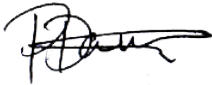




Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

School of Applied Computing

Student Name:	Peter Davies	Student P number:	P079868
Programme:	BSc Web Development		
Module:	Emerging Web Technology	Lecturer:	Mike Dacey
Assignment Title:	Investigating Emerging Web Technologies		
Hand Out Date:	December 2019	Submission deadline:	05/04/2019

Disclosure:	I declare that this assignment is all my own work and that I have acknowledged all materials used from the published or unpublished works of other people. All references have been duly cited.		
Student's Signature:		Date:	05/04/19

Contents

Connection Information	4
Node.js Application.....	4
Ajax Weather API Application.....	4
Database Connection.....	5
1. Introduction	1
2. Knowledge & Understanding Skills	2
2.1 Microsoft .NET Core Blazor.....	2
2.1.1 Web Assembly & MONO.....	4
2.1.2 Blazor Example	6
2.1.3 Conclusion	10
2.2 NoSQL Databases.....	12
2.2.1 Application of NoSQL Database Technology.....	16
2.2.2 Analysis & Evaluation of NoSQL.....	18
3. Practical Skills	20
3.1 Tools & Technologies Used.....	20
3.1.1 Node.js & Express.js.....	20
3.1.2 MySQL & WAMP	22
3.1.3 Visual Studio Code	22
3.2 Weather Application.....	24
3.2.1 Creating a Node.js & Express.js Server	24
3.2.2 Connecting to the MySQL Database	25
3.2.3 Using Controllers	27
3.2.4 CRUD MySQL Database Using REST	28
3.3 Server-Side Templates.....	33
3.3.1 Rendering JSON Array.....	34
3.3.2 Inserting Records Using Forms	37
3.3.3 Editing Data	39
3.3.4 Deleting a Record.....	42
3.3.5 Searching for Records	43
4. Ajax Requests to an API	45
4.1 Simple API Example	45
4.2 Weather API.....	48
4.2.1 Searching Weather API	50
4. Evaluation	54

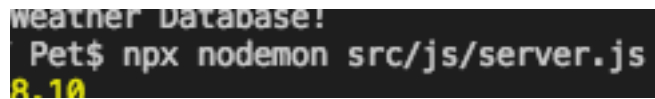
5. References	56
6. Bibliography.....	58
7. Appendix.....	60
7.1 Node.js Application.....	60
7.2 Ajax & JQuery Application	74

Connection Information

The following section outlines the connection information for each application.

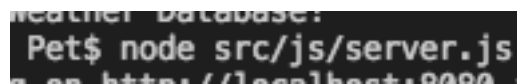
Node.js Application

To run the server on the *Node.js* application, in the terminal, either one of the following commands needs to be executed:



```
Weather Database!  
Pet$ npx nodemon src/js/server.js  
8.10
```

The above was executed inside Visual Studio Code's built in terminal, however, this command can be executed using the Windows CLI Command Prompt or Apple's Terminal. The above command starts the server and watches & restarts after any changes to the code.



```
Weather Database!  
Pet$ node src/js/server.js  
on http://localhost:8080
```

Alternatively, the above command can be executed which will also start the server but after any changes to the code, the server will need to be restarted.

Once the server has been started, the app can be viewed at the following localhost URL: <http://localhost:8080>

Ajax Weather API Application

To start and view the Ajax application, the folder needs to be placed inside the local server's root folder. If using WAMP, this would be the *www* directory, or in the case of MAMP, *htdocs*, found in the Applications directory.

Once the folder is in the root localhost directory, the application can be viewed at the localhost's port address. Usually <http://localhost:8080> for WAMP & <http://localhost:8888> for MAMP.

Database Connection

The database connection file is found in the following directory:

src/js/database.ts

To be able to connect to the database using WAMP, the *'socketPath'* connection variable shown in the screenshot below will need to be commented out. This was only required for working with MAMP.

```
const db = mysql.createConnection({  
  socketPath: '/Applications/MAMP/tmp/mysql/mysql.sock',  
  host: 'localhost',  
  user: 'root',  
  password: 'root',  
  database: 'ewt_weather'  
});
```

1. Introduction

The Emerging Web Technology module has largely focused on JavaScript technologies such as, Typescript, AJAX (Asynchronous JavaScript and XML) [1] and JQuery. However, certain techniques using C# have been explored to integrate with the aforementioned JavaScript technologies, such as building web services using the WCF (Windows Communication Foundation) [2] and REST (Representational State Transfer) [3], furthermore discussions have taken place on big data and NoSQL approaches.

The following assignment will be split into two parts, the first section for knowledge and understanding will be focusing on a Microsoft's non-JavaScript technology, in its experimental stage, called Blazor [4], which is considered to have a big impact on server and client side development in the future years.

However, by focusing on a technology which does not utilise JavaScript, this assignment strays somewhat from the assignment brief which specifies that an example of the discussed technology in section one should be integrated into an application for section two. Given that Blazor does not use JavaScript, this is not possible, therefore, an example of Blazor will be provided in section one and its equivalent feature will be written in Typescript for use in section two.

The second part of section one researches and discusses NoSQL databases and their different approaches, where they are applied in modern enterprise systems and the benefits and drawbacks to them.

In section three a *Node.js* application written with Typescript, using the *Express* and *EJS Templates* framework has been built and its REST features discussed. This also includes documentation on a separate application which integrates AJAX services which make calls to a weather API endpoint. Both the applications are accompanied by documentation explaining, in detail the applications code and how it works.

2. Knowledge & Understanding Skills

The following section will explore the concepts around Microsoft's new experimental technology called Blazor running through a technology called WASM (Web Assembly) [5]. A discussion on its key benefits and uses will precede a small example of Blazor with explanations of the code in use.

2.1 Microsoft .NET Core Blazor

In the world of SPA's (Single Page Applications) a developer's only option was the use of a JavaScript front-end framework such as Angular, Node.js or Vue.js. However, in 2018, Steve Sanderson and Dan Roth of Microsoft introduced a new experimental SPA framework called Blazor, following the MVC pattern [6][7] and built in .NET Core, it integrates Microsoft's Razor components and the browser, hence the name *Blazor* [8].

Blazor allows the use of C# or F# to build SPA's and run any .NET library in any modern browser, including mobile devices [9], removing the need for any third party JavaScript frameworks [10], which allows any developer used to working with .NET and C# to use the same language across the front and back-ends of a project. Additionally, there are no restrictions on server-side technologies, meaning that any server capable of storing static files can manage a Blazor application including those who don't have .NET support. Blazor is considered a true client side framework [8].

Roth and Latham [9] state that the Blazor application framework has the following benefits:

Written in C# – Blazor is written in C#, a commonly used language for web development. Excluding SQL, C# is the 3rd most popular language in use today by developers [11]. As stated earlier, if using C# for backend development, developers will find an easier transition to using it in the front end rather than learning and adding *another* JS framework.

.NET Core – using the .NET core ecosystem means Blazor is available to developers on all different platforms from OSX, Linux and Windows. Furthermore, .NET core allows access to some of the largest available libraries for building web, mobile, desktop and gaming applications [12] and is the third most popular technology in use today [11].

Full-Stack Development – offering the sharing of logic between the client and the server in the same language is the main benefit of Blazor. As C# is a feature-rich language which has had great success in recent years, furthermore, C# code from the server can be reused on the client side [5].

Speed & Scalability – as .NET was built for speed and scalability, it offers a great platform to build applications in Blazor. Furthermore, using C# across the front and back ends will provide some minor performance increases, admittedly negligible.

Leading Development Tools – being a Microsoft product, Blazor is available for development with the industry leading, Visual Studio range of products, such as VS Code and Visual Studio.

