(
    {
      lName: 'James',
    },
    {
      where: { id: 2 }, // only update user with id == 2
    }
  ).then(() => {
    console.log('successfully updated user 2');
  });
});
```

In order to "update" a record in the "Name" table, we make use of the update method. This method takes two parameters: an object that contains all of the properties and (updated) values for a record, and a second object that is used to specify options for the update - most importantly, the **"where"** property. The "where" property contains an object that is used to specify exactly *which* record should be updated. In this case, it is the row that has an **id** value of **2**.

Delete

To **delete** existing names in our **Name** table, we can use the following code:

```
sequelize.sync().then(() => {
  // remove User 3 from the database
  Name.destroy({
    where: { id: 3 }, // only remove user with id == 3
  }).then(() => {
    console.log('successfully removed user 3');
  });
});
```

The delete functionality is actually achieved via a method called destroy. In this case, we invoke the **destroy** method on the model that contains the record that we wish to remove (ie, "Name"). It takes a single options object as it's only parameter and like the **update** function, the most important option is the **"where"** property. The "where" property contains an object that is used to specify exactly *which* record should be removed. In this case, it is the row that has an **id** value of **3**.

Example Code

You may download the sample code for this topic here:

> Relational-Database-Postgres

JavaScript Object Notation (JSON)

JSON ("JavaScript Object Notation") is a plain-text format that easily converts to a JavaScript object in memory. Essentially, JSON is a way to define an object using "Object Literal" notation, **outside** your application. Using the native JavaScript built-in JSON Object, we can preform the conversion from plain-text (JSON) to JavaScript Object (and vice-versa) easily. For example:

Converting JSON to an Object

```
let myJSONStr =
'{"users":[{"userId":1,"fName":"Joe","lName":"Smith"},{"userId":2,"fName":"Jeffrey","lName":"Sherman"},{"userId":3,"fName":"Shantell","

// Convert to An Object:
let myObj = JSON.parse(myJSONStr);

// Access the 3rd user (Shantell McLeod)
console.log(myObj.users[2].fName); // Shantell
```

Converting an Object to JSON

```
let myObj = {
  users: [
    { userId: 1, fName: 'Joe', lName: 'Smith' },
    { userId: 2, fName: 'Jeffrey', lName: 'Sherman' },
    { userId: 3, fName: 'Shantell', lName: 'McLeod' },
  ],
};

let myJSON = JSON.stringify(myObj);

console.log(myJSON); // Outputs:
'{"users":[{"userId":1,"fName":"Joe","lName":"Smith"},{"userId":2,"fName":"Jeffrey","lName":"Sherman"},{"userId":3,"fName":"Shantell","
```

Caveats When Using JSON

The JSON format works exceptionally well to "serialize" (convert an object in memory to a byte / string representation) and "deserialize" (converting back to an object in memory). However, there are certain things that cannot be encoded to JavaScript Object Notation:

Object Instances

Instances of objects in memory cannot be stored in a JSON format. For example, consider the following "product" object:

```
let product = {
  name: 'Pencil',
  price: 3.95,
  added: new Date('December 17, 1995 03:24:00'),
};
```

Since the "added" property is an instance of the Date object, we can invoke methods such as "toLocalStimeString()":

```
console.log(product.added.toLocaleTimeString('fr-CA')); // 03 h 24 min 00 s
```

However, if we convert the product to JSON and back, we lose this ability:

```
// convert to JSON
let productJSON = JSON.stringify(product);

// restore (convert to object)
let productFromJSON = JSON.parse(productJSON);

console.log(productFromJSON.added.toLocaleTimeString('fr-CA')); // TypeError:
productFromJSON.added.toLocaleTimeString is not a function
```

This issue occurs because during the conversion to JSON, the Date object was implicitly converted to a string:

```
{
  "name": "Pencil",
  "price": 3.95,
  "added": "1995-12-17T08:24:00.000Z"
}
```

Functions (Methods)

Functions ("methods") that exist on the object also will not convert to JSON. For example:

```
let counter = {
  current: 0,
  increase: function () {
    this.current++;
  },
};

console.log(counter.current); // 0
counter.increase();
console.log(counter.current); // 1
```

Once again, if we attempt to convert this object to JSON and back, we lose the "increase()" function:

```
// convert to JSON
let counterJSON = JSON.stringify(counter);

// restore (convert to object)
let counterFromJSON = JSON.parse(counterJSON);

console.log(counterFromJSON.current); // 0
counterFromJSON.increase(); // TypeError: counterFromJSON.increase is not a function
```

In this case, this issue occurs because during the conversion to JSON, the "increase" function was not included:

```
{ "current": 0 }
```

> **NOTE:** For more information on how values are "stringified", refer to the MDN documentation on "JSON.stringify()"

AJAX Introduction

AJAX (Asynchronous JavaScript and XML) can be described as a collection of technologies used together to create a richer user experience by enabling data to be transferred between a web client (browser) and web server without the need to refresh the page:

> Ajax, is not a technology in itself, but rather an approach to using a number of existing technologies together, including HTML or XHTML, CSS, JavaScript, DOM, XML, XSLT, and most importantly the XMLHttpRequest object. When these technologies are combined in the Ajax model, web applications are able to make quick, incremental updates to the user interface without reloading the entire browser page. This makes the application faster and more responsive to user actions. Ajax's most appealing characteristic is its "asynchronous" nature, which means it can communicate with the server, exchange data, and update the page without having to refresh the page.
>
> Although X in Ajax stands for XML, JSON is preferred because it is lighter in size and is written in JavaScript. Both JSON and XML are used for packaging information in the Ajax model.
>
> https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX

The Fetch API

In modern browsers, we can use the "Fetch API" to make AJAX requests. Essentially, we can configure a **new Request** by providing two parameters:

- The location of the resource

- A set of "options", (defined using "object literal" notation)

The "location" parameter is simply the URI of the resource, ie: "https://reqres.in/api/users/", while the "options" parameter could contain any number of options, including:

- The http method, ie: 'POST'
- The 'body' of the request, ie: 'JSON.stringify({user:"John Doe", job:"unknown"})'
- An object consisting of a number of headers, ie: '{"Content-Type": "application/json"}'
- And Many Others

In practice, this would look something like this:

```
let myRequest = new Request('https://reqres.in/api/users/', {
  method: 'POST',
  body: JSON.stringify({ user: 'John Doe', job: 'unknown' }),
  headers: {
    'Content-Type': 'application/json',
  },
});
```

Once the request is configured, we can "Fetch" the data using "fetch()" with our request. This "fetch" method will return a promise that resolves with a "response" object that has a number of methods, including:

- **response.text()** - which we can use to read the 'response' stream. This method returns a promise that will resolve with text.
- **response.json()** - which we can use to read the 'response' stream. This method returns a promise that will resolve with an object.

To execute the request defined above (ie: myRequest), we can wire up the "fetch" using the following code (assuming that our resource is returning JSON-formatted data).

