# Welcome to Web Programming Tools And Frameworks



Welcome to **Web Programming Tools and Frameworks.** In this course we will be studying a wide range of technologies that are used to create dynamic content on the web. These include modern tools and libraries / frameworks that enable the programmer to quickly and efficiently create a functioning web-based application capable of responding to requests for content, reacting predictably to errors and storing / retrieving user and application data.

We will be looking at what exactly a web application is and how we can use a familiar programming language (JavaScript / ECMAScript) to create and maintain one. Additionally, we will be studying how web browsers send data to and from a web server and how we can ensure that our applications are scalable, secure and robust. We will also study methods of storing and

retrieving data from a data store (SQL & NoSQL Databases) and how to manage state (ie: "logged-in") information about users.

# 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).
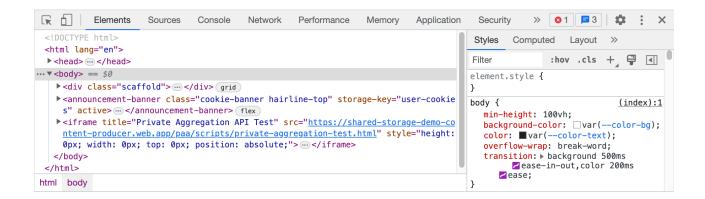
# Modern Web Browser



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:

| | |
|---|---|
| Element Inspector | **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 | **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 | **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 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 | **Sources Panel:** shows a list of all items included in the page (ie: all images, CSS, JavaScript, etc) and their corresponding locations of origin. Developers can click on an item to show it's 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 | **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 | **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 resource and the initiator of the request |

| | |
|---|---|
| Performance Panel | **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 Panel | **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 Panel | **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 (depreciated in favour 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 Panel | **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 referred to by the name of it's predecessor, SSL) and Subresource security (indicating whether the site loads insecure HTTP subresources – ie: |

| | |
|---|---|
| | "mixed content"). |
| Error Icon | **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 | **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 | **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)
- Tailwind (https://tailwindcss.com)
- Multer (https://github.com/expressjs/multer)
- Sequelize (https://sequelize.org)
- Mongoose (https://mongoosejs.com)

## 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.

## MongoDB



MongoDB is another open-source database that we will be exploring in this course. However, unlike MySQL MongoDB is classified as a "NoSQL" database and stores its data in JSON like format rather than in tables with fixed columns. The term NoSQL comes from "Not only SQL" and is intended to mean that it is a type of database system that can store data in non traditional tabular and relational format. It is because of this that NoSQL is quickly becoming a popular alternative to traditional Relational Databases (RDBMS).

We will be exploring how we can leverage NoSQL (MongoDB) to make data management simple and intuitive as well as how it compares to traditional RDBMS systems.

# 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.

# Node.js Globals

Regarding the code that we wrote, it's very simple; however we have made an important assumption: that we have access to a global **"console"** object. In Node.js we have access to a number of global objects / variables in addition to the built-in objects that are built into the JavaScript language. Some of the Node.js Globals that we will be using include:

## 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()

## process

The process object is a global instance of the EventEmitter class that provides information about, and control over, the current Node.js process. It exposes many properties, methods and events related to controlling system interactions.

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

- Methods: process.on(), process.abort(), process.kill(), process.exit()
- Properties: process.stdin, process.stdout, process.stderr, process.pid, process.env
- Events: beforeExit, Exit, uncaughtException

## __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 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.

> ⓘ **INFO**
>
> 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);

let checkMaximum = function () {
  if (count > maxCount) {
    clearInterval(myCountInterval);
  }
};
```

## 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
*/
```

# Built-In Modules / 'require()'

You may have noticed that some of the examples from the documentation include a mandatory 'require()' statement. For example, if we try to execute this *simplified* 'EventEmitter' sample from the documentation:

```
const myEmitter = new EventEmitter();

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

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

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 in the "Web Server Introduction" section (see: "Modules & Node Package Manager"), 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:

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

fs.readFile('names.csv', function (err, fileData) {
  if (err) console.log(err);
  else {
    namesArray = fileData.toString().split(',');
    console.log(namesArray);
  }
});
```

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

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

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

# 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:

```javascript
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
```

# readline

The 'readline' module provides a way to read data from a "Readable stream" (such as process.stdin) one line at a time. For example, we can use this to prompt the user to enter data in the console using the following code:

```javascript
const readline = require('readline');

const rl = readline.createInterface(process.stdin, process.stdout);

rl.question('First Name: ', function (fName) {
```

# Object Oriented JavaScript

Like many other modern languages, 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 (most noteworthy is the "arrow function" syntax - discussed further on) - one common way is to introduce a local variable (often named "that") into the current scope that **holds a reference to "this"**

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

Now, we aren't using the "this" keyword from within the setTimeout() function, but rather "that" from our outputNameDelay function and everything works as it should! (ie, "that" points to architect, since it was the architect that invoked the outputNameDelay method).

# 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;
```

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.

## Private Methods / Properties

Notice how we can access the "name" property of the new Architect objects, directly (ie: without using the "getName()" function)? This is because by default, all properties and methods are "public". If we wish to mark properties as "private" (preventing the property from being accessed directly), we must add a "#" character to the beginning of the property or method name. For example:

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

  constructor(setName = '', setAge = 0) {
    this.#name = setName;
    this.#age = setAge;
  }

  #privateMethod() {
    console.log("I'm a private method");
  }

  setName(newName) {
    this.#name = newName;
  }

  setAge(newAge) {
    this.#age = newAge;
```

If we now try to access the "#name" property directly on an object created with this class, we get the following error:

```
SyntaxError: Private field '#name' must be declared in an enclosing class
```

## Getters / Setters

If we do wish to provide direct access to the "name" and "age" properties however, we can use "setters" and "getters". This way, we have more control over how the properties are manipulated and retrieved, internally to the class. For example, if we want controlled access to the "name" and "age" properties, we could use the following syntax:

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

  constructor(setName = '', setAge = 0) {
    this.#name = setName;
    this.#age = setAge;
  }

  #privateMethod() {
    console.log("I'm a private method");
  }

  set name(newName) {
    this.#name = newName;
  }
```

# Inheritance

A core principal of Object-Oriented Programming is "inheritance":

> "a mechanism where you can to derive a class from another class for a hierarchy of classes that share a set of attributes and methods."
>
> https://stackify.com/oop-concept-inheritance

In JavaScript, this is implemented via the "Prototype Chain":

> "When it comes to inheritance, JavaScript only has one construct: objects. Each object has a private property which holds a link to another object called its prototype. That prototype object has a prototype of its own, and so on until an object is reached with null as its prototype. By definition, null has no prototype, and acts as the final link in this prototype chain."
>
> https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain.

This is why we can access properties like "length" on a string, even though it is a primitive - it exists on the **prototype** of the built-in String Object. When we access the "length" property, a string primitive is automatically wrapped in a "String" object via a process known as "auto-boxing" and we gain access to the property on the prototype.

To see this in action, let's modify the String prototype after we create a new string primitive. Once the string primitive is "auto-boxed" with the String object, we should have access to whatever we add on the prototype:

```
let name = 'Thomas Anderson';
```

Now that we know a little about how inheritance is implemented in JavaScript, you might be asking: "how does this work in our class definition?" and "do we have to modify the prototype of new objects directly?"

Fortunately, JavaScript has added the "extend" keyword so that we do not have to. For example, if our "architect" class inherits from a more generic "Job" class, we could write the code:

```javascript
class Job {
  #name;
  #age;

  constructor(setName = '', setAge = 0) {
    this.#name = setName;
    this.#age = setAge;
  }

  set name(newName) {
    this.#name = newName;
  }

  set age(newAge) {
    this.#age = newAge;
  }

  get name() {
    return this.#name;
  }

  get age() {
    return this.#age;
  }
}
```

for the "Job" class and add the new functionality for the "Architect" class using

the following code:

```
class Architect extends Job {
  #occupation = 'architect';

  constructor(setName = '', setAge = 0) {
    super(setName, setAge); // invoke the "parent" constructor
  }

  #privateMethod() {
    console.log("I'm a private method");
  }
}

let architect1 = new Architect('Joe', 34);
console.log(architect1.name);
```

To verify that Job is indeed part of the "prototype chain" of the new architect1 object, we can use the familiar "prototype" test from above, ie:

```
Job.prototype.sayHello = function () {
  console.log('Hello from Job!');
};

architect1.sayHello(); // Hello from Job!
```

# Modern Syntax

JavaScript is constantly evolving. Since 2015 with the release of ECMAScript 6 (ES6), there has been a new release every year. This steady release schedule means that it is extremely important to be familiar with some of the concepts released in the last few years as more and more examples, tutorials and online documentation make use of these features. It can be easy to fall behind and find some of the new syntax unusual or confusing.

To help navigate these notes and other related documentation, we have outlined some of the more important, ubiquitous features released since ES6:

# Functions

In JavaScript, functions are typically defined using either a declaration or expression and may contain either a fixed or variable list of parameters, which may or may not have default values.

However, as JavaScript evolved, additional options and features for working with functions have as well. The following sections outline some of the features that we will use in these notes:

## Arrow Functions

ES6 (ECMAScript 2015) introduced a compact version of function expressions known as "Arrow functions", for example:

```
let adder = function (num1, num2) {
  return num1 + num2;
```

Essentially, we have removed the "function" keyword and replaced it with an arrow following the parameter list. While this is indeed shorter, we can compress the function expression even further as arrow functions use an "implicit return". This means that if the curly brackets ("{" and "}") are *omitted* from the arrow function, the inner expression is returned:

```
let adderArrowShort = (num1, num2) => num1 + num2;
```

Additionally, if there is only a *single* parameter, the brackets *surrounding* the parameters may also be omitted, ie:

```
let squared = num => num * num;
```

> ⓘ **INFO**
>
> If there are 0 parameters to the function, empty round brackets must be used, ie:
>
> ```
> let getHello = () => 'Hello World';
> ```

**Lexical "this"**

Arrow functions are great for creating simplified code that is easier to read (sometimes referred to as "syntax sugar"), however there is another very useful and slightly misleading feature that we have yet to discuss: the notion of a "lexical 'this'". Recall that when we added the "outputNameDelay" method to the architect object, we had to overcome the issue with "this" pointing at the incorrect object by introducing a new local variable, "that":

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

While this does solve the problem, wouldn't it be better if we didn't have to always create a new local variable to sit in for "this"? Fortunately, arrow functions actually use a "lexical this" instead of their own value for "this", so functions defined using the arrow notation use the "this" value of their parent scope.

With this in mind, we can re-write the above function using an arrow function to achieve the same result without having to introduce any new variables to handle the "this" issue. Additionally, because it's such a simple function, we can transform it into a single line:

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

This is a typical use of arrow functions, ie: to simplify a scenario in which we need to declare a function in place, often as a parameter to other functions ("callbacks"). We don't have to concern ourselves with how "this" will behave in the new context and the added "syntax sugar" makes the operation much simpler to read and shorter to code.

## Destructuring Object Parameters

Another common feature introduced in ES6 is the ability to perform a

"destructuring assignment" for objects. For example, if we have the following code that defines a "product" object:

```
let product = {
  id: '145be9',
  price: 1.35,
  onSale: false,
};
```

and we wish to extract the "price" and "id" values into separate variables, we would typically use the following syntax:

```
let price = product.price;
let id = product.id;
```

However, this can be shortened to the following, using a "destructuring assignment":

```
let { price, id } = product;
```

This type of syntax is commonly used when passing object properties as parameters to functions. For example, instead of the following code:

```
function outputProduct(productObj) {
  console.log('Product', productObj.id, productObj.price);
}
```

we could use the more concise:

```
function outputProduct({ id, price }) {
```

# Arrays

## Iterating

An Array in JavaScript is technically an "indexed collection", ie: "an ordered list of values that you refer to with a name and an index". Because of this, the simplest ways to iterate over the collection are with the common for loop and do...while / while loops. However, there are other ways to iterate over an array, including:

### for...of loop

The for...of statement "executes a loop that operates on a sequence of values sourced from an iterable object. Iterable objects include instances of built-ins such as Array, String, TypedArray, Map, Set, NodeList (and other DOM collections), as well as the arguments object, generators produced by generator functions, and user-defined iterables."