Consistency & Stability – being from a trusted source like Microsoft, Blazor is built on a tried and tested, feature rich set of languages, frameworks and tools which are easy to use.

Blazor offers support for the core features a modern SPA would require, such as [6], [9]:

- Routing (like MVC apps).
- Layout (Server-side templates).
- Component models for composable user interfaces.
- Validation server side & client side.
- Dependency injection.
- JavaScript interoperability.
- Live reloading during the browser when developing.
- Server-side rendering.

- Debugging in browsers and the IDE.
- Publishing and app size trimming.
- Intellisense integration.

Blazor is still in its very early stages of experimental development, having only been on experimental release since 2018, the latest version is 0.9.0 and Roth [6] states that it should not be used for any production applications at this time. Furthermore, Microsoft have made no announcements on whether or not they will release Blazor [8], and even if they do there is no way of knowing if developers will adopt it.

It was chosen for investigation for this assignment as it is believed that this technology could be the future of applications for a C# and .NET developer. Blazor offers some promising features and the lure of moving away from overly complex JavaScript frameworks will appeal to many developers, furthermore, the .NET Core capabilities make it one to look out for in the future of web development.

2.1.1 Web Assembly & MONO

To be able run .NET and C# in the browser, Blazor uses two new technologies called WASM [13] and Mono [14]. Firstly, WASM is an open, standardised web technology, supported by all the modern browsers, without plug-ins, which is portably sized in bytecode format to be loading time efficient, making it suitable for compiling code to the web [6], [10].

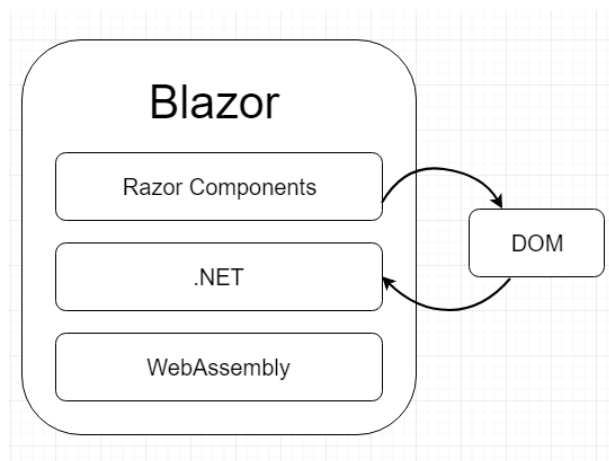


Figure 1 - Blazor

The official WASM [15] website states that:

“WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable target for compilation of high-level languages like C/C++/Rust, enabling deployment on the web for client and server applications.”

This means that WASM takes code written in languages other than JavaScript and is able to transform them into a language which can be read by the JavaScript engine (runtime) which consists of a parser, compiler and JIT (Just in Time) interpreter. This code is then compiled into browser readable code [10], [16], [17], shown in figure two.

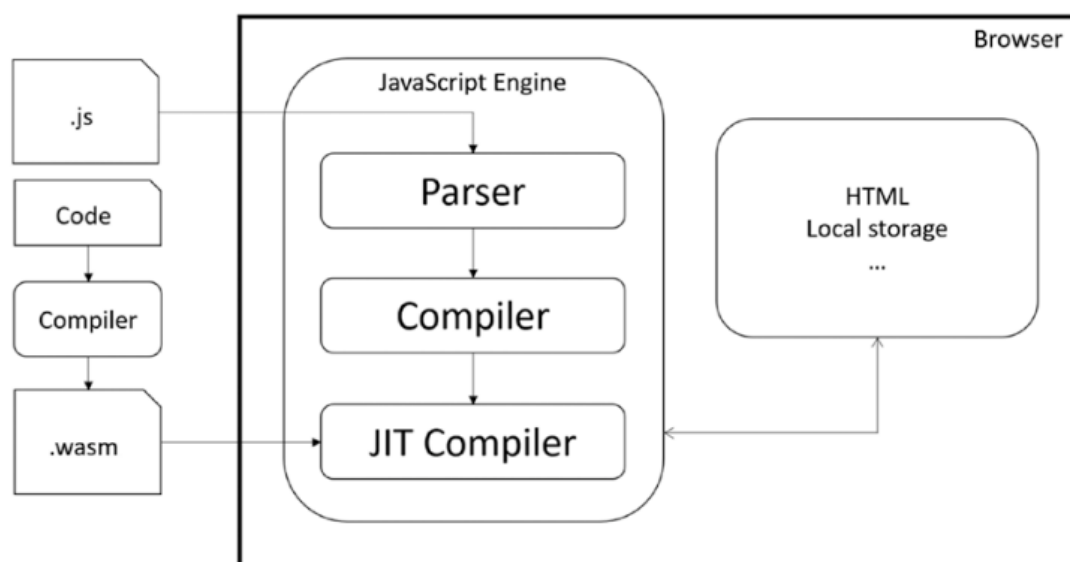


Figure 2 - Web Assembly, JavaScript Engine & the Browser [10]

However, WASM cannot just run Blazor apps by itself. As Blazor uses .NET Core, another technology is needed to evaluate and compile .NET assemblies: Mono. Mono is Microsoft’s own cross-platform and open source .NET runtime meaning that it can run .NET assemblies. It was originally built for running .NET on Linux systems but it can also be used for compiling .NET assemblies to WASM, which can then be compiled to the browser [10], [16].

Blazor uses Mono's JavaScript library to bootstrap the Mono WASM runtime into WASM which then loads the Blazor application's DLL files and the accompanying DLL files of the .NET framework, thus allowing the running of .NET code in the browser [5], [10].

WASM is promising a new future of quicker and more efficient ways of running all different types of high-level languages in the browser whilst still being compatible with existing web technologies in use today. WASM, like Blazor is still in its early stages of development, however, like Blazor, it offers a glimpse into the future technologies and possible trends for web development.

2.1.2 Blazor Example

The following example of Blazor was downloaded from the Microsoft guide to getting started with Blazor on Microsoft's documentation website [18]. However, this example was produced in December 2018 when testing out frameworks to use for the final year project, and therefore it is an older version of Blazor running on .NET core 2.1.2, not 3.0 used in the current documentation's examples.

Furthermore, the new documentation instructs the demo in Visual Studio 2019 preview, however, the examples shown were produced in Visual Studio Code. This is being noted as although the functions of the demo app are the same in the newer example, its implementation was slightly different, therefore the reference may not be the same.

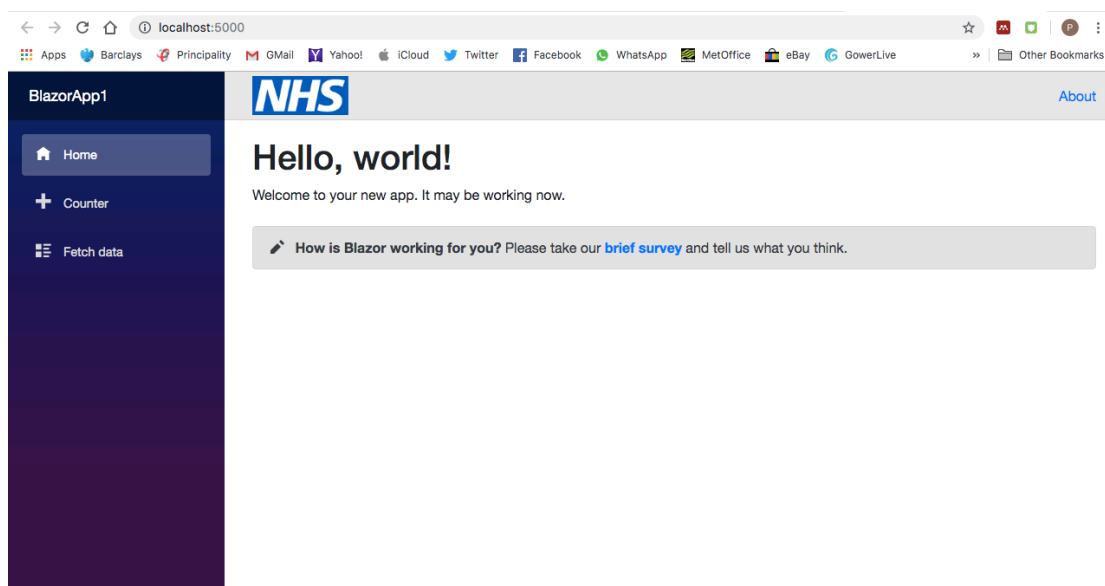
To first install the demo Blazor app, a new VS code project was started, and the template installed using the CLI.

```

1 # install the Blazor Templates
2 dotnet new -i Microsoft.AspNetCore.Blazor.Templates
3 # create a new Blazor App
4 dotnet new blazor -o BlazorApp
5 # run the app, as usual
6 cd BlazorApp
7 dotnet run