```
fetch(myRequest)
  .then((response) => {
    return response.json();
  })
  .then((json) => {
    console.log(json); // here is the parsed JSON response
  });
```

AJAX: The Fetch API (Compressed)

To save lines and make your code more readable and concise, the above two pieces of code can be combined, ie:

```
fetch('https://reqres.in/api/users/', {
  method: 'POST',
  body: JSON.stringify({ user: 'John Doe', job: 'unknown' }),
  headers: { 'Content-Type': 'application/json' },
})
  .then((response) => response.json())
  .then((json) => {
    console.log(json);
  });
```

NOTE: Our code is even shorter if we're simply doing a "GET" request, ie:

```
fetch('https://reqres.in/api/users/')
  .then((response) => response.json())
  .then((json) => {
```

Handling Responses with an "Error" Status

If we wish to handle a situation where the fetch fails, we can always add a catch statement at the end of the above code. However, it is important to note that if the response itself was successful (ie a connnection was made and a response was returned), then the "catch" callback code will not be executed *even if* the response status code indicates an error, ie 500 or 404. To handle these situations, we can leverage a method on the response object callded "ok" (see: response.ok) which will be true if the status code of the response was in the **200 range**. Practically speaking, it can be used like this:

```
fetch('https://reqres.in/api/unknown/23')
  .then((response) => {
    // return a rejected promise with the status code of the
response if it wasn't "ok"
    return response.ok ? response.json() :
Promise.reject(response.status);
  })
  .then((json) => {
    console.log(json);
  })
  .catch((err) => {
    console.log(err);
  });
```

API Introduction & Implementation

You may have heard of the term **REST** or **RESTful** API when reading about Web Programming. For our purposes, this can be described as way to use the HTTP protocol (ie, "GET", "POST", "PUT", "DELETE", etc.) with a standard message format (ie, JSON or XML) to preform CRUD operations (**C**reate, **R**ead, **U**pdate, **D**elete) on a data source.

> **NOTE:** To truly create a *fully compliant* **REST API** we must conform to the standards outlined in Roy Fielding's PhD dissertation, Architectural Styles and the Design of Network-based Software Architectures. The design pattern that we are using here could more appropriately be called a "Web API".

What makes this architecture so valuable, is that we remove any assumptions about how a client will access the data. A client could make HTTP requests to the API from a website, mobile app, etc. and it would be the website or app's job to correctly render the data once it's received. This simplifies development of front-end applications that use the data and even removes any specific programming language requirements for the client. If it can handle HTTP requests / responses and JSON, it can use our data.

Route Configuration

Before we think about getting any kind of persistent storage involved however, let's first see how we can configure all of our routes in our server to allow for CRUD operations on a simple collection of users in the format

```
{userId: number, fName: string, lName: string}
```

Route	HTTP Method	Description
/api/users	GET	Get all the users
/api/users	POST	Create a user
/api/users/:userId	GET	Get a single user
/api/users/:userId	PUT	Update a user with new information
/api/users/:userId	DELETE	Delete a user

When these routes are applied to our Express server code, we get something that looks like this:

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

const HTTP_PORT = process.env.PORT || 8080;

app.get('/api/users', (req, res) => {
  res.send({ message: 'fetch all users' });
});

app.post('/api/users', (req, res) => {
  res.send({ message: 'add a user' });
});

app.get('/api/users/:userId', (req, res) => {
```

Here, we have made use of the request object's params method to identify the specific user that needs to be fetched, updated or deleted based on the URL alone. In a sense, what we're allowing here is for the URL + HTTP Method to act as a way of querying the data source, as **/api/users/3**, **/api/users/4923** or even **/api/users/twelve** will all be accepted. They may not necessarily return valid data, but the routes will be found by our server and we can attempt to preform the requested operation.

AJAX Testing (View)

Now that we have all of the routes for our API in place, let's create a "view" that will make AJAX requests to test our API functionality. To begin, create a **views** folder and add the file **index.html**. This will be a simple HTML page consisting of 5 buttons (each corresponding to a piece of functionality in our API) and some simple JavaScript to make an AJAX request.

However, since we are serving this file from the same server that our API is on, we will need to add some additional code to our server file; specifically:

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

and

```
app.get('/', (req, res) => {
  res.sendFile(path.join(__dirname, '/views/index.html'));
});
```

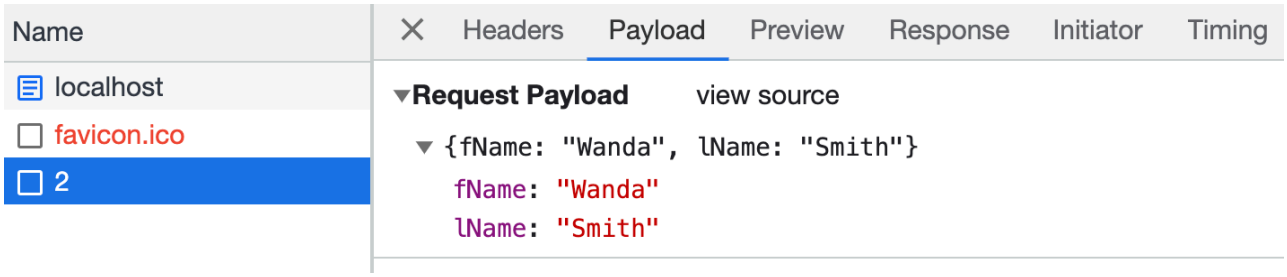
Finally - our server is setup and ready to serve the index.html file at our main route ("/"). Our next step is to add our client-side logic / JS to the index.html file. Here, we hard-code some requests to the API and output their results to

the web console to make sure they function correctly:

```
<!doctype html>
<html>
  <head>
      <title>API Test</title>
      <script>
          function makeAJAXRequest(method, url, body) {
              fetch(url, {
                  method: method,
                  body: JSON.stringify(body), // if missing
'body', 'undefined' is returned
                  headers: { 'Content-Type': 'application/json' }
              })
              .then(response => response.json())
              .then(json => {
                  console.log(json);
              });
          }
      </script>
  </head>
  <body>
      <p>Test the API by outputting to the browser console:</p>
      <!-- Get All Users -->
      <button type="button" onclick='makeAJAXRequest("GET", "/api/
users")'>Get All Users</button><br /><br />
      <!-- Add New User -->
      <button type="button" onclick='makeAJAXRequest("POST",
"/api/users", {fName: "Bob", lName: "Jones"})'>Add New
User</button><br /><br />
      <!-- Get User By Id -->
      <button type="button" onclick='makeAJAXRequest("GET", "/api/
users/2")'>Get User</button><br /><br />
      <!-- Update User By Id -->
      <button type="button" onclick='makeAJAXRequest("PUT", "/api/
users/2", {fName: "Wanda", lName: "Smith"})'>Update
```