```javascript
let sample = ['A', 'B', 'C'];

for (const element of sample) {
  console.log(element);
}
```

### forEach() Method

The forEach() method of the Array object can be used to execute a function once per element of the array, with the element (and optionally, the index) as the parameter(s), for example:

```
let sample = ['A', 'B', 'C'];

sample.forEach((element, index) => console.log(element + ' at
index: ' + index));
```

# Destructuring Elements

As we have seen above, ES6 introduced the "destructuring assignment". We used this feature to make the syntax for extracting properties from objects more concise and to clarify function parameters. Fortunately, this feature is also available for arrays using a similar process:

```
let sample = ['A', 'B', 'C'];

// let a = sample[0];
// let b = sample[1];

let [a, b] = sample;
```

Here, we assign the variables a & b at the same time by "destructuring" the array. This syntax is popular in libraries such as React (for example, when using the common "useState" hook"), so it's important that we become familiar with it.

## Spread Syntax

You have likely seen the "..." syntax before in JavaScript. A common use for it is in the form of "rest" parameters, which allow for the creation of functions that take on an *unknown* number of parameters:

```javascript
function sum(...numbers) {
  let total = 0;

  for (const num of numbers)
    total += num;

  return total;
}

console.log(sum(1, 2, 3, 4, 5, 6)); // 21
```

However, "..." can also be used outside of function parameters as a placeholder for values in an array (or properties in an object). This is commonly referred to as "spread" syntax. For example, if we wished to merge two arrays to create a new array (without using the built-in "concat" function), one option is to use the following code:

```javascript
let sample1 = ['A', 'B', 'C'];
let sample2 = ['D', 'E', 'F'];

let sample3 = [];
```

Here, we must loop through each array and add each element in turn to a new array. However, using the "spread" syntax, we can instead use the following code:

```
let sample1 = ['A', 'B', 'C'];
let sample2 = ['D', 'E', 'F'];

let sample3 = [...sample1, ...sample2];

console.log(sample3); // [ 'A', 'B', 'C', 'D', 'E', 'F' ]
```

By using the "..." syntax, we're essentially saying "the elements of the array".

> **⊘ INFO**
>
> This can be used for objects as well, ie:
>
> ```
> let product = {
>   id: '145be9',
>   price: 1.35,
>   onSale: false,
> };
>
> let productWithStore = { ...product, store: '53' };
>
> console.log(productWithStore); // { id: '145be9', price: 1.35,
> onSale: false, store: '53' }
> ```

It is important to note however, that while we are using the "..." to create a new copy of arrays / objects, it is only a **"shallow" copy** (ie: it will not copy "nested" elements and properties, leaving a reference to the original array / object).

# Strings

## Template Literals

A common way to place text and data together in a single string in JavaScript is to use the "+" operator. For example:

```
let x = 5, y = 6;
console.log(x + " + " + y + " = " + (x + y)); // 5 + 6 = 11
```

However, wouldn't it be simpler if we could have a single string with *placeholders* for data, rather than multiple strings placed *next* to data, concatenated using the "+" operator?

Fortunately, ES6 has introduced "Template literals" sometimes called "Template strings", which use the ( ` ) character to define the string and the "${expression}" syntax to insert an expression into the string to be evaluated.

Using this, we can re-write our above example to remove the "+" operator and instead use the more concise (and easier to read):

```
let x = 5, y = 6;
console.log(`${x} + ${y} = ${x + y}`); // 5 + 6 = 11
```

Additionally, since the "${}" syntax within the template literal allows to evaluate an expression, we can also execute functions and other logic within the string definition, such as:

```
let shapes = ['circle', 'square', 'triangle'];
console.log(`My favourite shapes are:${shapes.map((shape, index)
=> ` ${index + 1}: ${shape}`)}`);

// My favourite shapes are: 1: circle, 2: square, 3: triangle
```

> ⓘ **INFO**
>
> We also have the added bonus of creating **multi-line** strings, ie:
>
> ```
> let myString = `Hello
> World`;
>
> console.log(myString);
> // Hello
> // World
> ```

# Errors

One of the most important aspects of writing any program is elegantly handling errors. It is important to never let your program suddenly crash or enter an unknown state due to an unanticipated error. JavaScript features numerous mechanisms to handle certain types of logical errors; for example the global isNaN() function is a way to elegantly respond to a situation in which a number was expected, but not returned:

```
let x = 'twenty';
let y = parseInt(x);

if (isNaN(y)) {
```

Similarly, we can use the global isFinite() function to handle a situation where division by zero has occurred:

```
let x = 30, y = 0;
let z = x / y;

if (isFinite(z)) {
  console.log(`success! ${x} / ${y} = ${z}`);
} else {
  console.log(`${x} is not divisible by ${y}`);
}
```

## try / catch

While the above functions are extremely useful for handling logical errors, they are not sophisticated enough to handle a situation that would completely break your code and cause the program to fail. For example, consider the following example that uses our new "const" keyword:

```
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 get to exit the program gracefully. This can be a huge problem if, for example we were working with a live connection to a

service and an unexpected error occurred. Our program would crash and we would not be able to respond to the error by alerting the user and properly closing the connection. 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:

```javascript
const PI = 3.14159;

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

try {
  PI = 99;
} catch (ex) {
  console.log('uh oh, an error occurred!');
}

console.log(`Alas, it cannot be done, PI remains: ${PI}`);
```

If we execute the above code in Node.js we will find that our program doesn't crash and that our string: "Alas, it cannot be done, PI remains: 3.14159" gets correctly logged to the terminal! Additionally, we can execute a specific block of code right when the error is encountered; in this case we output "uh oh, an error occurred!". This is not very useful to help us debug the error, but it better than having the program crash and at least we know that an error did indeed occur. If we wish to obtain additional information about the error, we can make use of some of the properties / methods of the **Error** object that was thrown as an exception and caught in our "catch" block. For example, we can alter the code to use the "message" property of the caught exception (ex) to display a more helpful error:

```javascript
const PI = 3.14159;
```

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.

Lastly, if we have some code that we would like to execute regardless of whether or not the code in our "try" block is successful, we can use a "finally" block:

```javascript
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.
} finally {
  console.log('always execute code in this block');
}

console.log(`Alas, it cannot be done, PI remains: ${PI}`);
```

## Throwing Errors

Now that we know how to correctly handle errors that have been thrown by the Node.js runtime environment or by other code / modules included in our solutions, why don't we try throwing our **own exceptions**? This is very straightforward and only requires the use of the **"throw"** keyword and (typically) an **Error** Object:

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

let a = 3,
  b = 0,
  c;

try {
  c = divide(a, b);
} catch (ex) {
  console.log(`uh oh, an error occurred: ${ex.message}`);
  // outputs: uh oh, an error occurred: Division by Zero!
  c = NaN;
}

console.log(`${a} / ${b} = ${c}`); // 3 / 0 = NaN
```

Notice how the code below the "throw" statement does not get executed, and the flow of execution goes directly into the catch block. This prevents the error from propagating and ensures that it is handled immediately. As you can see, we can throw a new error whenever we detect that an error *may* occur anywhere in our code. In the above example, we check if our second parameter (y) is zero (0) and rather than trying to do the division, we immediately throw a custom error with the message "Division by Zero!". If the function call exists in a "try" block ( as above ), the execution of the code will immediately continue in the "catch" block and we mitigate the error by setting "c" to NaN.

# Example Code

You may download the sample code for this topic here:

JavaScript-Review

# 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 back to our "setTimeout" example; only this time we will wait 2 seconds (2000 milliseconds) and execute some code *before* and *after* the function:

```javascript
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).

```javascript
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:

```javascript
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:

```javascript
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:

```js
// 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:

```javascript
// 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:

```javascript
// 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:

```javascript
// 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:

```javascript
// 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
```

## 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*:

```javascript
// 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(() => {
```

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:

```javascript
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 (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().
```

> ⚠ **INFO**
>
> 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:
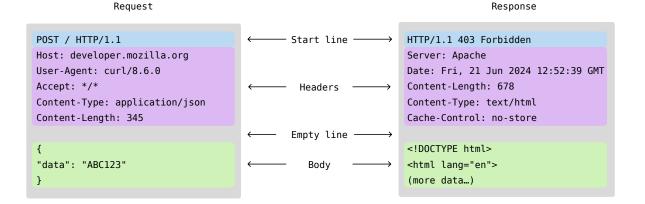
Handling-Asynchronous-Code

# 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.

```
POST / HTTP/1.1
Host: developer.mozilla.org
User-Agent: curl/8.6.0
Accept: */*
Content-Type: application/json
Content-Length: 345

{
"data": "ABC123"
}
```

←——— Start line ———→
←——— Headers ———→
←——— Empty line ———→
←——— Body ———→

```
HTTP/1.1 403 Forbidden
Server: Apache
Date: Fri, 21 Jun 2024 12:52:39 GMT
Content-Length: 678
Content-Type: text/html
Cache-Control: no-store

<!DOCTYPE html>
<html lang="en">
(more data…)
```

(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.

```
                    Request

POST / HTTP/1.1
Host: example.com/listener
User-Agent: curl/8.6.0              ←——————  Request headers
Accept: */*
Content-Type: application/json
Content-Length: 345                ←——————  Representation headers


{
  "data": "ABC123"
}
```

(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:
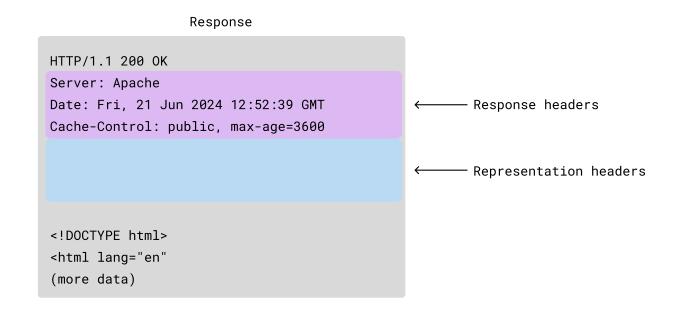   - `HTTP/1.1 404 Not Found`

# 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.

Response

```
HTTP/1.1 200 OK
Server: Apache
Date: Fri, 21 Jun 2024 12:52:39 GMT      ⟵——— Response headers
Cache-Control: public, max-age=3600


                                          ⟵——— Representation headers



<!DOCTYPE html>
<html lang="en"
(more data)
```

(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

Recall way back in the Introduction, the concept of "Built-In Modules / 'require()'" was discussed:

> "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 used this to gain access to some of the built in logic that ships with Node, including: 'fs', 'path' and 'readline'.

## 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**

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

message.writeMessage('Hello World!');
```

**file: ./modules/message.js**

```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
};

let localMessage = '';

module.exports.writeMessage = (msg) => {
  localMessage = msg;
};

module.exports.readMessage = () => {
  console.log(`${localMessage} from ${__filename}`);
};
```

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 [module name] | uninstall does exactly what you would think, it uninstalls a 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. |

| Command | Description |
| --- | --- |
| 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 used 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",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

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:

```json
{
  "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 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"**.

> ⓘ **INFO**
>
> 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:

```json
{
  "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 (your version may differ from the below):

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

You will also notice that a 2<sup>nd</sup> 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**

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

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
```

> ⓘ **INFO**
>
> 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 /)

> ⓘ **INFO**

> 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" />
```

**File: /MyServer/views/about.html**

```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:

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

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:

```js
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

# What is a UI Toolkit / Framework?

Creating a web site / application that is functional, visually appealing and provides a good user experience can be difficult. In fact, there are entire branches of web application development that focus entirely on the front end and/or user experience (https://brainstation.io/career-guides/what-is-a-ui-designer). However, when first developing your application, you may not have access to a UI / UX designer and so you must craft the user interface yourself. Fortunately, there are a plethora of resources that can help with this task. The easiest way to build your web application in a way that adheres to fundamental design principles, is to use a pre-built CSS / JS UI toolkit or framework. These typically come with user interface "components" that are styled and ready to use out of the box.