```

This creates the application from Microsoft's Blazor templates and starts the inbuilt server. When navigating to the local server's URL, the Blazor app is loaded.



Within the Blazor application, there are two main features, a counter, which simply adds and subtracts numbers to a count, and, a hard-coded weather forecast which fetches data from a.json file to demonstrate the capabilities of Blazor when rendering API's.

i. The Counter

The counter is simply a tiny C# program: it first sets the count to 0, stored in a variable called *currentCount* and then calls ++ or - - to increment the count in two separate functions (the latter additional to the demo example).

```

@page "/counter"

<h1>Counter and Decrease</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" onclick="@IncrementCount">Click me</button>
<button class="btn btn-primary" onclick="@DecrementCount">Decrease</button>

@functions {
    int currentCount = 0;

    void IncrementCount()
    {
        currentCount++;
    }

    void DecrementCount()
    {
        currentCount--;
    }
}

```

What is special about this little program is that the code is being rendered directly on a *.cshtml* (HTML) page, meaning that C# code is being executed directly in the browser.

The first line of code in this app is the routing system, much like a regular ASP.NET MVC application it uses the razor *@page/* syntax. Moving down the functions it can be seen that the functions inside the *@functions* block increase and decrease the count, these functions are called on two buttons, using a familiar JavaScript syntax *onclick* as an inline property in the HTML tag.

Counter and Decrease

Current count: 0



The current count is then printed out, in-between the *<p>* tags using the following syntax:

```
<p>Current count: @currentCount</p>
```

When each of the buttons is clicked, first the *onclick* method is fired, then the *IncrementCount* or *DecrementCount* functions are called, the count is either incremented or decremented, and finally, the component is rendered again by the runtime which compares the new and old content, subsequently applying the changed content to the DOM [18], specifically, without the need for JavaScript.

ii. *Fetching Data*

The Blazor example also demonstrates, although hard coded using a json file, the capabilities of Blazor to make HTTP requests to a server to get data and rendering the data in a HTML table.

Again, the page is routed using the razor *@page"/fetchdata"* syntax, but also added to the page is the razor syntax *@inject* which ties into .NET's dependency injection system where an instance of .NET's *HttpClient* is injected into the page so HTTP requests can be made by the client.

```
1 @page "/fetchdata"
2 @inject HttpClient Http
3
```

The function in this forecast component page contains standard C# code, firstly to asynchronously perform a HTTP GET request to a server for json file, which in this example is simply a hardcoded file within the project.

```
protected override async Task OnInitAsync()
{
    forecasts = await Http.GetJsonAsync<WeatherForecast[]>("sample-data/weather.json");
}
```

When the data is returned, it is deserialized as an array of weather forecast objects which is defined in a nested class within the function.

```

class WeatherForecast
{
    public DateTime Date { get; set; }
    public int TemperatureC { get; set; }
    public int TemperatureF { get; set; }
    public string Summary { get; set; }
}

```

The returned data is then iterated over in the HTML page by using razor syntax and printed out into a table.

```

<tbody>
    @foreach (var forecast in forecasts)
    {
        <tr>
            <td>@forecast.Date.ToShortDateString()</td>
            <td>@forecast.TemperatureC</td>

```

Weather Forecast

This component demonstrates fetching data from the server.

Date	Temp. (C)
05/06/2018	1

2.1.3 Conclusion

The examples shown are simple examples which have been demonstrated using existing Microsoft demo code, given more time to evaluate Blazor properly it would be possible to explore some of its features in more detail and it is being considered for the final year project. This example code will be included in the submission for this assignment.

It could be considered that Blazor, or the technologies which support it, like WASM, are the future of web development, which could see other languages being compiled in the browser, opening up a whole new chapter of web development. For example, languages such as Swift, C++ or Haskell could easily be compiled to the browser [7] using WASM, therefore, Blazor is not just a big glimpse of the future for .NET but could be considered the future for web development in all languages.

2.2 NoSQL Databases

In the early 1990's organisations like Google were finding it increasingly valuable to collect more and more data and process it quickly. As well as finding value in collecting large amounts of data they were also finding it expensive and next to impossible using the traditional RDBMS (relational database management system) paradigm [19].

The solution to the *big data* problem were a new breed of technologies grouped under the term *noSQL*. The *noSQL* technologies first entered the frame in 1998 [20] where Google [21] pioneered big data systems such as: distributed file systems, the MapReduce framework, distributed locking services, and distributed coordination systems [22], [23]. Further innovation by Amazon ensued, who created a distributed key/value system called Dynamo [23]. From these innovations, the open-source market exploded with solutions to big data problems, with solutions coming from Hadoop, MongoDB, Cassandra and Terrastore, to name a few [22], [23].

Vaish [22] states that the term *noSQL* means 'no-and-sql', meaning it is a database without the use of SQL, he adds it is a generic term used to denote any database which does not follow the conventional RDBMS model adding that a data store will come under this term if it non-relational and does not use SQL as a language, even if the data store does not hold much data.

Sadalage and Fowler [19] continue to add that there is no generally accepted definition for what can be classified as 'big data' or a *noSQL* system, but they do state that there are 5 common characteristics which all *noSQL* databases share at least one [24]:

- **Volume** – refers to the quantity of data gathered. Could be terabytes, petabytes or larger.
- **Velocity** – referring to the time in which big data can be processed. This should be as near to real time as possible.

- **Veracity** – referring to the uncertainty of data.
- **Variety** – referring to the type of data that a database can comprise of. Could be structured or unstructured, or even semi-structured (json).
- **Value** – the data's value is derived from the analysis of the data stored.

These types of *noSQL* databases are mainly for solving the scalability problem presented when scaling regular relational databases, although excellent for that purpose, they also offer features such as: schema less data representation, reduced development time through less *join* queries, speed of information retrieval and there is no need to think about future scaling and worrying about relationships between tables as nodes can be added and the systems balance is restored automatically [22], [23].

There are four widely used types of *noSQL* databases, these are: key-value pair databases, document databases, column family stores and graph databases [25]. Each one has its own characteristics and different methods of storing data:

i. Key-Value Stores

Key value store databases such as Riak [24], or Dynamo permit storage of data in an unstructured manner where the data contains a special key, known as an identifier, to the actual data. Key-value stores are considered the simplest to implement and are used where speed of data retrieval is a key factor and when modelling the data is difficult through complexity [25][26].