Adding Data (JSON)

Once you have entered the above code, save the changes and try running the server locally - you will see that All of the routes tested return a JSON formatted message. This confirms that our Web API will correctly respond to AJAX requests made by the client. Additionally, If you open the **Network tab** (Google Chrome) before initiating one of the calls to **Update** or **Add a New User**, you will see that our request is also carrying a payload of information, ie:



If we wish to capture this information in our routes (so that we can make the appropriate updates to our data source), we must make some small modifications to our server.js file and individual routes (ie: POST to "/api/users" & PUT to "/api/users/:userId"). The first thing that we must do is incorporate middleware to parse the incoming data, ie:

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

This should allow our routes to access data passed to our API using the req.body property. More specifically, we can update our POST & PUT routes to use req.body to fetch the new / updated **fName** and **lName** properties:

```
app.post('/api/users', (req, res) => {
  res.send({ message: `add the user: ${req.body.fName}
```

and

```
app.put('/api/users/:userId', (req, res) => {
  res.send({ message: `update User with Id: ${req.params.userId}
to ${req.body.fName} ${req.body.lName}` });
});
```

If we try running the server to test the API again, we will see that the messages returned back from the server correctly echo the data sent to the API. We now have everything that we need to preform simple CRUD operations via AJAX on a data source using a web service. The only thing missing is the data store itself.

> **NOTE:** If we want to allow the API to respond to requests from *outside* the domain (this is what https://reqres.in does), we will have to enable Cross-Origin Resource Sharing (CORS) - see the third-party CORS middleware

Example Code

You may download the sample code for this topic here:

> Web-API-Overview

HTTPS Introduction

HTTPS is HTTP communication between a web browser and a server over a secure, encrypted connection, using TLS (Transport Layer Security). The primary purpose for using HTTPS is to enable users to verify that a website that transfers sensitive data, can do so in a secure and safe manner. HTTPS uses SSL/TLS certificates on the server to encrypt the communication between the client and server so that packets in transmission cannot be intercepted and used to either steal or forge information. The concept of capturing packets in the middle of transmission between client and server or vice versa, is called a man in the middle attack.

Digital Certificates

HTTPS uses a protocol known as "TLS" (formerly "SSL" or "Secure Sockets Layer") to enable secure communication across a network in order to prevent tampering / eavesdropping. This is achieved through the use of something called a "digital certificate":

> Digital Certificates have a key pair: a public and a private key. These keys work together to establish an encrypted connection. The certificate also contains what is called the “subject,” which is the identity of the certificate/website owner.
>
> The most important part of a certificate is that it is digitally signed by a trusted CA ("Certificate Authority"), like DigiCert. Anyone can create a certificate, but browsers only trust certificates that come from an organization on their list of trusted CAs. Browsers come with a pre-installed list of trusted CAs, known as the Trusted Root CA store. In order to be added to the Trusted Root CA store and thus become a Certificate

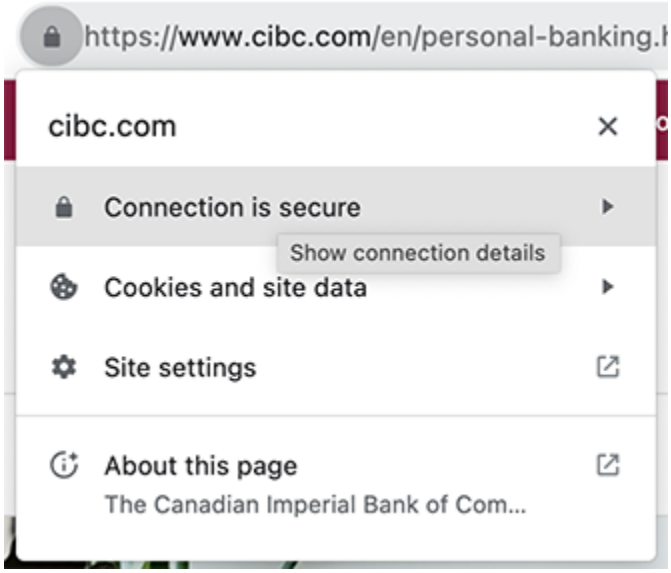
> Authority, a company must comply with and be audited against security and authentication standards established by the browsers.
>
> https://www.digicert.com

Essentially, for a website / app to use HTTPS, a certificate from a trusted source (such as "Digicert") is required. This certificate contains a "digital signature", signed by the Certificate Authority (ie: "Digicert") which proves the validity of the certificate and the website. It also contains a public / private key pair, enabling messages to be encrypted using a public key, but only read using the corresponding private key. Encrypting messages using a trusted website's public key is the first step to enabling two way encrypted communication:

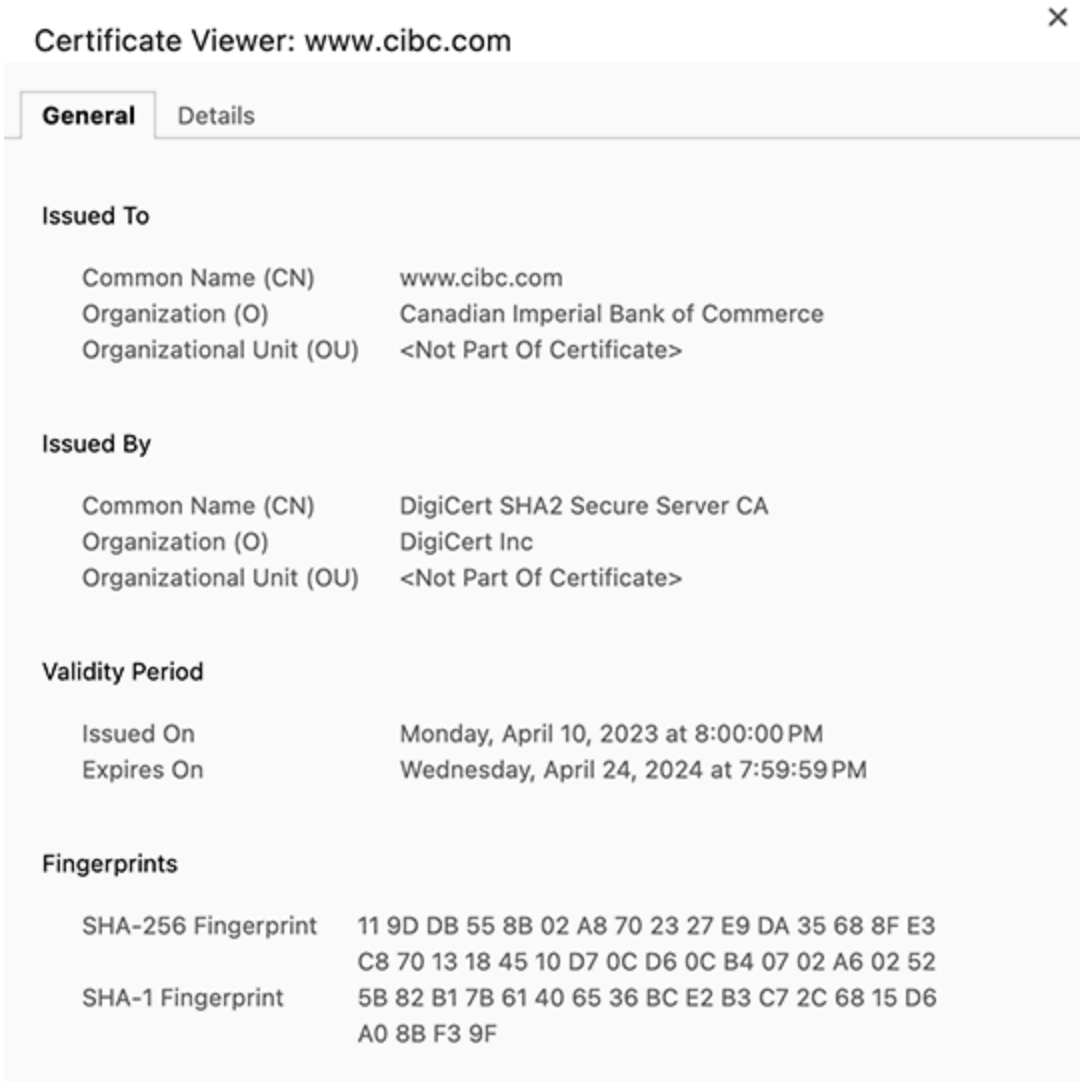
> 1. When a web browser (or client) directs to a secured website, the website server shares its TLS/SSL certificate and its public key with the client to establish a secure connection and a unique session key.
>
> 2. The browser confirms that it recognizes and trusts the issuer, or Certificate Authority, of the SSL certificate—in this case DigiCert. The browser also checks to ensure the TLS/SSL certificate is unexpired, unrevoked, and that it can be trusted.
>
> 3. The browser sends back a symmetric session key and the server decrypts the symmetric session key using its private key. The server then sends back an acknowledgement encrypted with the session key to start the encrypted session.
>
> 4. Server and browser now encrypt all transmitted data with the session key. They begin a secure session that protects message privacy, message integrity, and server security.

Viewing Certificates

Information about a website's digital certificate can be easily viewed in a modern web browser. Typically, to the left of the URL bar, you will find a "lock" icon. Click on it to view information about your connection with this website (screenshot taken in Chrome).



Notice how it shows that the site is using a secure connection with an option to "Show Connection Details". Clicking this allows us to confirm that the certificate is indeed valid and was issued by "DigiCert Inc".



We may also switch to the **"Details"** pane, which provides information about the certificate, such as the issuer, expiration date, and the encryption algorithms used.

With this information, we can confirm that sending login credentials and retrieving banking information from CIBC is achieved using encrypted packets between the web browser and server. Anyone who might capture them in transit would not be able to obtain any useful information.

Self Signed Certificates

SSL/TLS certificates can be created on your own and technically they can be used, however it is important to note that these certificates should not be used in a production environment. This is because using your own "self signed" certificates will result in a warning from the browser that your website is using an "untrusted" certificate, since it did not come from a trusted CA.

Creating Self Signed Certificates (Development)

When testing HTTPS locally and during development, it is common to use a self signed certificate. We can generate them in the terminal using the following command:

```
openssl req -new -x509 -nodes -out server.crt -keyout server.key
```

This will initiate the following prompts for information about the organization the certificate will be issued to. The only important one for now is the Common Name - this must be **localhost** (ie: the domain the certificate will be valid for), since we will be running our server locally:

```
Generating a 2048 bit RSA private key
..............................+++
...+++
writing new private key to 'server.key'
-----
You are about to be asked to enter information that will be
incorporated
```

This should generate two files: **"server.crt"** and **"server.key:**

Using SSL Certificates

Now that we have the required files (ie: "server.crt" and "server.key"), we can begin to configure our "server.js" code to start listening for both HTTP and HTTPS connections:

```
const fs = require('fs');
const http = require('http');
const https = require('https');
const express = require('express');
const app = express();

const HTTP_PORT = process.env.PORT || 8080;
const HTTPS_PORT = 4433;

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

// read in the contents of the HTTPS certificate and key

const https_options = {
  key: fs.readFileSync(__dirname + '/server.key'),
  cert: fs.readFileSync(__dirname + '/server.crt'),
};

// listen on ports HTTP_PORT and HTTPS_PORT.

http.createServer(app).listen(HTTP_PORT, () => {
  console.log(`http server listening on: ${HTTP_PORT}`);
});
https.createServer(https_options, app).listen(HTTPS_PORT, () => {
```