## Popular Frameworks

The following is a list of some popular frameworks and how to quickly get started using them. Typically all we need to do is include their CSS / JS files using a CDN, then reference the documentation for some boilerplate, starter code, ie:

```html
<!-- Create a button in Bootstrap using the "Primary" colour -->
<button type="button" class="btn btn-primary">Primary</button>
```

### Bootstrap

One of oldest and most popular UI frameworks is "Bootstrap":

> "Powerful, extensible, and feature-packed frontend toolkit. Build and customize with Sass, utilize prebuilt grid system and components, and bring projects to life with powerful JavaScript plugins."

If we wish to incorporate Bootstrap into our projects, we can link to the files directly using the "jsdelivr" Content Delivery Network (CDN):

```html
<link
  href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css"
```

> ⚠️ **CAUTION**
>
> It is important that we include the CDN links *before* our own CSS / JS.

## Materialize

Back in June, 2014 Google introduced "Material Design":

> "Material Design is an adaptable system of guidelines, components, and tools that support the best practices of user interface design. Backed by open-source code, Material Design streamlines collaboration between designers and developers, and helps teams quickly build beautiful products."
>
> https://m3.material.io

A simple way to get started with Material Design is to use "Materialize", which is described as a "responsive front-end framework based on Material Design". Essentially, Materialize provides the CSS and JS for components that follow the Material Design guidelines.

To get started using Materialize, we can follow the same strategy as we used for Bootstrap, ie: using the "cloudflare" CDN links directly in our HTML documents:

```html
<link
  rel="stylesheet"
  href="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/css/materialize.min.css"
/>
<script src="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/js/materialize.min.js"></script>
```

## Bulma

Another alternative is "Bulma": It was released in 2016 and has had a number of releases - at the time of writing, Bulma is at 0.9.4.

> "Bulma is a free, open source framework that provides ready-to-use frontend components that you can easily combine to build responsive web interfaces.
>
> No CSS knowledge required."
>
> https://bulma.io

You will notice that Bulma does not require any JS to run, making it more straightforward to incorporate into existing projects. It also provides a simple "modular" approach to including "only what you need" from the framework.

As with other frameworks on this list, the simplest way to start is to use the minified CSS, available on the "jsdelivr" CDN:

```html
<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bulma@0.9.4/css/bulma.min.css" />
```

## Starling UI

Starling UI is a lightweight UI toolkit that provides color tokens, typography, buttons, cards, navigation, a grid system, and even a simple modal with optional JavaScript helpers. You can include the entire toolkit or just the components you need.

To get started, you can either include everything:

```html
<link rel="stylesheet" href="https://starlingui.netlify.app/sui.css" />
```

or just the pieces you need. For example, to include a button:

```html
<link rel="stylesheet" href="https://starlingui.netlify.app/utils/color.css" />
<link rel="stylesheet" href="https://starlingui.netlify.app/pages/components/button/button.css" />

<button class="sui_btn btn_outline">Click Me</button>
```

## Foundation

Finally, we should mention "Foundation" - a framework released by "Zurb" back in 2011. It has gone through a number of major releases since then and (at the time of writing) is currently on version 6, released in 2015. Foundation has extensive documentation and while it may be more complex then some of the other frameworks, it has may resources such as "starter projects" and video tutorials to help new users. Interestingly, it also has a version that is used to help design "responsive HTML emails", which can be cumbersome and difficult.

To get started using it, the simplest way is to use the CDN links:

```
<link
  rel="stylesheet"
  href="https://cdn.jsdelivr.net/npm/foundation-sites@6.7.5/dist/css/
foundation.min.css"
  crossorigin="anonymous"
/>
<script
  src="https://cdn.jsdelivr.net/npm/foundation-sites@6.7.5/dist/js/
foundation.min.js"
  crossorigin="anonymous"
></script>
```

# Introduction to Sass

After exploring the documentation for the above frameworks, you will notice that all them make use of something called "Sass":

> "CSS on its own can be fun, but stylesheets are getting larger, more complex, and harder to maintain. This is where a preprocessor can help. Sass has features that don't exist in CSS yet like nesting, mixins, inheritance, and other nifty goodies that help you write robust, maintainable CSS."
>
> https://sass-lang.com/guide

Sass, or "Syntactically Awesome StyleSheets" is a superset of CSS that adds power and elegance to the basic language. It allows you to use variables, nested rules, mixins, modules, and more, all with a fully CSS-compatible syntax. Sass helps keep large stylesheets well-organized as well as getting small stylesheets up and running quickly. This is a natural choice for large CSS frameworks like those mentioned above.

There are two syntaxes available for Sass. The first, known as SCSS (defined using the ".scss" extension) is an **extension** of the syntax of CSS while the other syntax SASS (defined using the ".sass" extension), provides a more concise way of writing CSS. It uses indentation rather than brackets to indicate nesting of selectors, and newlines rather than semicolons to separate properties.

## Getting Started

To get started working with Sass, first create a simple web server using Express, making sure that there is at least one route and a "public" static folder has been configured.

Once this is complete run the following command to install the "sass" command as a "devDependency" (ie: a package that that is only needed for local development and testing.)

```
npm i -D sass
```

We require this command because Sass functions as a CSS precompiler - it adds functionality to CSS in a layer **above** it and we must run a script / program to convert our Sass files into regular CSS so that the browser can interpret it.

With our command correctly installed, we should add a "scss" folder *outside* the "public" folder (it does not need to be accessed by the client) and create a couple of .scss files (main.scss & _reset.scss):

```
/scss
  ↳ main.scss
  ↳ _reset.scss
```

Finally, to make sure our new "sass" CLI works to "watch" our **.scss** files for changes and correctly update a new file: **/public/css/main.css**, we must add the following "scripts" property to our "package.json" file:

```
"scripts": {
  "build-css": "sass --no-source-map --watch scss:public/css"
}
```

> ⊙ **INFO**
>
> From the documentation: "When compiling whole directories, Sass will ignore partial files whose names begin with _. You can use partials to separate out your stylesheets without creating a bunch of unnecessary output files." This is why we do not get a "_reset.css" file as a result of our build step (below)

Notice how the "build-css" script is set to run "sass" with "**scss**" as the ***source*** and "**public/css**" as the ***destination***. We also use the **--no-source-map** and **--watch** flags to ensure that the command will not generate a source map as well as "watch" our source directory for changes and automatically re-compile on save.

To get sass running, simply execute the command:

```
npm run build-css
```

Now, every time we make a change to any file within our **/scss** directory our Sass will be compiled and the resulting CSS will be saved in the pubic css folder as **main.css**. We can leave this process running in the background and not have to worry about any additional "compile" steps.

You will know when the process has completed successfully when you see the following green message in the terminal: **Compiled scss/main.scss to public/css/main.css.**. Similarly, you will know that there was an error compiling your SCSS if you see an "Error" message in the terminal.

## Working with SCSS

With our script humming along in the background waiting for changes, why don't we try out some of the great features of our new CSS extension language?

**Variables**

Think of variables as a way to store information that you want to reuse throughout your stylesheet. You can store things like colors, font stacks, or any CSS value you think you'll want to reuse. Sass uses the $ symbol to make something a variable. Here's an example:

```
$font-stack: Helvetica, sans-serif;
$primary-color: #333;

body {
  font: 100% $font-stack;
  color: $primary-color;
}
```

When the Sass is processed, it takes the variables we define for the $font-stack**and**primary-color$ and outputs normal CSS with our variable values placed in the CSS. This can be extremely powerful when working with brand colors and keeping them consistent throughout the site.

```
body {
  font:
    100% Helvetica,
    sans-serif;
```

**Nesting**

When writing HTML you've probably noticed that it has a clear nested and visual hierarchy. CSS, on the other hand, doesn't. Sass will let you nest your CSS selectors in a way that follows the same visual hierarchy of your HTML. Be aware that overly nested rules will result in over-qualified CSS that could prove hard to maintain and is generally considered bad practice. With that in mind, here's an example of some typical styles for a site's navigation:

```
nav {
  ul {
    margin: 0;
    padding: 0;
    list-style: none;
  }

  li {
    display: inline-block;
  }

  a {
    display: block;
    padding: 6px 12px;
    text-decoration: none;
  }
}
```

You'll notice that the **ul**, **li**, and **a** selectors are nested inside the **nav** selector. This is a great way to organize your CSS and make it more readable. When you generate the CSS you'll get something like this:

```
nav ul {
  margin: 0;
  padding: 0;
  list-style: none;
}

nav li {
  display: inline-block;
}

nav a {
  display: block;
```

**Import (Use)**

CSS has an import option that lets you split your CSS into smaller, more maintainable portions. The only drawback is that each time you use **@import** in CSS it creates another HTTP request. Sass builds on top of the current CSS **@import** (as **@use**) but instead of requiring an HTTP request, Sass will take the file that you want to import and combine it with the file you're importing into so you can serve a single CSS file to the web browser.

In our Sass directory, we have two scss files: **_reset.scss** and **main.scss**. We want to import **_reset.scss** into **main.scss**.

```
// _reset.scss

html,
body,
ul,
ol {
  margin: 0;
  padding: 0;
}
```

```
// main.scss

@use '_reset';

body {
  font:
    100% Helvetica,
    sans-serif;
  background-color: #efefef;
}
```

Notice we're using **@use '_reset';** in the **main.scss** file. When you import a file you don't need to include the file extension **.scss**. Sass is smart and will figure it out for you. When you generate the CSS you'll get:

```
html,
body,
ul,
ol {
```

**Mixins**

Mixins let you make groups of CSS declarations that you want to reuse throughout your site. You can even pass in values to make your mixin more flexible, making it more like a function definition within your CSS. The "Bulma" framework makes use of the following "overlay" mixin which can be used to help position elements on the page using "absolute" positioning (ie a "modal" window):

```scss
@mixin overlay($offset: 0) {
  bottom: $offset;
  left: $offset;
  position: absolute;
  right: $offset;
  top: $offset;
}

.modal {
  @include overlay(150px);
}
```

To create a mixin you use the **@mixin** directive and give it a name, ie "overlay". We're also using the variable **$offset** inside the parentheses so we can pass in an offset of whatever we want (using "0" as the default). After you create your mixin, you can then use it as a CSS declaration starting with **@include** followed by the name of the mixin. When your CSS is generated it'll look like this:

```css
.modal {
  bottom: 150px;
  left: 150px;
  position: absolute;
  right: 150px;
  top: 150px;
}
```

**Extend / Inheritance**

This is one of the most useful features of Sass. Using **@extend** lets you share a set of CSS properties from one selector to another. It helps keep your Sass very DRY ("Don't Repeat Yourself"). In our example we're going to create a simple series of messaging for errors, warnings and successes.

```
.message {
  border: 1px solid #ccc;
  padding: 10px;
  color: #333;
}

.success {
  @extend .message;
  border-color: green;
}

.error {
  @extend .message;
  border-color: red;
}

.warning {
  @extend .message;
  border-color: yellow;
}
```

What the above code does is allow you to take the CSS properties in **.message** and apply them to **.success**, **.error**, & **.warning**. The magic happens with the generated CSS, and this helps you avoid having to write multiple class names on HTML elements. This is what it looks like:

```
.message,
.success,
.error,
.warning {
  border: 1px solid #cccccc;
  padding: 10px;
  color: #333;
}

.success {
  border-color: green;
}

.error {
  border-color: red;
}

.warning {
  border-color: yellow;
```

**Operators**

Doing math in your CSS is very helpful. Sass has a handful of standard math operators like **+**, **-**, **\***, **/**, and **%**. For example we can do the following simple math to calculate widths for an **aside** & **article**.

```
.container {
  width: 100%;
}

article[role='main'] {
  float: left;
  width: calc(600px / 960px * 100%); // gets 600px as a percentage of 960px
}

aside[role='complementary'] {
  float: right;
  width: calc(300px / 960px * 100%); // gets 300px as a percentage of 960px
}
```

In the above case, the generated CSS will look like:

```
.container {
  width: 100%;
}

article[role='main'] {
  float: left;
  width: 62.5%;
}

aside[role='complementary'] {
  float: right;
  width: 31.25%;
}
```

# Tailwind CSS & daisyUI

Tailwind CSS is another popular CSS framework that we may choose to use with our projects. It is defined as a "utility-first CSS framework packed with classes like **flex**, **pt-4**, **text-center** and **rotate-90** that can be composed to build any design, directly in your markup."