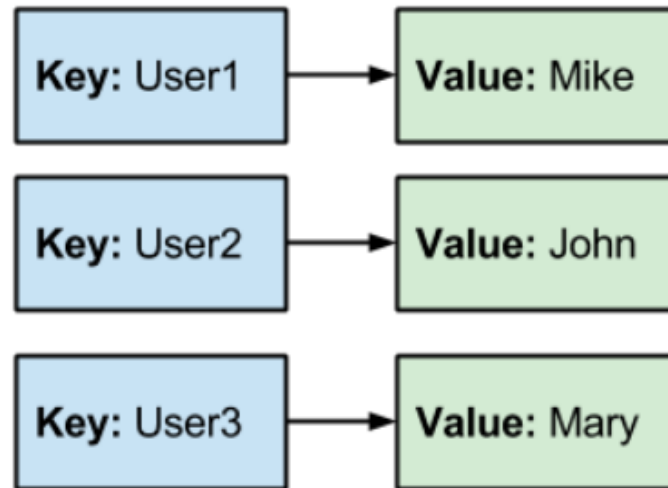


Figure 3 - Key-Value Stores [26]

ii. Column Family Stores

Considered the most complex of *noSQL* database technologies, they share some commonalities with relational databases, in the sense that they have rows and columns. They are designed to handle vast amounts of data spread over many different servers. Like key-value stores, they also use a key system for identifying records although the columns are organised by the column name [25]–[27]. Example of this type of database include Google’s BigTable [21] and HBase.

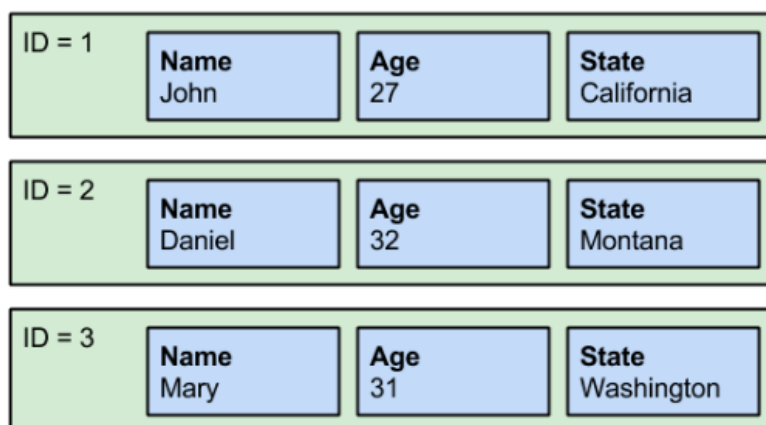


Figure 4 - Column Family Stores [26]

iii. Graph Databases

Represented in the form of nodes, they are the most specialised form a noSQL database. Graph databases do away with columns and rows and instead use a structure of objects representing the identifier and properties being the data attached to that node or object. Nodes can be representative of cities and their relationships or people and their relationships on a social media site. Examples of these databases include Neo4j and InfoGrid [25]–[27].

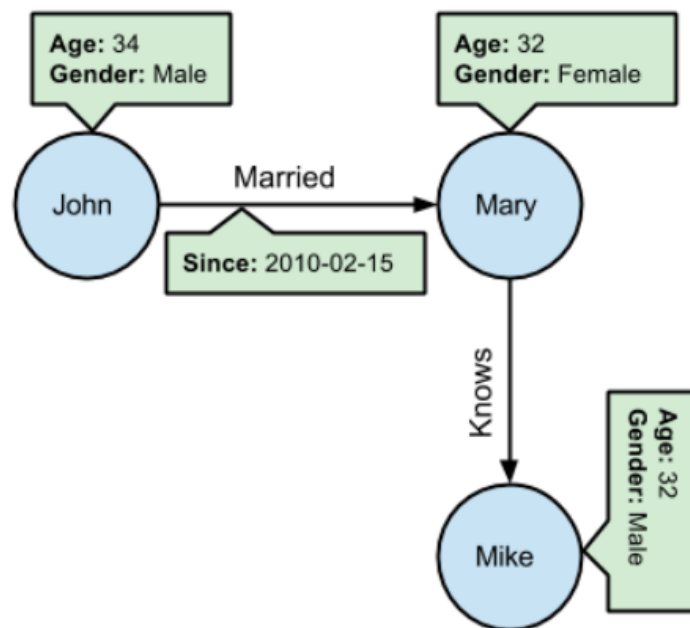


Figure 5 - Graph Databases [26]

iv. Document Databases

Document databases, also known as document orientated databases, such as MongoDB and CouchDB, come in two forms, one which is known as a repository for documents and the other is for storing document components [27]. Document databases are semi-structured in nature and data is usually stored in formats like or similar to json, where each document has multiple attributes or properties [25]. These databases are used when there is a need to store data in a hierarchy or multi-level format without a fixed schema and have been used in popular web applications like Linked-in and Dropbox Mailbox [26].

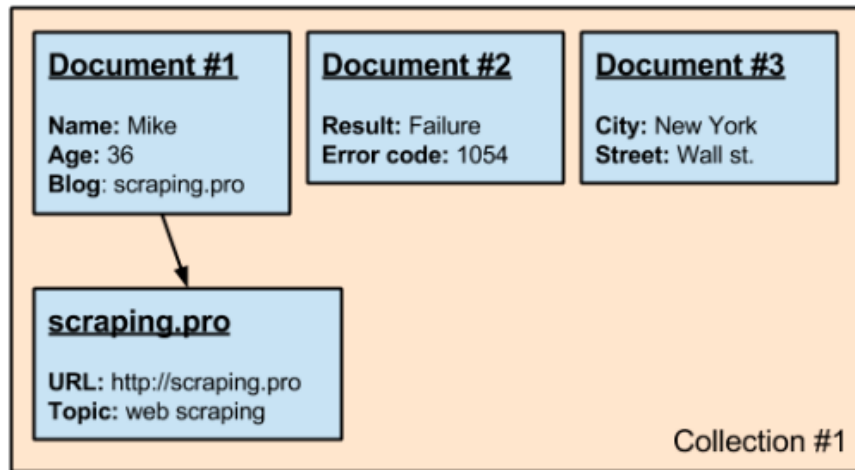


Figure 6 - Document Databases [26]

2.2.1 Application of NoSQL Database Technology

There are many examples of where a *noSQL* database would be applied, for example, on social media sites like Facebook, who use Hive (Hadoop) or multi-dimensional companies like Google who use their own BigTable [21], to financial institutions who would also need to use a *noSQL* approach for concurrency reasons. However, a perfect real-world example of a *noSQL* database is that of an e-commerce site, such as Amazon, who incidentally use their own big data technology.

If Amazon were to use a traditional database, with the sheer size of their product list, any CRUD queries to the database would take an age to complete, not to mention the headache of organising all the relationship between the large number of tables. Furthermore, if *customer A* wanted to buy a book, and complete a transaction on that book, with a regular relational database part of that database would be locked, specifically the stock listing. Obviously this is a problem as every other person trying to access Amazon's products would have to wait until customer A's transaction was completed [22], [25].

To solve this Amazon uses a *noSQL* approach, but more specifically their data stores follow the *BASE* principle invented by Brewer [28], which stands for '*basically*

available, soft state, eventually consistent'. Where relational databases must enforce consistency, *noSQL* databases following the *BASE* principles are always in a state of inconsistency, thus allowing for multiple transactions simultaneously, as required by Amazon.

The *BASE* principles are as follows [22]:

- **Basic Availability** – every request made to a database is guaranteed a response, successful or failed. For Amazon's case, this would be responding to a customer's request for a purchase on one database instantly whilst writing to another with the updated data. Here there is a minimal time of inconsistency, one that importantly, the customer or any other customers will notice.
- **Soft State** – a system may change over time without input for eventual consistency, meaning that older data may be overwritten with new data. Again, for the Amazon example, this would be the data being written to the second database will eventually be overwritten with the final request from the customer on completion of their purchase.
- **Eventually Consistent** – the databases will always be, at some point, inconsistent but will eventually become consistent. For the Amazon example, this means that during the customers transaction using the UI of the website, the data around Amazon's databases is always in an inconsistent state, however, once the transaction is complete the database restores to a consistent state.