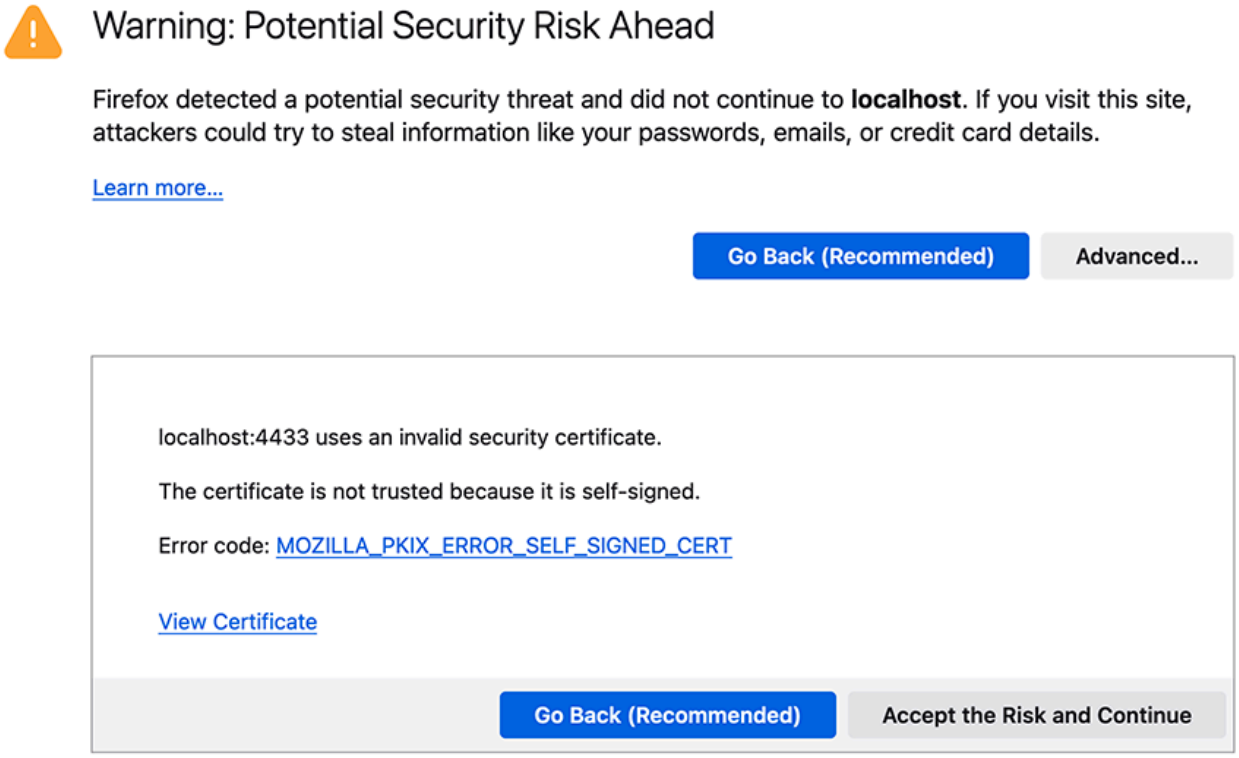
You will notice that a few key changes have been made to our usual "simple web server". Primarily:

- We import both the "http" and "https" modules, as well as the "fs" module (to read the .crt and .key files)
- Use createServer() method for both "http" and "https", making sure to provide the values for both the **key** and **cert** for the "https_options" parameter when using "https"

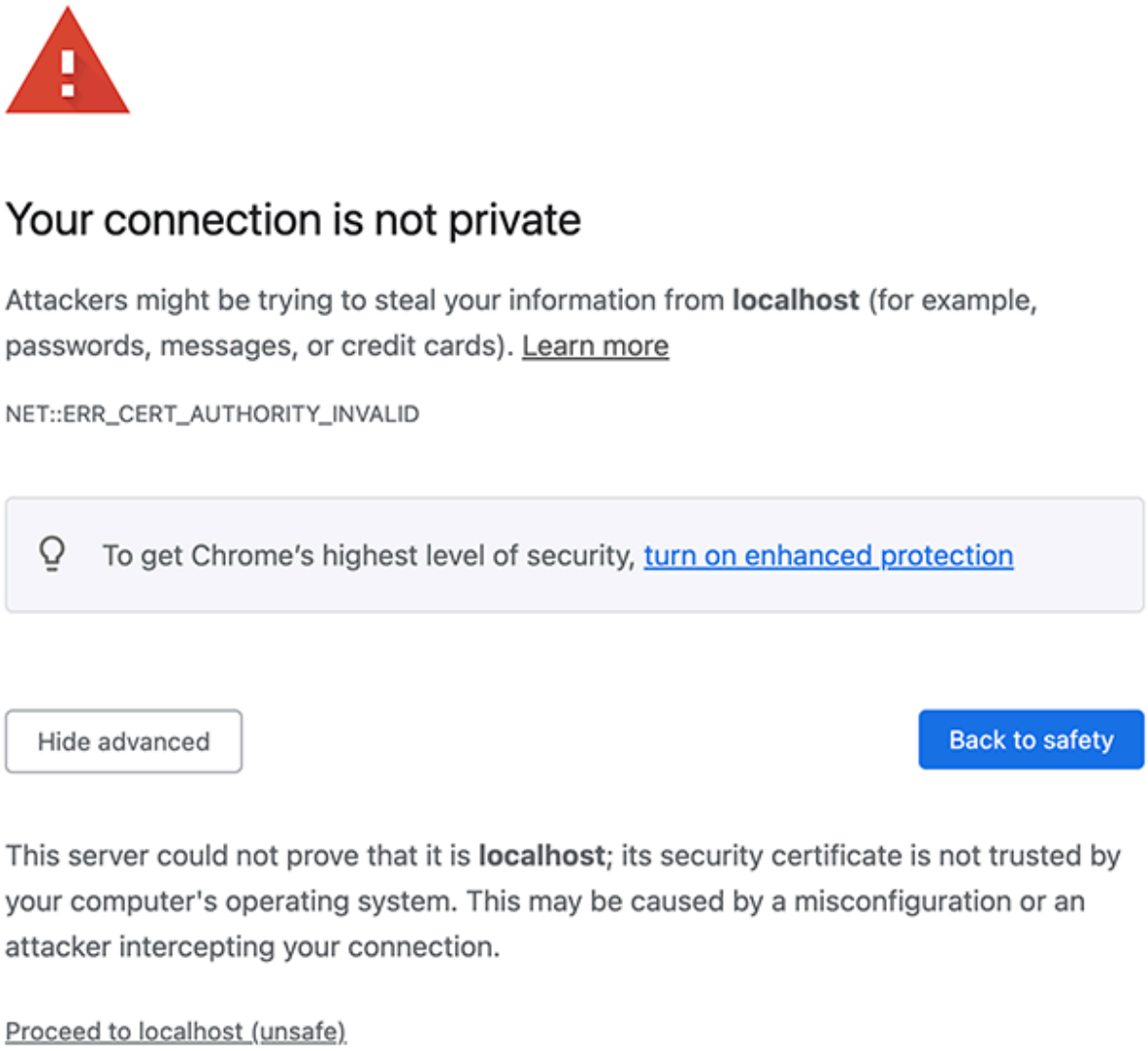
Finally, start the server and navigate to: https://localhost:4433

Depending on your web browser, you may observe a security warning if the system is functioning correctly. If you get this warning (with "Advanced" selected) everything is working as intended so far.