> "Utility classes help you work within the constraints of a system instead of littering your stylesheets with arbitrary values. They make it easy to be consistent with color choices, spacing, typography, shadows, and everything else that makes up a well-engineered design system."

Essentially, Tailwind CSS provides an **extensive** set of CSS classes that can be used together to create specific designs by adding them to elements in your HTML (Markup), without writing any CSS. For example:

```html
<div class="w-[150px] h-[80px] shadow-2xl bg-white rounded-lg flex justify-center items-center">
  <p class="text-center">shadow-2xl</p>
</div>
```

Creates a block that is 150px wide by 80px tall with a large outer shadow, a white background color and large rounded corners. It also uses flexbox (flex) to center the items horizontally and vertically. The inner paragraph is also aligned in the center.

This is certainly a different approach to the previous CSS frameworks that we have seen. It adds a lot of extra markup to your "view" (".html") files and can be difficult to maintain and read. However, it does add a lot of flexibility and consistency to the user interface design without writing any CSS code yourself.

If we are able to use this alongside some kind of "component library" that also adds expertly designed, pre-built user interface elements (such as Bootstrap's "btn"), it would be much quicker (and cleaner) for us to adopt in our projects. Fortunately, Tailwind has the notion of "plugins" which allow us to "register new styles for Tailwind" (this is where daisyUI comes in).

# Setting up Tailwind CSS

To begin using Tailwind, we will once again create a simple web server using Express, making sure that there is at least one route and a "public" static folder has been configured.

Once this is complete, run the following command to install the "tailwindcss" command as a "devDependency":

```
npm install -D tailwindcss@latest @tailwindcss/cli@latest
```

With Tailwind CSS correctly installed, we must create a primary css file as our

starting point to include Tailwind CSS. To do this, crate a "tailwind.css" file within "/public/css" (ie: `/public/css/tailwind.css`) with the following code:

```
@import "tailwindcss";
```

## "Build" main.css

For the next step (as we did with "SASS"), we must add a "build" command to the "scripts" section of our package.json file:

**File: package.json**

```
"scripts": {
  "tw:build": "npx @tailwindcss/cli -i ./public/css/tailwind.css
-o ./public/css/main.css --watch"
}
```

To test this out, add the following import statement and HTML to one of your "view" (."html") files:

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

```
<div class="w-[150px] h-[80px] shadow-2xl bg-white rounded-lg flex
justify-center items-center">
  <p class="text-center">shadow-2xl</p>
</div>
```

Finally, "build" the "main.css" file by executing the command in the Integrated Terminal:

```
npm run tw:build
```

**Congratulations!** You have crated a new main.css file with all of the required Tailwind CSS classes for your view, including "shadow-2xl", "bg-white", etc. Now, every time you decide to update any of your "view" files or "tailwind.css", you can re-run the `npm run tw:build` command to re-generate your optimized main.css file.

> ⊙ **INFO**
>
> Tailwind CSS Recommends the Visual Studio Extension "Tailwind CSS IntelliSense", which has features such as:
>
> - **Autocomplete:** Intelligent suggestions for class names, as well as CSS functions and directives
> - **Linting:** Highlights errors and potential bugs in both your CSS and your markup
> - **Hover Preview:** See the complete CSS for a Tailwind class name by hovering over it

# Introducing daisyUI

As mentioned above, daisyUI is a "plugin" for Tailwind CSS. It adds UI components while still providing the full flexibility of Tailwind's utility classes:

> "[daisyUI is] the most popular component library for Tailwind CSS"
>
> "daisyUI adds component class names to Tailwind CSS so you can make beautiful websites faster than ever".
>
> https://daisyui.com

This sounds like the perfect compromise - we get beautifully styled UI components while still having the freedom to use the extensive set of Tailwind's utility classes to configure them and design new components.

# Installing

To get started using daisyUI, the first thing that we need to do is "install" the required packages using NPM:

```
npm install -D @tailwindcss/typography@latest daisyui@latest
```

Once this is complete, the next step is to register them as "plugins" in the "tailwind.css" file:

**File: /public/css/tailwind.css**

```
@plugin "@tailwindcss/typography";
@plugin "daisyui";
```

> ⚠️ **CAUTION**
>
> The "@tailwindcss/typography" plugin is recommended by daisyUI and is **required** if we wish to see text such as headings and paragraphs **correctly styled**. To read more on this, including using the **"prose"** class ("that you can slap on any block of vanilla HTML content and turn it into a beautiful, well-formatted document"), see the following link to the documentation:
>
> https://daisyui.com/docs/layout-and-typography

This should be all that is required to register daisyUI with Tailwind CSS. To

verify that it did indeed work, try adding a component from daisyUI to one of your .html files using main.css. The simplest example is a button - we'll use the "primary" variation:

```
<button class="btn btn-primary">Button</button>
```

To test if this worked, all that is needed is to execute another "build" of tailwind, using the previously-configured "tw:build" command:

```
npm run tw:build
```

You should now see ".btn" and ".btn-primary" classes added to your main.css, as well as a working purple button in your view!

## Theming

Before we discuss some of the important design patterns and components available from daisyUI, let's quickly discuss how we can modify the "theme" that is used when rendering our components:

> "daisyUI comes with a number of themes, which you can use with no extra effort. Each theme defines a set of colors which will be used on all daisyUI elements"

At it's core the various "themes" that daisyUI provides are alternate color schemes and variations on the roundness of corners, thickness and length of shadows, etc. for the provided components. Recall from our "btn-primary" example above: a purple button was rendered on the screen (since this is the default "primary" color). If we were to use one of the built in themes such as "cupcake", the "primary" color would be a dark cyan and the button would have more rounded edges.

To change the theme used by daisyUI, all that is required is that a named theme from the list of themes be listed in a "themes" property for "daisyui" in the tailwind.css file. For example, if we wish to use the aforementioned "cupcake" theme, we could update our tailwind.css file as follows:

**File: /public/css/tailwind.css**

```
@plugin "daisyui" {
  themes: cupcake;
}
```

Additionally, we must update any <html> elements for pages using the theme, ie: your "view" (".html") files to use the "data-theme" property:

```
<html data-theme="cupcake">
```

## Components

At the time of writing, DaisyUI ships with a total of 52 Components. The library is extremely extensive and the documentation is both very well written and *searchable*. If you plan on working with Tailwind CSS, daisyUI is an excellent addition that can greatly speed up your development time and make your apps look professional without having to do any additional design work.

The following is a list of a few *key* components that are used in most web applications, along with the (.html) code to include them in your views. For additional components and patterns, refer to the official documentation.

> ⚠️ **CAUTION**
>
> Do not forget to "build" your Tailwind CSS before testing newly added

**Navbar**

The first component that we will discuss is the "Navbar". The navbar (short for "navigation bar") provides a consistent, user friendly and *widely recognized* way to navigate through a web site / app. To begin using daisyUI's implementation, add the following HTML:

```
<div class="navbar bg-base-100">
  <div class="flex-1">
    <a class="btn btn-ghost normal-case text-xl">daisyUI</a>
  </div>

  <div class="navbar-center flex">
    <ul class="menu menu-horizontal px-1">
      <li><a>Link</a></li>
      <li>
        <details>
          <summary>Parent</summary>
          <ul class="p-2 bg-base-100 right-0">
            <li><a>Link 1</a></li>
            <li><a>Link 2</a></li>
          </ul>
        </details>
      </li>
    </ul>
  </div>
</div>
```

This is a very simple navigation bar using boilerplate code from the documentation. It shows a "daisyUI" logo / link on the left hand side and two navigation links (including a "dropdown" menu). To add new items, we simply

modify the unordered list. Features such as "Search Input" and "Icons with indicators" are also supported.

The only issue here is that the menu is not "responsive" (ie: it does not collapse to accommodate smaller devices). To remedy this, we should make the navbar *only visible* if the viewport is a certain width or *larger*. If the viewport is smaller than the breakpoint (minimum width), then an alternate navbar should be shown, featuring an icon indicating that the user can click to view the menu items.

To modify the above boilerplate code to make it responsive, we must first change:

```
<div class="navbar-center flex"></div>
```

to:

```
<div class="navbar-center hidden sm:flex"></div>
```

Next, add the "responsive" version of the navigation bar *above* the recently-modified "navbar-center" <div>...</div>, making sure to include the same links / menu items:

```
<div class="dropdown">
  <label tabindex="0" class="btn btn-ghost sm:hidden">
    <svg xmlns="http://www.w3.org/2000/svg" class="w-6 h-6"
fill="none" viewBox="0 0 24 24" stroke="currentColor"><path
stroke-linecap="round" stroke-linejoin="round" stroke-width="2"
d="M4 6h16M4 12h16m-7 6h7" /></svg>
  </label>
  <ul tabindex="0" class="menu menu-sm dropdown-content right-0
```

This should provide a navigation bar that appears normal if the viewport is larger than the "sm" size (640px), and compressed (ie: converted to a drop-down menu with an appropriate icon) when the viewport is smaller.

> ⚠️ **CAUTION**
>
> There is currently an issue where dropdowns do not close when clicking away on some mobile devices (iPhone / IOS) - see: "Dropdown not closing when clicking outside on mobile #824". At the time of writing, the fix involves setting a negative `tabindex` on the `<body>` element, ie: `<body tabIndex="-1">`

## Grid System

Another important feature of any design system is it's "grid" implementation. A grid system will let us place elements on the page that are spaced consistently and are resized / rearranged to accommodate multiple device sizes (ie: "responsive"). Interestingly, daisyUI does not provide it's own grid system, instead relying on Tailwind's excellent implementation.

To get started using the grid system, we will crate a responsive grid that consists of 4 columns for the large size, 2 columns for the medium size, and 1 column for the small size.

```
<div class="container mx-auto">
  <div class="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-4 gap-4">
    <div class="border-2">01</div>
    <div class="border-2">02</div>
    <div class="border-2">03</div>
    <div class="border-2">04</div>
    <div class="border-2">05</div>
  </div>
```

You will notice that the grid is also placed within a responsive "container", which ensures that the grid is given a correct width depending on the viewport size.

**Cards**

A "Card" is basically a user interface element that serves as a "content container" for a specific item to be presented to the user (ie: a product from a store, or service offered, etc). Cards typically include elements such as an image, title, description, call to action, and often incorporate subheadings or icons.