For a real-world application like Amazon which is handling 1,000,000's of transactions simultaneously, the databases must nearly always be stored in an inconsistent state, however, Pritchett [28] states that this problem is manageable to cope with and can provide the levels of scalability needed by Amazon which would not otherwise be achievable with a relational DB.

2.2.2 Analysis & Evaluation of NoSQL

As with any technology there are always pros and cons and decisions to make when considering using them. As mentioned in section 2.2.1, a *noSQL* approach offers flexibility and scalability for large scale data applications.

There are, however, as Marz [23] states, some serious trade-offs, firstly, *noSQL* technologies are relatively new meaning knowledge and support is much further behind relational DB', and, in Hadoop, for example, computations have a seriously high latency, meaning that data retrieval is very slow, furthermore, *noSQL* databases such as Cassandra provide great scalability but vastly scale down the data model meaning that fitting an applications data can be difficult and complex, additionally, they are mutable databases, so they are not human fault tolerant [23].

On the contrary, relational databases have been around since the 1970's after being invented by Edgar Codd [29] and have become part of our computing culture [19] offering a standardised model [19]. There is a possibility that every time a human interacts with a computer, they are using a relational database in some form. This means that there is wealth of information about relational databases and creating and using one does not require years of training or experience.

Relational databases also offer value in storing persistent data, which is easily accessed and manipulated through a structured programming language called SQL [30], however, when relational databases are scaled up, with many tables, the queries using SQL joins can get very complex and confusing, furthermore, concurrency is extremely difficult to get right, although relational databases have transactions to help manage concurrency, it still causes difficulties [19].

When choosing an approach for an application it is important to remember that each has their value and that different applications will require different types of databases. Largely, this choice will come down to scale: if a database will be required to scale up then a *noSQL* approach should be favoured as they use a much more limited data

models than relational data models, furthermore there are a wide range of models to choose from, as discussed in section 2.2, with each one offering a greater speed and efficiency over relational DBMS [30]. Additionally, *noSQL* databases do not need an administrator like relational databases, meaning that there can be vast amounts of data stored without the need for someone looking after it, which is just as well as there is no standard query language applied to *noSQL* databases or a standard interface and maintenance is difficult [30].

3. Practical Skills

The following section of the report will document the implementation of a Weather Forecast TypeScript web application which CRUD's a database using a REST web service.

As stated in the introduction, a JavaScript example was not included in section two as the emerging technology investigated Blazor which uses C# for the back and front-end code. An example of consumption of a weather API was given in Blazor, therefore, a weather API will now be used in the following Typescript application. The weather forecast is a dummy forecast hardcoded into a MySQL database to demonstrate the use of REST web service to read and query data.

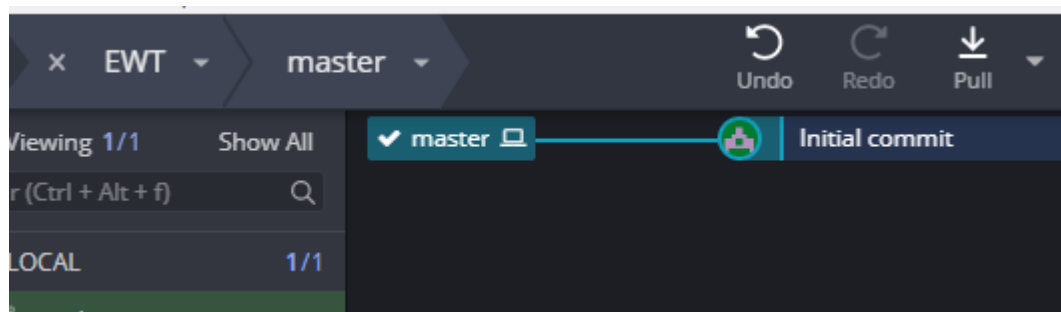
To aid in development of this application a Git repository has been created and a third-party Git tool, GitKraken [31] has been used.

3.1 Tools & Technologies Used

3.1.1 Node.js & Express.js

The application produced will be using *Node.js* [32], a JavaScript runtime which also includes the node package manager for installing project dependencies, and *Express.js* [33], a minimalist web framework for building web servers and applications.

To begin, Node.js is installed and a new directory called 'EWT' was created using the *mkdir* command. This directory was then initialised as a repo using GitKraken.



The project is then initialised using NPM to create a *package.json* configuration file.

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

The express web application framework is then installed using the CLI.

```
C:\Users\Peter\Desktop\gitrepos\EWT>npm install express --save
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN EWT@1.0.0 No repository field.

+ express@4.16.4
added 48 packages from 36 contributors and audited 121 packages in 4.929s
found 0 vulnerabilities
```

Followed by the MYSQL node.js driver which is required to be able to manipulate the MySQL weather database discussed in section 3.1.2.

```
C:\Users\Peter\Desktop\gitrepos\EWT>npm install mysql --save
npm WARN EWT@1.0.0 No repository field.

+ mysql@2.16.0
added 9 packages from 14 contributors and audited 134 packages in 2.287s
found 0 vulnerabilities
```

Finally, the node.js *body-parser* module package is installed which enables the handling of HTTP requests and parses the JSON encoded data.

```
C:\Users\Peter\Desktop\gitrepos\EWT>npm install body-parser --save
npm WARN EWT@1.0.0 No repository field.

+ body-parser@1.18.3
updated 1 package and audited 164 packages in 1.577s
found 0 vulnerabilities
```

3.1.2 MySQL & WAMP

To create the database and have it hosted on a local server, MySQL is installed through the WAMP [32] client for development on Windows and MAMP for OSX. The database itself is made up of one table which is populated with fictitious test data.

id	date	location	temp_low	temp_high	conditions	wind_dir	wind_spd
1	01/04/19	Swansea	2	10	sunny	SSW	8
2	02/04/19	Swansea	8	2	rain showers	W	19
3	03/04/19	Swansea	3	8	sunny intervals	N	17
4	04/04/19	Swansea	4	8	cloudy	SW	14
5	05/04/19	Swansea	6	10	light cloud	E	19

3.1.3 Visual Studio Code

VS code is a free, open-source, cross-platform code editor which has built in Git version control, intellisense (Intelligent Code Completion), syntax highlighting [34] and allows for debugging directly in the editor.