Warning in Firefox



Warning in Chrome



> **NOTE:** If you do not see the option to "Proceed to localhost", then typing "**thisisunsafe**" will allow you to proceed.

Accept the warnings to add an exemption and proceed to the page.

Password Encryption

HTTPS is a significant factor to consider when securing a website online; however, it does not encompass all aspects of security. For example, what if unauthorized users gain access your database? This could result in the theft of personal information such as credit card information or user passwords.

This is where the integration of an encryption library becomes important. One option to solve the above problem is to enable "one-way" encryption, effectively encrypting plain text data in a way that makes it impossible decipher. To verify if the encrypted data corresponds to specific plain text data, the plain text data must be encrypted using the original method and compared.

This is a standard way to store and work with passwords. Encrypt them in the database when a user registers and when they try to login, encrypt them once again and compare the encrypted passwords for a match. This way we are never storing users' plain text passwords in the database and anyone who has access to the database cannot read them.

Bcrypt

A famous encryption algorithm to achieve "one-way" encryption, is "bcrypt"

> bcrypt is a password-hashing function designed by Niels Provos and David Mazières, based on the Blowfish cipher and presented at USENIX in 1999. Besides incorporating a salt to protect against rainbow table attacks, bcrypt is an adaptive function: over time, the iteration count can be increased to make it slower, so it remains resistant to brute-force search attacks even with increasing computation power.

> https://en.wikipedia.org/wiki/Bcrypt

This sounds like exactly what we need. Fortunately, bcrypt is available on NPM via a module called: "bcrypt.js".

```
npm install bcryptjs
```

```
const bcrypt = require('bcryptjs');
```

Encrypting Passwords

If we wish to encrypt a plain text password (ie: "myPassword123"), we can use bcrypt to generate a "salt" and "hash" the text:

```
// Encrypt the plain text: "myPassword123"
bcrypt
  .hash('myPassword123', 10)
  .then((hash) => {
    // Hash the password using a Salt that was generated using 10
rounds
    // TODO: Store the resulting "hash" value in the DB
  })
  .catch((err) => {
    console.log(err); // Show any errors that occurred during the
process
  });
```

Validating Encrypted Passwords

If we wish to compare the "hashed" text with plain text (to see if a user-entered password matches the value in the DB), we use:

```
// Pull the password "hash" value from the DB and compare it to
"myPassword123" (match)
bcrypt.compare('myPassword123', hash).then((result) => {
  // result === true
});

// Pull the password "hash" value from the DB and compare it to
"myPasswordABC" (does not match)
bcrypt.compare('myPasswordABC', hash).then((result) => {
  // result === false
});
```

Secure HTTP Headers

When attempting to secure our websites / apps, we have seen how to implement important features such as "HTTPS" and "Password Encryption". However, there are other attacks such as "Cross-Site Scripting (XSS)", "Cross-Site Request Forgery (CSRF)", "Clickjacking Attacks", and so on that we must also consider. Fortunately, we can set a number of **headers** on our HTTP Responses that can help mitigate these issues, for example:

- **Content Security Policy**: This header can be used to control what resources the user agent is allowed to load for that page. For example, a page that uploads and displays images could allow images from anywhere, but restrict a form action to a specific endpoint. A properly designed Content Security Policy helps protect a page against a cross-site scripting attack.

- **X-Frame-Options**: Tells the browser whether the website can be embedded in a frame or iframe. By setting the X-Frame-Options header to "DENY" or "SAMEORIGIN," we prevent the web application from being embedded in a frame from another domain, effectively mitigating clickjacking attacks.

- **X-Permitted-Cross-Domain-Policies**: This header is used to limit which data external resources, such as PDF documents, can access on the domain. Failure to set the X-Permitted- Cross-Domain-Policies header to “none” value allows other domains to embed the application’s data in their content.

and so on - see https://owasp.org/www-project-secure-headers for more information.

Introducing Helmet.js

To help us work with these secure headers, we can use an NPM module called "helmet.js". Helmet.js functions as middleware in our Node / Express.js applications that automatically sets or removes certain **response headers** in an effort to enhance security.

To get started using helmet, we must install it from NPM and **require** it in our server.js code:

```
npm install helmet
```

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

Once it is required, we can use the *default configuration* by simply invoking it an "app.use()" to register it as middleware, ie:

```
app.use(helmet());
```

If you test an express server (ie: our "simple web server") with this configuration, you should see a similar set of headers have been automatically added to the response:

Response Header	Value
Content-	default-src 'self';base-uri 'self';font-src 'self' https: data:;form-

Response Header	Value
Security-Policy	action 'self';frame-ancestors 'self';img-src 'self' data:;object-src 'none';script-src 'self';script-src-attr 'none';style-src 'self' https: 'unsafe-inline';upgrade-insecure-requests
Cross-Origin-Opener-Policy	same-origin
Cross-Origin-Resource-Policy	same-origin
Origin-Agent-Cluster	?1
Referrer-Policy	no-referrer
X-Content-Type-Options	nosniff
X-Dns-Prefetch-	off

Response Header	Value
Control	
X-Download-Options	noopen
X-Frame-Options	SAMEORIGIN
X-Permitted-Cross-Domain-Policies	none
X-Xss-Protection	0

Additionally, the `X-Powered-By` header has also been removed.

For configuration options, see the "official Helmet.js documentation"

Example Code

You may download the sample code for this topic here:

> Security-Considerations

Getting Started with Vercel

The main server environment that we will be using in this course is Vercel

> "Vercel is a platform for developers that provides the tools, workflows, and infrastructure you need to build and deploy your web apps faster, without the need for additional configuration.
>
> Vercel supports popular frontend frameworks out-of-the-box, and its scalable, secure infrastructure is globally distributed to serve content from data centers near your users for optimal speeds."
>
> https://vercel.com/docs/getting-started-with-vercel.