"A card UI design is an entire interface based largely or exclusively on

presenting the user content on cards. The logic behind this is to avoid long texts and render content more scannable. Even though users might not be familiar with the concept of a card from a design point of view – they instantly know how to use UI cards."

https://www.justinmind.com/ui-design/cards

Fortunately, daisyUI has a card component that is ready to be incorporated into our designs:

```html
<div class="card w-96 bg-base-100 shadow-xl">
  <figure>
    <img class="w-full" src="https://placehold.co/
375x375?text=[+SHOES+]" alt="Shoes" />
  </figure>
  <div class="card-body">
    <h2 class="card-title">Shoes!</h2>
    <p>If a dog chews shoes whose shoes does he choose?</p>
    <div class="card-actions justify-end">
      <button class="btn btn-primary">Buy Now</button>
    </div>
  </div>
</div>
```

> ⓘ **INFO**
>
> If you wish to position the card as a "grid" item, the "w-96" class can be removed and the entire "card" can be placed within the grid

For other options, such as "responsive", "custom colors", etc. please refer to the documentation.

**Tables**

If your content is more "tabular" (ie: displayed using tables / columns), such as sports scores, results from an experiment, sales reports, etc. then it's best placed with a styled "table" element.

To achieve this using daisyUI, the documentation recommends placing your table within a `<div>` element with class `overflow-x-auto`. Additionally, the `<table>` element itself should have the class `table`:

```
<div class="overflow-x-auto">
  <table class="table">
    <!-- ... -->
  </table>
</div>
```

For other options, such as "striped rows", "visual elements", "compact tables", etc. please refer to the documentation.

**Forms**

Finally, we should discuss how form elements are styled using Tailwind CSS / daisyUI. Generally, form controls are styled using a class that matches their type - for example, to style an `<input>` element the class "input" would be used. Similarly, to style a `<select>` element the "select" class would be used. This pattern extends to whether or not we wish to style the control with a border (ie, using the styles "input-bordered" and "select-bordered").

The below HTML snippet captures the major form types: "input", "textarea", "select", "radio" and "checkbox" within a responsive grid that is 3 columns wide.

```
<form>
  <div class="container mx-auto">
    <div class="grid grid-cols-1 md:grid-cols-3 gap-4">
      <div>
        <label class="label"><span class="label-
text">Name</span></label>
        <input
          class="input input-bordered w-full"
          type="text"
          id="name"
          name="name"
          placeholder="Enter your name"
        />
      </div>
      <div>
        <label class="label"><span class="label-
text">Message</span></label>
        <textarea
          class="textarea textarea-bordered w-full"
          id="message"
          name="message"
          placeholder="Enter your message"
        ></textarea>
      </div>
      <div>
        <label class="label"><span class="label-text">Fast
Food</span></label>
        <select class="select select-bordered w-full">
          <option disabled selected>Fast Food</option>
          <option>Pizza</option>
          <option>Hamburger</option>
        </select>
      </div>
      <div>
        <label class="label cursor-pointer justify-normal">
          <input type="radio" name="pets" class="radio" /> <span
class="label-text px-2">Dogs</span>
```

# Example Code

You may download the sample code for this topic here:

UI-Toolkits

# 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`):

```javascript
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)

> ⚠️ **INFO**
>
> 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;
```

> ⚠️ **INFO**
>
> 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');
});
```

> ⚠ **CAUTION**
>
> 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:

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

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

## 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:

```javascript
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):

```javascript
// 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:

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

app.use((err, req, res, next) => {
  res.status(500).send(`500 - ${err.message}`);
});
```

# 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:

```javascript
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").

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

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

> ⓘ **INFO**
>
> 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.

# 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
```

```javascript
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

# 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 }
```

> ⊙ **INFO**
>
> For more information on how values are "stringified", refer to the MDN documentation on "JSON.stringify()"

# AJAX Review

**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

# AJAX Introduction: 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://jsonplaceholder.typicode.com/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:

```javascript
let myRequest = new Request('https://jsonplaceholder.typicode.com/users/', {
  method: 'POST',
  body: JSON.stringify({ username: 'jDoe', name: 'John Doe' }),
  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).

```javascript
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:

```javascript
fetch('https://jsonplaceholder.typicode.com/users/', {
  method: 'POST',
  body: JSON.stringify({ username: 'jDoe', name: 'John Doe' }),
  headers: { 'Content-Type': 'application/json' },
})
  .then((response) => response.json())
  .then((json) => {
    console.log(json);
  });
```

> ⓘ **INFO**
>
> Our code is even shorter if we're simply doing a "GET" request, ie:

```
fetch('https://jsonplaceholder.typicode.com/users/')
  .then((response) => response.json())
  .then((json) => {
    console.log(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://jsonplaceholder.typicode.com/unknown')
  .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.

> ⚠️ **CAUTION**
>
> To truly create a *fully compliant* **REST API** we must conform to the standards outlined in Roy Fielding's PhD dissertation, <u>Architectural Styles and the Design of Network-based Software Architectures</u>. 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:

```javascript
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' });
```

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'));
});
```
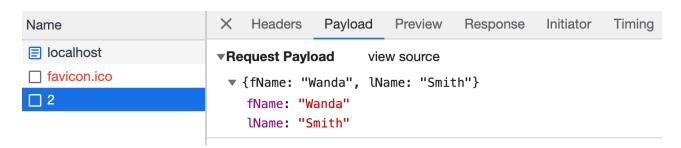
Finally - our server is setup and ready to serve the index.html file at our main route ("/"). Our next step is to add our client-side logic / JS to the index.html file. Here, we hard-code some requests to the API and output their results to

the web console to make sure they function correctly:

```html
<!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:

| Name | ✕ Headers Payload Preview Response Initiator Timing |
|---|---|
| 🗐 localhost | ▼**Request Payload**        view source |
| ☐ favicon.ico | ▼ {fName: "Wanda", lName: "Smith"} |
| ☐ 2 | fName: "Wanda" |
| | lName: "Smith" |

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.

> ⚠ **CAUTION**
>
> If we want to allow the API to respond to requests from *outside* the domain (this is what https://jsonplaceholder.typicode.com 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

# 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:

```javascript
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):

```javascript
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:

```json
{ "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:

```javascript
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":

```html
<!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: "<br />" will be rendered as "&lt;br /&gt;" 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: `&lt;br /&gt;`, 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 %>
<% }) %>
```

> **ⓘ INFO**
>
> 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)

> ⓘ **INFO**
>
> 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>
    <td><%= data.company %></td>
  </tr>
<% } else { %>
  <tr>
    <td colspan="4">User: '<%= data.name %>' has hidden their
information</td>
  </tr>
<% } %>
```

## 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>
  </tr>
<% }) %>
```

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,
  },
  {
    name: 'Sarah',
    age: 32,
    occupation: 'manager',
    company: 'TD',
    visibility: true,
  },
];
```

**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
```

# 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:

```html
<h1>EJS Practice - <%= page %></h1>
<hr />
  <a href="/"><%- (page=="/") ? '<strong>Home</strong>' : 'Home'
%></a> |
  <a href="/about"><%- (page=="/about") ? '<strong>About</strong>'
: 'About' %></a> |
  <a href="/viewData"><%- (page=="/viewData") ? '<strong>View
Data</strong>' : 'View Data' %></a>
<hr />
<br />
```

> ⚠ **INFO**
>
> 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

# HTML Form Elements Review

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*.

```html
<form method="post" enctype="application/x-www-form-urlencoded" action="https://httpbin.org/post">
  <!-- ... -->
</form>
```

> ⊙ **INFO**
>
> 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" />
```

> ⚠️ **CAUTION**
>
> 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"]`.

```html
<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" />
```

# 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".

```html
<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".

```html
<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:

```html
<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"

> ⊕ **INFO**
>
> 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.
>
> 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:

```js
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 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:
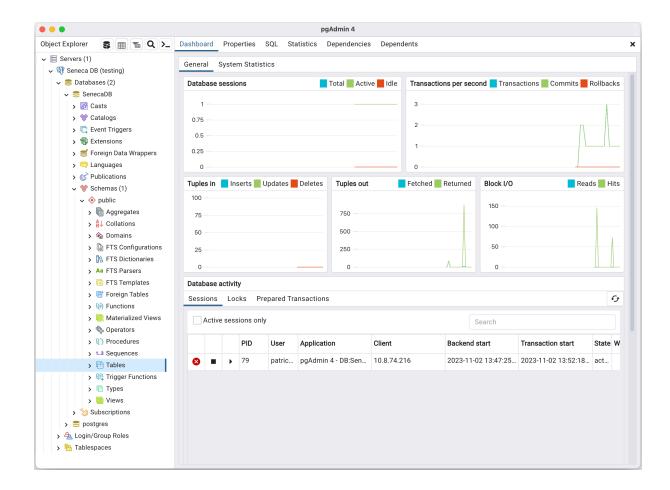
- [https://www.pgadmin.org/download/]()

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:

```javascript
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!');
    })
```
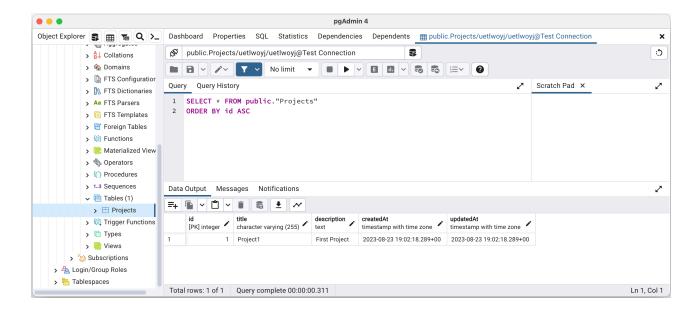
Once again, **database** is your randomly generated "User & Default database" value, **user** is also your randomly generated "User & Default database" value, **password** is your password and lastly, **host** will be your server host url.

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).

> ⓘ **INFO**
>
> 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 <span style="color:red">list of types</span>, 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:

```js
// 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
});
```

> ⓘ **INFO**
>
> 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:

# 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:

```javascript
// 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 Task with User & automatically create a foreign key
// relationship on "Task" via an automatically generated "UserId"
field
```

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:

```javascript
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.

> **① INFO**
>
> 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:

```javascript
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**.