As the application will be using Typescript, it needs to be installed into the project locally. This is done using the VS Code terminal and the following command:

```
PS C:\Users\Peter\Desktop\gitrepos\EWT> npm install --save-dev typescript
npm WARN EWT@1.0.0 No repository field.

+ typescript@3.4.1
added 1 package from 1 contributor and audited 165 packages in 2.573s
found 0 vulnerabilities
```

A new directory is created to contain the source Typescript code and the root directory initialised with the following command to create a *tsconfig.json* file which will be used to compile the Typescript to JavaScript.

```
PS C:\Users\Peter\Desktop\gitrepos\EWT\src> tsc --init
message TS6071: Successfully created a tsconfig.json file.
PS C:\Users\Peter\Desktop\gitrepos\EWT\src> []
```

A new *tasks.json* file is then created using the VS code UI which will watch the *src* directory's Typescript files and build them to JavaScript on save.

```
"version": "2.0.0",
"tasks": [
  {
    "type": "typescript",
    "tsconfig": "tsconfig.json",
    "option": "watch",
    "problemMatcher": [
      "$tsc-watch"
    ],
    "group": {
      "kind": "build",
      "isDefault": true
    }
  }
]
```

3.2 Weather Application

3.2.1 Creating a Node.js & Express.js Server

Now that the project has been set up following the steps outlines section 3.1 the application's development can begin. The first step in creating the RESTful weather service application is to create a Typescript *app.ts* file which will contain the *express* application configuration and a simple end point to return a response to the browser, this file is the entry point to the application, the front controller.

```
1  import express = require('express');
2  const app = express();
3
4  app.set('port', process.env.PORT || 8080);
5
6  app.get('/', (req, res) => {
7
8      res.send('Welcome to the EWT assignment...');
9  });
10
11 export default app;
```

This code simply imports the *express* web app framework and then sets the port which the server should run on. The code then sets a default endpoint which accepts a request and response, in this case, the response is a simple message to be displayed in the browser to ensure the server is up and running. The file is then exported for use in different parts of the application. Now that the first end point has been created, the next step is to create the server. A new *server.ts* file is created to start the server.

```
import app from './app';

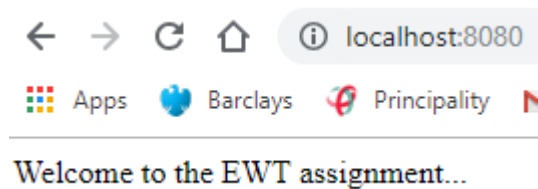
const server = app.listen(app.get('port'), () => {
    console.log('App is running on http://localhost:%d in %s mode',
        app.get('port'),
        app.get('env'))
});

export default server;
```

Here, the *app.ts* file is imported to allow access to the port. The app then sets the server to listen on the specified port and logs a message to the terminal to state the server's current status. The server is then started using the *node* command in the terminal on the *server.js* file.

```
PS C:\Users\Peter\Desktop\gitrepos\EWT> node src/js/server.js
App is running on http://localhost:8080 in development mode
```

When navigating the specified endpoint in the *app.ts* file the response from the routing *app.get* function is displayed.



3.2.2 Connecting to the MySQL Database

The next step is to create a connection to the MySQL database set up in section 3.1.2 and return a JSON array response from the database. Firstly, a *database.ts* file is created with the following code added.

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

// MySQL Connection Config
const db = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: 'p079868',
  database: 'ewt_weather'
});

// Connect To Database
db.connect((err:any) => {
  if (err) {
    console.log('Error connecting to Database!');
    return;
  } else {
    console.log('Connected to Weather Database!');
  }
});

export default db;
```

A MySQL connection is created using the log in credentials of the local MySQL server running on WAMP. The function *connect* is then called on the connection and responses logged to the console. When the *server.js* file is executed again, the following is displayed in the console.

```
App is running on http://localhost:8080 in development mode
Connected to Weather Database!
```

Now that a connection to the database is established, another route function can be added to the *app.ts* file to which contains a query to fetch all entries in the weather database table to test that the database connection is truly working.

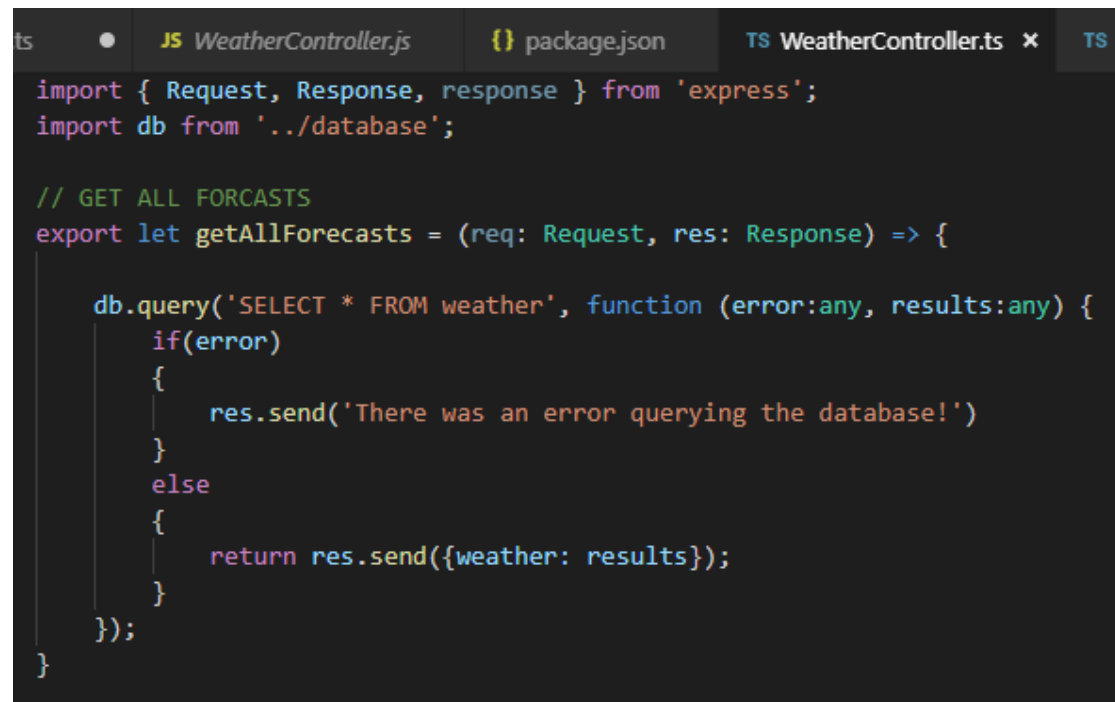
```
// GET All Weather Records
app.get('/swansea', function (req, res) {
  db.query('SELECT * FROM weather', function (error:any, results:any) {
    if(error) {
      res.send('There was an error querying the database!')
    }
    else {
      return res.send({weather: results});
    }
  });
});
```

Now, with the server running and navigating to the specified */swansea* route the records in the database are displayed as a JSON array thanks to the *body-parser* NPM package installed in section 3.1.1.

```
{
  "weather": [
    {
      "id": 1,
      "date": "01/04/19",
      "location": "Swansea",
      "temp_low": 10,
      "temp_high": 15,
      "temp_h": 15,
      "temp_l": 10,
      "wind_dir": "W",
      "wind_spd": 19,
      "weather": "showers"
    },
    {
      "id": 3,
      "date": "03/04/19",
      "location": "Swansea",
      "temp_low": 10,
      "temp_high": 15,
      "temp_h": 15,
      "temp_l": 10,
      "wind_dir": "W",
      "wind_spd": 19,
      "weather": "showers"
    },
    {
      "id": 4,
      "date": "04/04/19",
      "location": "Swansea",
      "temp_low": 4,
      "temp_high": 10,
      "temp_h": 10,
      "temp_l": 4,
      "wind_dir": "W",
      "wind_spd": 19,
      "weather": "showers"
    }
  ]
}
```

3.2.3 Using Controllers

To adhere to REST principles and utilise the MVC pattern, the next step is to create a controller to handle all the weather requests and responses. Firstly, a new *WeatherController.ts* file is created and an endpoint created to get the forecasts from the database.



```
ts    ●    JS WeatherController.js    {} package.json    TS WeatherController.ts    TS
import { Request, Response, response } from 'express';
import db from '../database';

// GET ALL FORECASTS
export let getAllForecasts = (req: Request, res: Response) => {

  db.query('SELECT * FROM weather', function (error:any, results:any) {
    if(error)
    {
      res.send('There was an error querying the database!')
    }
    else
    {
      return res.send({weather: results});
    }
  });
}
```

This controller can then be imported into the *app.ts* file to allow access to the endpoints created in *WeatherController.ts*. The *app.get* requests can then be modified to hit those endpoints, which returns the same JSON array from the database as shown in section 3.2.2.