Essentially, Vercel manages the hardware infrastructure and deployment tasks for our node.js applications in a remote environment. Apps deployed using Vercel are hosted on Vercel's infrastructure, which utilizes AWS services such as AWS Lambda, S3, CloudFront, and DynamoDB, among others

To get started, developers push their code to GitHub and Vercel does the rest. Additionally, Vercel provides a range of projects as "templates". These can be used to get started quickly or can be used as reference implementations to see how a particular framework can be deployed effectively.

The best thing - **getting started is free!** - This is where we come in:

Required Software

- By now, you should have **Node.js** (available here) and **Visual Studio Code** (available here). However we will also need git
- To download git, proceed to this download page and download git for your operating system.
- Proceed to install git with the default settings. Once this is complete, you can verify that it is installed correctly by opening a command prompt / terminal and issuing the command **git --version**. This should output something like: git version 2.37.2 (...). If it does not output the installed version of git, then something is wrong and it is not installed correctly.
- Lastly, for Vercel to gain access to our code, we must eventually place it on GitHub. Therefore, you must also have account on GitHub before proceeding.

Configuring your App for Vercel

Before we can start working with Vercel, we must make a few key changes to our code to ensure that it can be successfully deployed on Vercel. These include:

Adding a "vercel.json" file.

For our applications (defined in a "server.js" file), we must add the following **"vercel.json"** file to the root of our project:

```
{
  "version": 2,
  "builds": [
    {
      "src": "server.js",
      "use": "@vercel/node",
      "config": { "includeFiles": ["dist/**"] }
    }
  ],
  "routes": [
    {
      "src": "/(.*)",
      "dest": "server.js"
    }
  ]
}
```

Setting the "views" Application Setting

If you are using a template engine in your application (ie: EJS), then you will need to add the line:

```
app.set('views', __dirname + '/views');
```

before your route definitions.

Updating your "express.static()" Middleware

Similarly, if you are using the "express.static()" middleware to define a "public" folder, you must also include the "__dirname" in your path, ie:

```
app.use(express.static(__dirname + '/public'));
```

Explicitly Requiring the "pg" Module

If you are using Sequelize with the "pg" / "pg-hstore" modules, Vercel will give you an error if you do not explicitly require the "pg" module, ie:

```
require('pg'); // explicitly require the "pg" module
const Sequelize = require('sequelize');
```

Committing Your Code

Once you have configured your code for Vercel and you are ready to publish it, the next steps are to initialize a Git repository at the root of your project folder and push your code to GitHub:

1. First, issue the following command from the integrated terminal at the "root" folder of your project: **git init** - this will initialize a local git repository in your helloworld folder.

2. Next, create a file called **.gitignore** containing the text:

   ```
   node_modules
   ```

   This will ensure that the node_modules folder does not get included in your local git repository

3. Finally, click the "Source Control" icon in the left bar (it should have a blue dot next to it) and type **"initial commit"** for the message in the

"Message" box. Once this is done, click the checkmark above the message box to commit your changes.

NOTE: If, at this point, you receive the error: "Git: Failed to execute git", try executing the following commands in the integrated terminal:

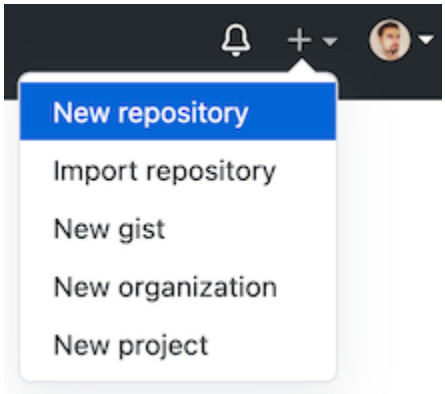
```
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
```

Once this is complete, attempt to click the checkmark again to commit your changes.

Create a GitHub Repository

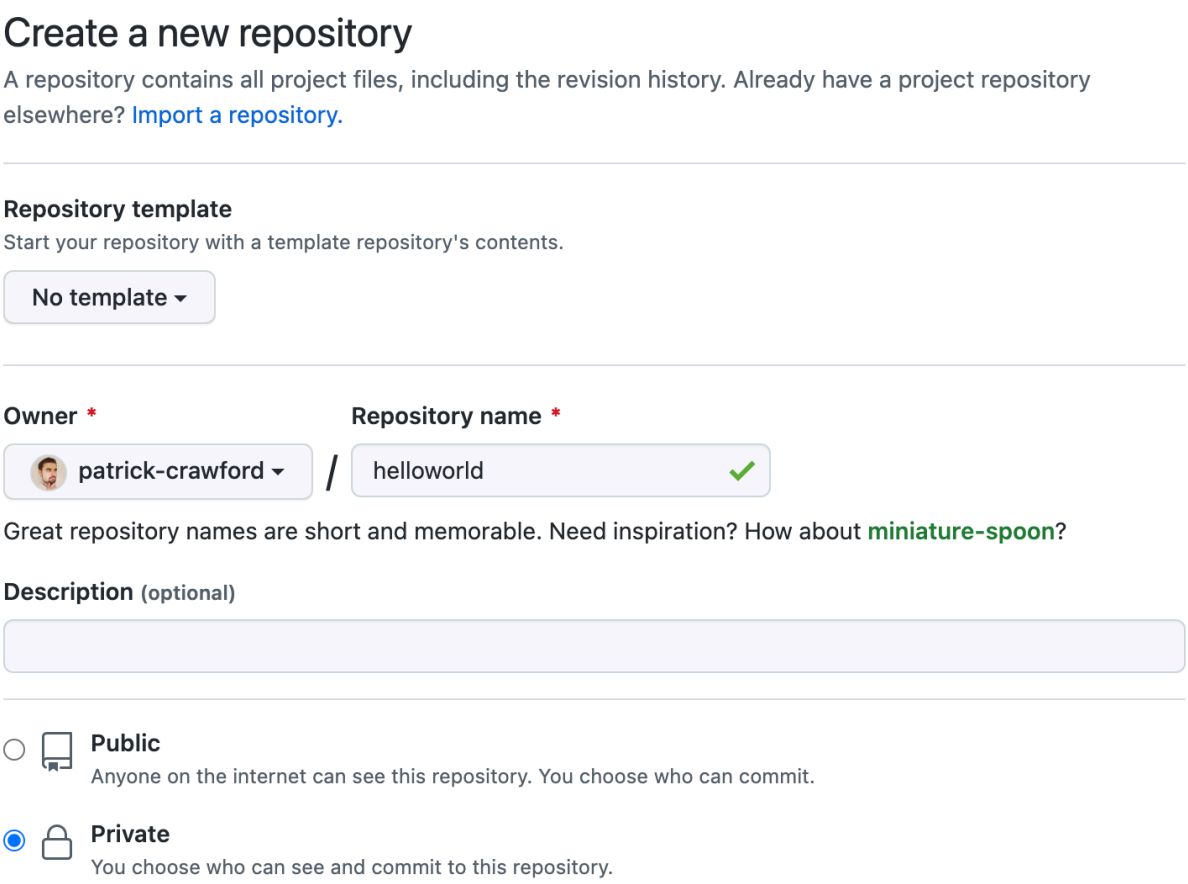
For Vercel to gain access to our code, we must place it on GitHub. Therefore, the next step in this process is creating a GitHub repository for your code:

1. Sign in to your GitHub account.
2. Find and click a "+" button on the Navigation Bar. Then, choose "New Repository" from the dropdown menu.



3. Fill in the repository name text field with the name of your project. Also,

please make sure that the "Private" option is selected:

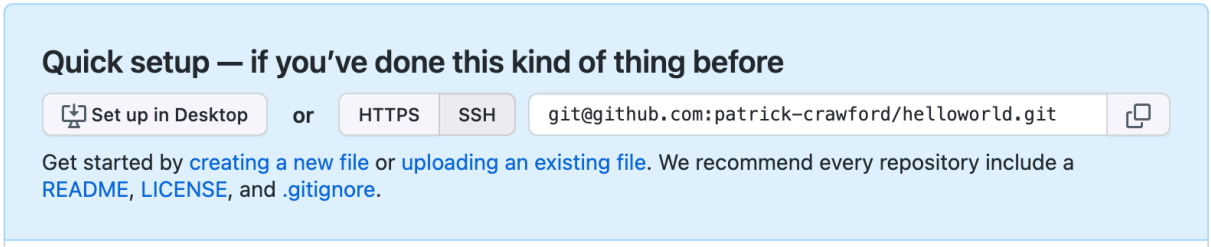


4. Once you're happy with the settings, hit the **"Create repository"** button.

Connect the Local Git Repository to GitHub

Now that our GitHub repository is created, we can proceed to update it with our local copy:

1. First, go to your newly-created GitHub repository and click the "copy" button in the "Quick Setup" block:



This will copy the URL of your remote GitHub repository.

2. Next, go back to your Terminal again and add this remote URL by running the following command:

```
git remote add origin URL
```

where **URL** is the remote repository URL that you have copied in the previous step.

3. To confirm that "origin" was added correctly, run the command: `git remote -v`. You should see something like this:

```
origin git@github.com:patrick-crawford/helloworld.git (fetch)
origin git@github.com:patrick-crawford/helloworld.git (push)
```

4. Finally, you can push the code from your local repository to the remote one using the command:

```
git push origin master
```

> **Important Note:** If at this point, you see the error: "src refspec master does not match any" then "master" is not set as your default branch. Execute the command `git status` to determine which branch

> you're on (it may be "main") and push that instead, ie: `git push origin main`, for example

You can verify that the code was pushed by going back to your Browser and opening your GitHub repository.

Connect the GitHub Repository to Vercel

You should now be ready to push your code to Vercel. First, browse to https://vercel.com and hit the "Start Deploying" button.

1. Next, press the "Continue with GitHub" button, since our code is located on GitHub.

2. If you are not currently logged in to GitHub, you will need to provide your credentials in a pop-up window before continuing.

3. Once you have logged in to GitHub, you will be taken to the **Let's build something new.** screen in Vercel, which prompts you to "Import Git Repository". From here, you will need to click "+ Add GitHub Account"

4. This will prompt you to "Install Vercel". Feel free to install it for "All repositories"

5. You should now see your repository available for import. To proceed, click **Import**

6. At the next page, you are not required to make any changes, as Vercel should detect that we are using Node.js. If you had any *environment variables*, you could set them here as well. Once you are done, click **Deploy**.

7. Once the deploy step has completed, you should be taken to a

"Congratulations!" page with a black button labeled **Go To Dashboard**. Click this to see the information about your deployment.

Make Changes and Push to GitHub

Finally, our code is linked to Vercel via. GitHub!

You should now be able to make any changes you wish and trigger a redeploy of your server on Vercel by simply making changes locally, checking in your code using git and "pushing" it to GitHub, using the above instructions.

Good luck and **Happy Coding!**

Alternative (Render)

Render, like Vercel, also has a free tier that is available without a credit card or separate account (you can use GitHub to sign in):

> "It’s easy to deploy a Web Service on Render. Link your GitHub or GitLab repository and click Create Web Service. Render automatically builds and deploys your service every time you push to your repository. Our platform has native support for Node.js, Python, Ruby, Elixir, Go, and Rust. If these don’t work for you, we can also build and deploy anything with a Dockerfile."
>
> https://render.com/docs/web-services.

Unfortunately, the main drawback of using the free services of Render is that our deployments (web services) are **spun down** after 15 minutes of inactivity. This will cause a **significant** delay in the response of the first request after a period of inactivity while the instance spins up.

For more information see the official documentation on "Free Web Services".

To get started using Render, click the **"GET STARTED FOR FREE"** button on their main site. This will take you to a login page where you can use your GitHub account for authentication.

Once logged in, click the blue **"New +"** button in the top menu bar and choose **"Web Service"**. This will take you to a page where you can choose your GitHub repository for deployment. If you do not see your repository in the "Connect a repository" section, Click "Configure account" under the "GitHub" heading in the right sidebar. This will allow us to grant "Render" permission to all of our repositories (essentially performing the same task that was necessary for Vercel to access our repositories).

Once this is complete and you can see your repository in the list, click the corresponding "Connect" button. You will then be taken to a screen where you must provide:

- A unique name for your web service
- A "start" command (this will typically be **"node server.js"**, ie: the same "start" command that you will find in your package.json file)

Finally, ensure that the "Free" instance type is checked and click **"Create Web Service"** and wait for your code to build and deploy.