> ⚠️ **CAUTION**
>
> 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:

```javascript
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);
    }
  });
});
```

> ⓘ **INFO**
>
> See the documentation for advanced queries and fetching associated

# Update

To **update** existing names in our **Name** table, we can use the following code:

```javascript
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

# Introduction to MongoDB

MongoDB is an open source database that stores its data as collection of JSON like documents known as BSON or "Binary JSON", instead of tables with rows / columns. Technically, it is classified as a "NoSQL" database - a popular alternative to traditional Relational Databases (RDBMS):

NoSQL ("non SQL" or "not only SQL") databases were developed in the late 2000s with a focus on scaling, fast queries, allowing for frequent application changes, and making programming simpler for developers. Relational databases accessed with SQL (Structured Query Language) were developed in the 1970s with a focus on reducing data duplication as storage was much more costly than developer time. SQL databases tend to have rigid, complex, tabular schemas and typically require expensive vertical scaling.

Some of the main benefits include:

- **Flexible data models**

    NoSQL databases typically have very flexible schemas. A flexible schema allows you to easily make changes to your database as requirements change. You can iterate quickly and continuously integrate new application features to provide value to your users faster.

- **Horizontal scaling**

    Most SQL databases require you to scale-up vertically (migrate to a larger, more expensive server) when you exceed the capacity requirements of your current server. Conversely, most NoSQL databases allow you to scale-out horizontally, meaning you can add cheaper commodity servers whenever you need to.

- **Fast queries**

    Queries in NoSQL databases can be faster than SQL databases. Why? Data in SQL databases is typically normalized, so queries for a single object or entity require you to join data from multiple tables. As your tables grow in size, the joins can become expensive. However, data in NoSQL databases is typically stored in a way that is optimized for queries. The rule of thumb when you use MongoDB is **data that is accessed together should be stored together**. Queries typically do not require joins, so the queries are very fast.

- **Easy for developers**

    Some NoSQL databases like MongoDB map their data structures to those of popular programming languages. This mapping allows developers to store their data in the same way that they use it in their application code. While it may seem like a trivial advantage, this mapping can allow developers to write less code, leading to faster development time and fewer

> bugs.
>
> https://www.mongodb.com/nosql-explained

# NoSQL vs Traditional SQL

As we have seen, one of the major differences between NoSQL and traditional SQL systems is the way the data is *structured*, ie: SQL databases are table-based. This means they use a rigid schema where data is organized into tables with rows and columns and primary / foreign keys to establish relationships between them. NoSQL databases however, can have different structures such as document-oriented (in the case of MongoDB), key-value pairs, or graph structures. In a NoSQL database, a document can contain key-value pairs and can be ordered and nested. Ths leads to additional benefits mentioned above, such horizontal scaling and fast queries.

Before we get started with MongoDB, we should be familiar with how some of the terms translate to traditional RDBMS:

| RDBMS term | MongoDB term |
|---|---|
| Table | Collection |
| Record | Document |
| Column | Field |
| Joins | Embed data or link to another collection |

See: MongoDB vs. MySQL Differences

# Setting up a MongoDB Atlas account

MongoDB Atlas is a **free** online service that hosts MongoDB in the cloud:

> MongoDB Atlas is a multi-cloud database service by the same people that build MongoDB. Atlas simplifies deploying and managing your databases while offering the versatility you need to build resilient and performant global applications on the cloud providers of your choice.
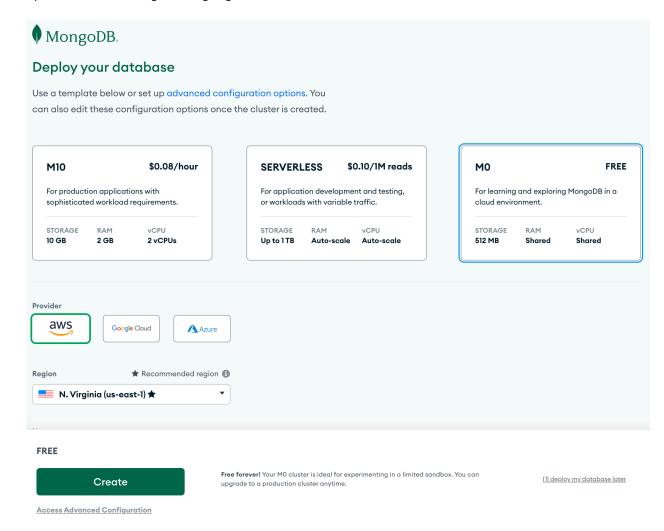>
> https://www.mongodb.com/docs/atlas

To get started, open https://www.mongodb.com/cloud/atlas and click the "Try free" button.

This will take you to the "register" page, where you can either create an account with Atlas, or sign in with Google. If you prefer to sign in with GitHub, you can proceed directly to: https://account.mongodb.com/account/login and click the "GitHub" button.

Once you have logged in, you should be prompted to **"Create a deployment"**. To begin:

- Click the **"+ Create"** button to continue.

- At the next screen, we will see the **"Deploy your database"** options. Be sure to Choose the **"FREE"** option before clicking the large, green **"Create"** button:



- You will then be taken to the **"Security Quickstart"** screen, which should have the "Username and Password" option checked with a form containing pre-filled values for the "Username" and "Password". **Write these down** as we will need them later, and click the green **"Create User"** button

- With this section complete, you should be taken to a **"Where would you like to connect from?"**
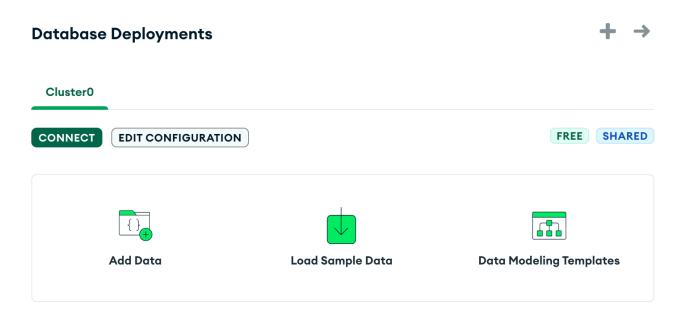
screen with "My Local Environment" selected and an "IP Access List". For now, we will *allow access from anywhere*, so ensure that you enter the following values "0.0.0.0/0" and "any" before clicking the "Add Entry" button:

**IP Address**

| 0.0.0.0/0 |

**Description**

| any |

| Add My Current IP Address |

| Add Entry |

- Finally, click the **"Finish and Close"** button

# Obtaining your Connection String

Once your cluster has been created, you should be taken to the "Overview" screen, where we can view our deployments.

**Database Deployments**                                    ➕  ➡️

**Cluster0**

| CONNECT |   | EDIT CONFIGURATION |                          **FREE**   **SHARED**

**Add Data**                    **Load Sample Data**              **Data Modeling Templates**

- From here, click the **"CONNECT"** button and choose the **"Drivers"** option

## Connect to your application



**Drivers**
Access your Atlas data using MongoDB's native drivers (e.g. Node.js, Go, etc.)

- Scroll down to **"3. Add your connection string into your application code"** and **write down** the connection string (we will need it later) - it should look something like:

```
mongodb+srv://user:<password>@cluster0.abc123.mongodb.net/?retryWrites=true&w=majority
```

- Finally, replace the <password> value with the password recorded from earlier, ie:

`mongodb+srv://user:yourPassword@cluster0.abc123.mongodb.net/?retryWrites=true&w=majority`

and click the **"Close"** button to return to the "Overview" screen.

# Mongoose ODM with MongoDB

When we work with MongoDB in Node, we won't work directly with the MongoDB driver. Instead, we will use a popular open source module called "Mongoose" - an ODM ("Object Data Modeling") library that serves as a wrapper for the Mongo driver and provides extra functionality:

> "Mongoose provides a straight-forward, schema-based solution to model your application data. It includes built-in type casting, validation, query building, business logic hooks and more, out of the box.".
>
> https://mongoosejs.com

To begin working with Mongoose, we need to retrieve it from NPM:

```
npm install mongoose
```

and 'require' it in our code:

```js
const mongoose = require('mongoose');
```

## Mongoose Schemas

Before we look at how to establish a connection to our MongoDB Atlas DB and work with the data using Mongoose, let's first determine the type of data that we wish to store. For example, let's say that our application requires "company" information to be persisted. Each "company" used by our system can be represented using the following properties (ie, its "shape"), as illustrated below for "The Kwik-E-Mart":

```js
{
  companyName: "The Kwik-E-Mart",
  address: "Springfield",
  phone: "212-842-4923",
  employeeCount: 3,
  country: "U.S.A"
}
```

To begin working with "companies" like this in our database using Mongoose, the first step is to create a "schema".

### Creating a Schema

From the documentation: "Everything in Mongoose starts with a Schema. Each schema maps to a MongoDB

collection and defines the shape of the documents within that collection". So, for us to work with a specific collection in our MongoDB database, we must first define a "schema", which defines the structure of the documents to be added to the collection (as well as to provide other features such as "validators", etc.).

To represent the above company data as a Mongoose Schema, we can use the following code:

```
const mongoose = require('mongoose');
let Schema = mongoose.Schema;

let companySchema = new Schema({
  companyName: String,
  address: String,
  phone: String,
  employeeCount: {
    type: Number,
    default: 0,
  },
  country: String,
});

let Company = mongoose.model('companies', companySchema);
```

Essentially, a schema is like a blueprint for a document that will be saved in the DB. Here, we define the fields that can exist on a document for this collection, and setting their expected types, default values, and sometimes if they are required, or have an index on them.

In the above code, we have defined a Company schema with 5 properties as discussed, and set their types appropriately. The employee count is not just a simple number, we also want to include a default value of 0 of the count field is not supplied. Using defaults where it makes sense to have them is good practice.

The last line of code tells mongoose to register this schema (companySchema) as a model and connect it to the companies collection (Note: the "companies" collection will be automatically created if it doesn't exist yet). We can then use the Company variable to make queries against this collection and insert, update, or remove documents from the Company model.

## Unique Index

A unique index may also be applied at the database level and can be attached to one or more fields of a document.

The most common use for this is when we want to enforce a unique value across all documents in a collection for a certain field. A perfect use case for this is the companyName field of our company schema, ie: it wouldn't make sense to have multiple companies with the same name in the system. To prevent this and add a unique index in to the companyName field, we just have to add `unique: true` to the schema declaration from before.

```
// define the company schema
const companySchema = new Schema({
  companyName: {
    type: String,
    unique: true,
  },
  address: String,
  phone: String,
  employeeCount: {
    type: Number,
    default: 0,
  },
  country: String,
});
```

> ⚠ **INFO**
>
> With the "unique: true" property set on the "companyName" field, Mongoose will return "E11000 duplicate key error" if we try to save two companies with the same "companyName" field.

# Adding Data

Now that we have determined the "schema", let's see how Mongoose works to add our data ("The Kwik-E-Mart") to the database.

> ⚠ **CAUTION**
>
> For the below code to function correctly, you will need to place your connection string (determined earlier in Introduction to MongoDB), in place of the `'Your connection string here'`. You will **also** have to **update** it to include a database name. For example, if your connection string looks like the following:
>
> ```
> mongodb+srv://user:yourPassword@cluster0.abc123.mongodb.net/?retryWrites=true&w=majority
> ```
>
> You must update it to include a **database** name so that the default name: "test" is not used. For example, if you wish your database to be called "demo", you would update the connection string to include "demo" after "mongodb.net/", ie:
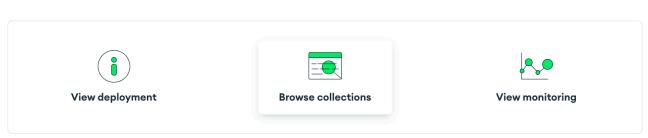>
> ```
> mongodb+srv://user:yourPassword@cluster0.abc123.mongodb.net/
> demo?retryWrites=true&w=majority
> ```

```javascript
// require mongoose and setup the Schema
const mongoose = require('mongoose');
let Schema = mongoose.Schema;

// connect to Your MongoDB Atlas Database
mongoose.connect('Your connection string here');

// define the company schema
let companySchema = new Schema({
  companyName: String,
  address: String,
  phone: String,
  employeeCount: {
    type: Number,
    default: 0,
  },
  country: String,
});
// register the Company model using the companySchema
// use the companies collection in the db to store documents
let Company = mongoose.model('companies', companySchema);

// create a new company
let kwikEMart = new Company({
  companyName: 'The Kwik-E-Mart',
  address: 'Springfield',
  phone: '212-842-4923',
  employeeCount: 3,
  country: 'U.S.A',
});

// save the company
kwikEMart
  .save()
  .then(() => {
    console.log('The Kwik-E-Mart company was saved to the companies collection');
    process.exit();
  })
  .catch((err) => {
    console.log('There was an error saving the Kwik-E-Mart company');
    process.exit();
  });
```

# Reading Data

To confirm that our data was indeed added to the database, we can either log back in to MongoDB Atlas and click "Browse collections" for our cluster:

**CONNECT**   EDIT CONFIGURATION                                              FREE   SHARED

**Data Size: 40.44 KB**



**View deployment**          **Browse collections**          **View monitoring**

or we can query the data using Mongoose to confirm that it was entered correctly. Here, we will use the same code as above (being sure to include the code to create the "Company" object), except we can omit creating & saving a new "Company" (ie: "kwikEMart"). Instead, we will use the "find" method on the "Company" model to locate our "The Kwik-E-Mart" company:

```javascript
Company.find({ companyName: 'The Kwik-E-Mart' })
  .exec()
  .then((company) => {
    if (!company) {
      console.log('No company could be found');
    } else {
      console.log(company);
    }
    // exit the program after saving and finding
    process.exit();
  })
  .catch((err) => {
    console.log(`There was an error: ${err}`);
    process.exit();
  });
```

> ⓘ **INFO**
>
> If you examine the output, you will notice that the data returned includes two extra fields, added by default to our document:
>
> • _id: A unique ObjectID
> • __v: The versionKey

## .exec()

The .exec() call is added after a mongoose query to tell mongoose to return a promise. If you leave out the .exec(), mongoose will still work with .then() calls but the object returned will not be a proper promise. It is good practice to always use .exec() after your query has been setup and before the .then() method is invoked.

# Arrays and Recursive Schemas

A "recursive schema" is a schema that contains an array of elements with the same schema as the definition. We can use this to store tree structures such as file / folder hierarchies or comment trees for a blog post. For example: say we wish to store a tree of comments, where each comment can have one or more comments, which can have one or more comments, and so on. We can specify our recursive "commentSchema" using the following code:

```javascript
const commentSchema = new Schema({
  comment: String,
  author: String,
  date: Date,
});

commentSchema.add({ comments: [commentSchema] });

let Comment = mongoose.model('comments', commentSchema);
```

Here, we add a "comments" field with a type of "[commentSchema]" to the original "commentSchema". Using this syntax, we indicate that all "comments" will consist of an Array defined by "commentSchema". Now, we can easily create documents that appear in this format, ie:

```javascript
let commentChain = new Comment({
  comment: 'Star Wars is awesome',
  author: 'Author 1',
  date: new Date(),
  comments: [
    {
      comment: 'I agree',
      author: 'Author 2',
      date: new Date(),
      comments: [
        {
          comment: 'I agree with Author 2',
          author: 'Author 3',
          date: new Date(),
          comments: [],
```

# Multiple Connections

Using Mongoose, it is also possible to have multiple connections configured for your application. If this is the case, we have to make a few small changes on how we **connect** to each DB, and how we define our models

> ⚠️ **CAUTION**
>
> The use of the "encodeURIComponent" is necessary if your password contains special characters, ie "$"

```js
// ...

let pass1 = encodeURIComponent('pa$$word1'); // this step is needed if there are special
characters in your password, ie "$"
let db1 = mongoose.createConnection(
  `mongodb+srv://user:${pass1}@cluster0.abc123.mongodb.net/demo?retryWrites=true&w=majority`
);

// verify the db1 connection

db1.on('error', (err) => {
  console.log('db1 error!');
});

db1.once('open', () => {
  console.log('db1 success!');
});

// ...

let pass2 = encodeURIComponent('pa$$word2'); // this step is needed if there are special
characters in your password, ie "$"
let db2 = mongoose.createConnection(
  `mongodb+srv://dbUser:${pass2}@cluster0.2def3.mongodb.net/db2?retryWrites=true&w=majority`
);

// ...

let model1 = db1.model('model1', model1Schema); // predefined "model1Schema" used to create
"model1" on db1

let model2 = db2.model('model2', model2Schema); // predefined "model2Schema" used to create
"model2" on db2

// ...
```

Instead of using **"connect"**, we instead use **"createConnection"** and save the result as a reference to the

connection (ie: **"db1"** and **"db2"** from above). We can then use **db1** or **db2** to create models on each database separately. Additionally, if we want to *test* the connection, we can use the **.on()** and **.once()** methods of each connection.

# Operations (CRUD) Reference

Once again, we will be discussing the four major operations typically performed on data: **C**reate, **R**ead, **U**pdate and **D**elete (**CRUD**). The operations in the code below will each work with the familiar "companySchema" using the "Company" model:

```
let companySchema = new Schema({
  companyName: String,
  address: String,
  phone: String,
  employeeCount: {
    type: Number,
    default: 0,
  },
  country: String,
});

let Company = mongoose.model('companies', companySchema);
```

## Create

To "save" (create) a new document, we must first create the document in code using the model we want (ie: "Company"). Then we can call a built in method, "save" on the new object to save it.

```
const kwikEMart = new Company({ ... });
```

# Read

To "find" (read) documents from the database, we use the "find" method on the model object itself (ie: "Company"), ie:

```
Company.find({ companyName: 'The Kwik-E-Mart' })
  //.sort({}) //optional "sort" - https://docs.mongodb.com/manual/
reference/operator/aggregation/sort/
  .exec()
  .then((companies) => {
    // companies will be an array of objects.
    // Each object will represent a document that matched the
query
    console.log(companies);
  });
```

## Selecting specific fields

If we wish to limit the results to include only specific fields, we can pass the list of fields as a space-separated string in the second parameter to the find() method, ie:

```
Company.find({ companyName: 'The Kwik-E-Mart' }, 'address phone')
  //.sort({}) //optional "sort" - https://docs.mongodb.com/manual/
reference/operator/aggregation/sort/
  .exec()
  .then((companies) => {
    // companies will be an array of objects.
    // Each object will represent a document that matched the
query
    console.log(companies);
```

For complex queries (ie: "greater than", "in", "or", etc, etc.) see the Mongoose Query Guide and the MongoDB documentation under Query and Projection Operators

# Update

To update documents in the database, we use the updateOne() / updateMany() methods on the model object (ie: "Company"). We typically pass these function two arguments: the query to select which documents to update and the fields to set for the documents that match the query.

> ⓘ **INFO**
>
> See update operators, ie: $set, $push and $addToSet for more information.

```
Company.updateOne( { companyName: 'The Kwik-E-Mart' }, { $set: {
employeeCount: 3 } })
  .exec()
  .then(() => {
    // updated company
    console.log('updated company');
  })
  .catch((err) => {
    console.log(err);
  });
```

# Delete

To delete documents in the database, we use the deleteOne() / deleteMany() methods on the model object (ie: "Company").

```javascript
Company.deleteOne({ companyName: 'The Kwik-E-Mart' })
  .exec()
  .then(() => {
    // removed company
    console.log('removed company');
  })
  .catch((err) => {
    console.log(err);
  });
```

# Example Code

You may download the sample code for this topic here:

> NoSQL-Database-MongoDB

# Key Terminology

When we talk about "State Management" in an Express application, we generally refer to the process of maintaining and managing user data *between* requests in a "session". For example, this may involve keeping track of whether a user has logged in, or what items are in their shopping cart. Our application needs to know the status (ie: "state") of the client so that it can respond appropriately. Has the user previously been authenticated? What are the product id's of the items in their cart?

To enable this functionality within our applications, we must work with:

1. Cookies
2. Sessions
3. Encryption

We are going to cover Cookies and Sessions below and briefly touch on encryption as it applies to cookies and sessions. In "Security Considerations", we will go into more detail on encryption for other purposes.

## Cookies

Recall from Advanced Routing & Middleware, "cookies" are pieces of data that are passed back and forth from the browser to the server that hold state information about the current audience interacting with your application.

Each time a request is made from the browser or a response is sent from the server, a set of headers is included. Request headers contain the **'Cookie'** header and response headers contain the **'Set-Cookie'** header. These headers are a string of semicolon separated values that can be referenced in server side code using the 'req' object. The most common type of data we want to place in the cookie is a "session" value.

Here is an example of what a cookie might look like in the request header when inspecting it in the Chrome dev tools

> **Cookie:** COOKIE_CONSENT_ACCEPTED=true; PS_DEVICEFEATURES=width:1680 height:1050 pixelratio:2 touch:0 geolocation:1 websockets:1 webworkers:1 datepicker:1 dtpicker:1 timepicker:1 dnd:1 sessionstorage:1 localstorage:1 history:1 canvas:1 svg:1 postmessage:1 hc:0 maf:0;
> AWSELB=25B9EB610A4727BBBAAA553BD60CC37D8297F3411BEB083D3A756E7C927A16B55DE1AF9292A34C533329A16DEEFAB2 D1F0A8885F83FB98BB17D96810C5F56F19DD91CE2710;
> AWSELBCORS=25B9EB610A4727BBBAAA553BD60CC37D8297F3411BEB083D3A756E7C927A16B55DE1AF9292A34C533329A16DEE FAB2D1F0A8885F83FB98BB17D96810C5F56F19DD91CE2710; BbClientCalenderTimeZone=America/Toronto;
> JSESSIONID=30A73795E59C58AA9DE10E9A55611D84;
> samlCookie=33323A4F65773352327570346E792F7138547478616F417A38613172304E6A797A517065457A74684F574E746F476C5 9412B474F4E7963465151695275304D49526E72;
> BbRouter=expires:1692983209,id:FA96503450967749E204905096629DDA,sessionId:4644480316,signature:b0f5dc80530391a6aaa 921cf7e2f29bc18af9e19b62e6f9a34d5287cb910f86c,site:47804be6-4f5a-41e6-9752-f1324b876acb,timeout:10800,user:94048796fa d9425cb85b38fe6cc9a794,v:2,xsrf:a2aaa822-9f18-49ef-b62c-911a845d78c5

And here is an example of the 'Set-Cookie' header in a response from the server to update it after a request

> **Set-Cookie:**
> BbRouter=expires:1692983219,id:FA96503450967749E204905096629DDA,sessionId:4644480316,signature:7aa65c9e8d01384c968 97743bc1ef4ff919b0342f37c8f8e0525c6566aa411fb,site:47804be6-4f5a-41e6-9752-f1324b876acb,timeout:10800,user:94048796fa d9425cb85b38fe6cc9a794,v:2,xsrf:a2aaa822-9f18-49ef-b62c-911a845d78c5; Path=/; Secure; HttpOnly

Notice how the 'Cookie' header contains a session ID and digital signature. It may also contain other elements depending on what the app intends to retain in the 'state' between user requests. In the above case, an 'encrypted session' is utilized (which is why we're unable to read it). This is **extremely** important in a production environment, as we do not want sensitive user information to be compromised if the cookie is accessed by a third party. As a general rule, all sessions that persist between the client and server should be encrypted and transmitted exclusively over HTTPS. We will cover HTTPS encryption in "Security Considerations".

# Sessions

Implementing sessions is fairly straightforward in Express.js. The platform is mature and libraries have been tested with thousands of websites and billions of logins. These libraries are easy to use and integrate into your own projects.

A popular library for implementing client sessions is Mozilla's "client sessions" Node library. This library focuses on keeping sessions between the client and server on the *client*. This has several advantages over storing and keeping track of them in memory on the server. For example, if the server restarts and you have not yet saved session information in a persistent storage location, it will be lost and all your users will be logged out.

It is also beneficial to keep your sessions on the client (and have them continuously sent with each request), as this enables you to host a website or app with multiple web servers, while ensuring the session remains active regardless of the server used to process the response. If this were not the case, we must ensure each users' requests are always sent to the same web server to preserve their session. Alternatively, you could attempt to replicate server session information between web servers, however this can be very complex. For these reasons, storing the session on the client makes scaling and session management a lot easier.

# Authentication vs Authorization

Authentication and Authorization mean two entirely different things. It can be easy to confuse them, so let's discuss them a little bit before we begin to implement sessions and secure routes. This will enable you to be more comfortable explaining and debugging the two different concepts.

Authentication is the answer to "Who are you?". It involves supplying credentials to identify yourself to the server and establish a session for your user account.

Authorization is the answer to "What do you have access to?". It involves checking your permissions to resources you have requested and acting accordingly. You may be authenticated with the server and have a user session but you might not be authorized to view a certain resource. (No permissions!)

Here is quick video from MongoDB University that explains it nicely. MongoDB also has the concept of authentication and authorization.

# Status codes

There are a number of standard http response status codes that can be used by your application to inform the browser of whether a request was rejected because of an authentication problem or an authorization problem.

- **401 (Unauthorized)**: Authentication error. The resource exists but it requires the user to be authenticated first to view it. It may also require permissions and be checked again after authenticating for proper authorization.

- **403 (Forbidden)**: Authorization error. The resource exists but the user does not have permission to view it.

- **404 (Not Found)**: The resource that was requested was not found on the server. This is commonly used when a url is requested that simply doesn't exist.

# Introduction to "Client Sessions"

We have established that there are multiple benefits to storing "session" data on the client in an encrypted cookie:

> - The data is always available, regardless of which machine is serving a user
> - There is no state to manage on servers
> - Nothing needs to be replicated between the web servers
> - New web servers can be added instantly
>
> "Using secure client-side sessions to build simple and scalable Node.JS applications"

Additionally, we have seen that this technology is widely tested and has been made available via the "client sessions" Node library. In the following sections, we will see how we can implement and test this library in our servers.

## The "client-sessions" Library

The "client-sessions" library is available on NPM and can be included in our project using the familiar steps to install:

```
npm install client-sessions
```

and 'require' it in our code

```js
const clientSessions = require('client-sessions');
```

### Middleware

Once we have a reference to "clientSessions", we register it as middleware and configure it using the "cookieName", "secret", "duration" and "activeDuration" properties:

```js
app.use(
  clientSessions({
    cookieName: 'session', // this is the object name that will be added to 'req'
    secret: 'o6LjQ5EVNC28ZgK64hDELM18ScpFQr', // this should be a long un-guessable string.
    duration: 2 * 60 * 1000, // duration of the session in milliseconds (2 minutes)
    activeDuration: 1000 * 60, // the session will be extended by this many ms each request (1 minute)
  })
);
```

### Testing

To ensure that clientSessions is working properly, add the following simple routes and start the server:

```js
app.get('/session-test-add', (req, res) => {
  req.session.message = req.query.message || ''; // add a "message" property to the session
  res.send("session created with using 'message' query parameter");
});

app.get('/session-test-read', (req, res) => {
  res.send(`session message: ${req.session.message}`); // read the "message" property from the session
});
```

- When you navigate to the "/session-test-add" route with a "message" query parameter, ie:

```
/session-test-add?message=Hello World!
```

You should see that a **"Set Cookie"** header was sent in the response with a value that should look like the following:

```
session=25uFcTuHZzZlSWntEs-Kzg.D96gsJqB0lLKj4DBZsc3KSj4Z4_76pkoCy4uXUqgS1C4uuHbaxMZ6l9dTCWu-
ijc.1692988779453.120000.FprcH5eIT-o6Iedv-vP2i0P8HmzCRMXGdm813oveVBc; path=/; expires=Fri, 25 Aug 2023 18:41:40
GMT; httponly
```

This confirms that our session value was indeed encrypted and sent to the client.

- To test whether or not our server can read it, navigate to the other route:

```
/session-test-read
```

You should see the response text: "session message: Hello World!". Additionally, you should see that a **"Cookie"** header was sent in the request with a value like:

```
session=AqnLAnL7dqAr9QqXnpD5Xw.wcCqiCvVSRgllI1mLOAC9yHmjjLygsur7AQaKX50_9vkugEnTKhhz3V4U8V_xgFa.1692989253302.120000.kCRDEuDgPpEuEsu
```

# Practical Application

We can now confirm that "client sessions" is working correctly - we are able to add values to the session, which are encrypted and sent to the client using a "cookie". As a more practical test of this technology, we will implement a simple app with a "login" view and a protected "dashboard" view that may **only** be accessed once the user has logged in.

To begin, create a simple web server using Express, making sure to also install and configure EJS (see: "Template Engines" - EJS).

## Routes

The server should have three routes:

- **GET "/login"** - renders a "login.ejs" file with an empty "message"

```javascript
app.get('/login', (req, res) => {
  res.render('login', { message: '' });
});
```

- **POST "/login"** - renders a "/login.ejs" file with an "invalid login" message

```javascript
app.post('/login', (req, res) => {
  res.render('login', { message: 'invalid login' });
});
```

- **GET "/dashboard"** - renders a "dashboard.ejs" file with a "user" object from the session

```javascript
app.get('/dashboard', (req, res) => {
  res.render('dashboard', { user: req.session.user });
});
```

## Templates

Next, we must create our two EJS template files: "login.ejs" and "dashboard.ejs" in a **views** directory:

- **views/login.ejs**

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Login</title>
  </head>
  <body>
    <h1>Log In</h1>
    <hr />
    <br />
    <form method="post" action="/login">
      <input type="text" placeholder="User Name" name="userName" />
      <input type="password" placeholder="Password" name="password" />
      <button type="submit">Log In</button>
    </form>
    <br />
    <%= message %>
  </body>
</html>
```

- **views/dashboard.ejs**

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Dashboard</title>
  </head>
  <body>
    <h1>Dashboard</h1>
    <hr />
    <br />
    <h3>Hello <%= user.userName %></h3>
    <p>Welcome to your dashboard</p>
    <p>Here is the information we have on file for you:</p>
    <h4>userName: <%= user.userName %></h4>
    <h4>email: <%= user.email %></h4>
    <a href="/logout">Logout</a>
  </body>
</html>
```

## Middleware

The next part of our application is the **middleware**. We will require:

- The built-in urlencoded middleware:

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

- The client-sessions middleware - installed using NPM and included using:

`const clientSessions = require("client-sessions");`

```javascript
app.use(
  clientSessions({
    cookieName: 'session', // this is the object name that will be added to 'req'
```

- Custom middleware, used to protect a route from unauthorized access:

```js
function ensureLogin(req, res, next) {
  if (!req.session.user) {
    res.redirect('/login');
  } else {
    next();
  }
}
```

## Route Updates (Logic)

The final piece of our application is to add some *logic* to our routes. Specifically, we should add logic to validate the userName / password combination against a predefined (hard-coded) user. If the user is authenticated, a session should be created which gives them access to the "dashboard" route.

To begin, let's add our "ensureLogin" middleware function to the "/dashboard" route to prevent unauthorized access:

```js
app.get('/dashboard', ensureLogin, (req, res) => {
  res.render('dashboard', { user: req.session.user });
});
```

Next, we will update our POST "/login" route to authenticate the "userName" and "password" values from the login form against a mock user. If the credentials match, create a session for the user, otherwise re-render the "login" template with the error message.

```js
app.post('/login', (req, res) => {
  let mockUser = {
    userName: 'sampleuser',
    password: 'samplepassword',
    email: 'sampleuser@example.com',
  };

  if (req.body.userName == mockUser.userName && req.body.password == mockUser.password) {
    req.session.user = {
      userName: mockUser.userName,
      email: mockUser.email,
    };

    res.redirect('/dashboard');
  } else {
    res.render('login', { message: 'invalid login' });
  }
});
```

> **ⓘ INFO**
>
> If we also wish to implement "log out" functionality, we could **reset** the session with the following code:
>
> ```js
> req.session.reset();
> ```

# Example Code

You may download the sample code for this topic here:

Managing-State-Information

# 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.
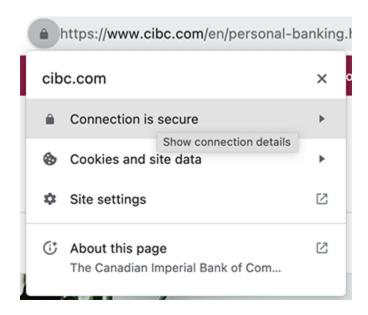
https://www.digicert.com

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".

## Certificate Viewer: www.cibc.com

**General**  Details

### Issued To

| | |
|---|---|
| Common Name (CN) | www.cibc.com |
| Organization (O) | Canadian Imperial Bank of Commerce |
| Organizational Unit (OU) | <Not Part Of Certificate> |

### Issued By

| | |
|---|---|
| Common Name (CN) | DigiCert SHA2 Secure Server CA |
| Organization (O) | DigiCert Inc |
| Organizational Unit (OU) | <Not Part Of Certificate> |

### Validity Period

| | |
|---|---|
| Issued On | Monday, April 10, 2023 at 8:00:00 PM |
| Expires On | Wednesday, April 24, 2024 at 7:59:59 PM |

### Fingerprints

| | |
|---|---|
| SHA-256 Fingerprint | 11 9D DB 55 8B 02 A8 70 23 27 E9 DA 35 68 8F E3 C8 70 13 18 45 10 D7 0C D6 0C B4 07 02 A6 02 52 |
| SHA-1 Fingerprint | 5B 82 B1 7B 61 40 65 36 BC E2 B3 C7 2C 68 15 D6 A0 8B F3 9F |

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: Potential Security Risk Ahead

Firefox detected a potential security threat and did not continue to **localhost**. If you visit this site, attackers could try to steal information like your passwords, emails, or credit card details.

Learn more...

Go Back (Recommended)          Advanced...

---

localhost:4433 uses an invalid security certificate.

The certificate is not trusted because it is self-signed.

Error code: MOZILLA_PKIX_ERROR_SELF_SIGNED_CERT

View Certificate

Go Back (Recommended)          Accept the Risk and Continue

# Warning in Chrome



## Your connection is not private

Attackers might be trying to steal your information from **localhost** (for example, passwords, messages, or credit cards). Learn more

NET::ERR_CERT_AUTHORITY_INVALID

> To get Chrome's highest level of security, turn on enhanced protection

Hide advanced                                              Back to safety

This server could not prove that it is **localhost**; its security certificate is not trusted by your computer's operating system. This may be caused by a misconfiguration or an attacker intercepting your connection.

Proceed to localhost (unsafe)

> ⚠ **CAUTION**
>
> 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:

```javascript
// 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.

> **⚠ CAUTION**
>
> If, at this point, you receive the error: "Git: Failed to execute git", try executing the following commands in the integrated terminal:
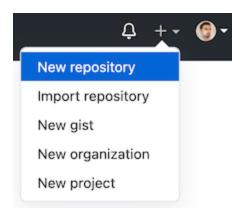>
> ```
> git config --global user.email "you@example.com"
> git config --global user.name "Your Name"
> ```
>
> Once this is complete, attempt to click the checkmark again to commit your changes.

## 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:
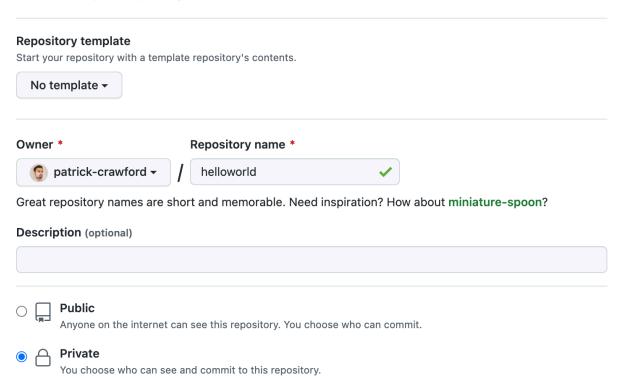
1. Sign in to your GitHub account.

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:
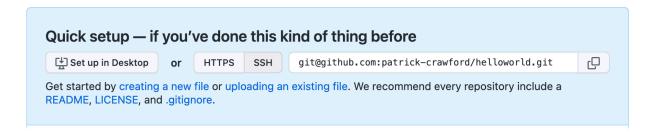


4. Once you're happy with the settings, hit the **"Create repository"** button.

# 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
```

> ⚠️ **CAUTION**
>
> 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.