```
app.get('/swansea', WeatherController.getAllForecasts);
```

```
{ "weather": [{ "id": 1, "date": "01/04/19", "location": "Swansea", "temp": "19", "cloud": "0", "wind_dir": "W", "wind_spd": 19 }, { "id": 3, "date": "03/04/19", "location": "Swansea", "temp": "19", "cloud": "0", "wind_dir": "W", "wind_spd": 19 }, { "id": 4, "date": "04/04/19", "location": "Swansea", "temp": "19", "cloud": "0", "wind_dir": "E", "wind_spd": 19 }, { "id": 6, "date": "06/04/19", "location": "Swansea", "temp": "19", "cloud": "0", "wind_dir": "E", "wind_spd": 19 } ] }
```

3.2.4 CRUD MySQL Database Using REST

The previous sections of the report documented how a server has been created, the database connection implemented with a simple read query for testing and how the application has been set up using a simple controller class. The following section will outline how database records can be created, updated and deleted using a REST web service.

i. *Insert a Record*

To insert a record to the database, firstly the endpoint module is defined as *insertForecast* and within this module, values have been hardcoded into variables for testing purposes. These values are then used in a mySQL query and used in the *db.query* function to insert the record into the database.

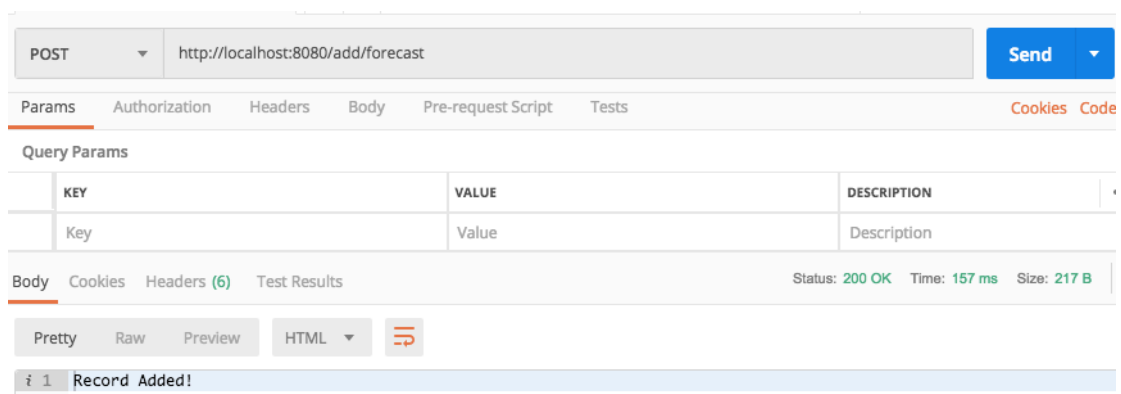
```
export let insertForecast = (req: Request, res: Response) => {  
  
  //let id = 'NULL';  
  let fDate = '08/02/19';  
  let fLoc = 'Pembrokeshire';  
  let fTlow = 1;  
  let fThigh = 11;  
  let fCon = 'Sunny';  
  let fWdir = 'SSW';  
  let fWspd = 3;  
  
  let insertWeather = "INSERT INTO `weather` (`date`, `location`, `temp_low`, `temp_high`, `c`  
  db.query(insertWeather, (error:any, results:any) =>  
  {  
    if(error)  
    {  
      res.status(500).send(error);  
      //res.send('There was an error querying the database!')  
    }  
    else  
    {  
      return res.send('Record Added!');  
    }  
  });  
});
```

In the *app.ts* class the route to the module is specified, however, instead of using the *app.get* request which handles a HTTP GET request the route uses *app.post* which handles an HTTP POST method request as the module is requesting that data be enclosed as the body of the request to insert the data to the database.

```
app.post('/add/forecast', WeatherController.insertForecast);
```

As the data is using the HTTP POST method, it is not possible to simply enter the URI into the browser to test the insert endpoint which was used when using HTTP GET methods, this is because information using POST requests cannot be passed through the URL like the GET methods, firstly for security reasons as it conceals the information being sent to the server and secondly it avoids the query string size limit imposed by a GET request [35].

To test the insert endpoint a third-party application called Postman [36] is required to POST the request. The URI is entered and POST selected, then the request to the server is sent with the following message:



The record has now been inserted into the database. When checking in MySQL, the record can be seen.

			10	08/02/19	Pembrokeshire	1	11	Sunny	SSW	3
--	--	--	----	----------	---------------	---	----	-------	-----	---

And the JSON response has been updated when navigating to the 'get all' URL *'/swansea'*.

```
{
  "id": 8,
  "date": "08/04/19",
  "location": "Lampeter",
  "temp_low": 4,
  "temp_high": 16,
  "condi
  {
    "id": 9,
    "date": "08/04/19",
    "location": "Lampeter",
    "temp_low": 4,
    "temp_high": 16,
    "condi
    {
      "id": 10,
      "date": "08/02/19",
      "location": "Pembrokeshire",
      "temp_low": 1,
      "temp_high": 11,

```

ii. Update Weather Record

Updating a weather record employs much the same code as the insert with just the SQL query being slightly different. This can be seen in the screenshot below.

```
export let updateForecast = (req: Request, res: Response) => {

  let id = 9;
  let fDate = '10/04/19';
  let fLoc = 'Milford Haven';
  let fTlow = 2;
  let fThigh = 11;
  let fCon = 'Light Rain';
  let fWdir = 'SW'
  let fWspd = 4;

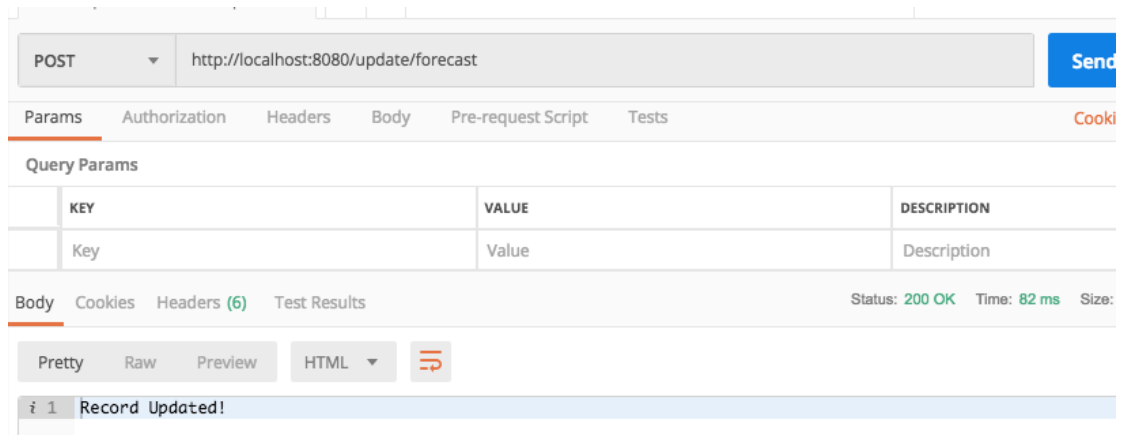
  let updateWeather = "UPDATE `weather` SET `date` = '" + fDate + "', `location` = '" + fLoc + "', `te
  db.query(updateWeather, (error:any, results:any) =>
  {
    if(error)
    {
      res.status(500).send(error);
      //res.send('There was an error querying the database!')
    }
    else
    {
      return res.send('Record Updated!');
    }
  });
};
}
```

The hardcoded variables for testing will be updating the weather forecast with an ID of 9, the record in the database currently has the following information:

☐	Edit	Copy	Delete	9	08/02/19	St. Davids	5	6	Hail	SW	27
--------------------------	------	------	--------	---	----------	------------	---	---	------	----	----

And will be updated to the information shown hardcoded in the screenshot of the node module. A URI route endpoint is set in the *app.ts* class and the endpoint tested in Postman.

```
app.post('/add/forecast', WeatherController.insertForecast);
app.post('/update/forecast', WeatherController.updateForecast);
```



The record has now been updated in the database. Results of the test are shown below. Record with id = 9 has been successfully amended.

id	date	location	temp	humidity	wind	cloud
9	10/04/19	Milford Haven	2	11	Light Rain	SW
10	08/03/19	Bombroschire	1	11	Sunny	SSW

iii. Deleting a Record

Again, deleting a record is much the same as inserting and updating one but with a different module endpoint, a different route, and a different query.

```
export let deleteForecast = (req: Request, res: Response) => {

  let id = 5;

  let deleteWeather = "DELETE FROM `weather` WHERE `id` = '" + id + "'";
  db.query(deleteWeather, (error:any, results:any) =>
  {
    if(error)
    {
      res.status(500).send(error);
      //res.send('There was an error querying the database!')
    }
    else
    {
      return res.send('Record Deleted!');
    }
  });
}
```

```
app.post('/update/forecast', weatherController.updateForecast);
app.post('/delete/forecast', WeatherController.deleteForecast);
```

In the preceding code, weather record with id = 5 will be deleted. Currently this shown below in the database.

4	04/04/19	Cardiff	4	8	cloudy	SW	14
5	05/04/19	Swansea	6	10	light cloud	E	19
6	06/04/19	Cardiff	5	11	Raining	WSW	17

The POST request to delete record 5 is sent via Postman using the specified endpoint.

The screenshot shows the Postman interface for a POST request to `http://localhost:8080/delete/forecast`. The request is successful, returning a status of 200 OK. The response body, displayed in 'Pretty' HTML format, contains the message: `Record Deleted!`.

Record 5 has successfully been deleted.

6	06/04/19	Cardiff	5	11	Raining	WSW	17
7	07/04/19	Newport	2	7	Cloudy	NNW	13
8	08/04/19	Cardiff	1	10	Cloudy	WSW	14

3.3 Server-Side Templates

The preceding examples of performing CRUD operations on a MySQL database demonstrate how REST can be used to query a database, however, at present there are no HTML pages or any formatting to the results, furthermore, all of the variables containing data to insert/update/ into and from the database are hard-coded.

To solve this problem and give the application an interface, the application will utilise the EJS (Embedded JavaScript Templates) [35] template engine which allows for the generation of HTML using JavaScript. It is a template engine very similar to that of .NET's Razor pages or Syfony's Twig Templates, which follow the template pattern [37]. EJS is added using the NPM and the CLI.

```
MacBook:EWT Pet$ npm install ejs
npm WARN EWT@1.0.0 No repository field.

+ ejs@2.6.1
added 1 package from 2 contributors and audited 2496 packages in 10.393s
found 0 vulnerabilities
```

In the *app.ts* the view engine is set to *'ejs'*, and the default route set to render a newly created *index.ejs* page within the *views* directory. The *res.render()* function automatically looks for a directory called *views* for the view files [35].

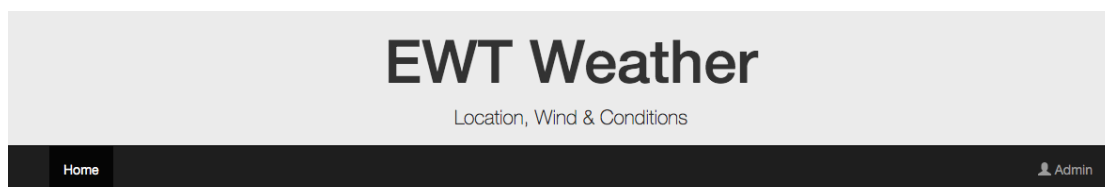
```
// SET TEMPLATE ENGINE
app.set('view engine', 'ejs');
// SET DEFAULT VIEW TO INDEX IN VIEWS FOLDER
app.get('/', (req, res) => {
  res.render('/index');
});
```

Then a simple HTML template was used from W3 schools [35] to save time on coding HTML & CSS on the interface, and split up into separate partial files to make use of the EJS *include* function which works the same as a PHP include where different

section of a page can be split up and reused on different pages to avoid repetition of code.

```
<!doctype html>
<html lang="en">
<% include header.ejs %>

<body>
<div class="page-wrapper container">
  <style>
    .button {
```



3.3.1 Rendering JSON Array

At present the records returned from the MySQL database are returned in a difficult to read JSON array format. To make this information readable, it will now be rendered into a HTML table.

Firstly, a EJS template page was created and another bootstrap template [38] used for the table. A route was then created for this page in the *app.ts* file and linked on the index page from the 'Admin' button.

ID	Date	Location	Temp Low	Temp High	Conditions	Wind Direction	Wind Speed
							<button>Add Record</button>
							<button>Edit</button> <button>Delete</button>

```
app.get('/admin', (req, res) => {
  res.render('./table');
});
```

To get the data into the table on the *table.ejs* page the module returning the JSON data array from the MySQL database will need to be modified. Instead of using the *res.send()* method as before to return the JSON to a page, now the *res.render()* method is used to pass the JSON results to the view and store them in a variable *weatherReport*.

```
export let getAllForecasts = (req: Request, res: Response) => {
  db.query('SELECT * FROM weather', function (error:any, results:any) {
    if(error)
    {
      res.send('There was an error querying the database!')
      // Redirect back home if there is an error.
      res.redirect('/');
    }
    else
    {
      res.render('./table', {
        'weatherReport': results
      });
    }
  });
}
```

Now using the EJS template syntax on the *table.ejs* view, the *weatherReport* variable containing the JSON array is looped through using a *foreach* loop with each property in the array printed out in each table row, again using the special EJS template syntax.

```

<% weatherReport.forEach((weather, index) => { %>
  <tr scope="row">
    <td><%= weather.id %></td>
    <td><%= weather.date %></td>
    <td><%= weather.location %></td>
    <td><%= weather.temp_low %></td>
    <td><%= weather.temp_high %></td>
    <td><%= weather.conditions %></td>
    <td><%= weather.wind_dir %></td>
    <td><%= weather.wind_spd %></td>
    <td>
      <a href="/edit/" target="_blank" rel="noopener" >
      <a href="/delete/" class="btn btn-sm btn-danger">
    </td>
  </tr>
<% }) %>

```

The route to this endpoint can then be updated in the *app.ts* class to allow this endpoint to be accessed at */admin*.

```

app.get('/admin', WeatherController.getAllForecasts);

```

Now the records in the database are displayed in the HTML table, shown below.

								Add Record
ID	Date	Location	Temp Low	Temp High	Conditions	Wind Direction	Wind Speed	
1	01/04/19	Swansea	2	10		SSW	8	Edit Delete
2	02/04/19	Swansea	8	2		W	19	Edit Delete
3	03/04/19	Swansea	3	8		N	17	Edit Delete
4	04/04/19	Swansea	4	8		SW	14	Edit Delete
6	06/04/19	Cardiff	5	11		WSW	17	Edit Delete
7	07/04/19	Newport	2	7		NNW	13	Edit Delete
8	08/04/19	Lampeter	4	16		WSW	1	Edit Delete
9	10/04/19	Milford Haven	2	11		SW	4	Edit Delete
10	08/02/19	Pembrokeshire	1	11		SSW	3	Edit Delete

3.3.2 Inserting Records Using Forms

At present the application only inserts data to the database via a hardcoded *node.js* module. The next steps will be to programme the application to allow users to add records via a web form interface.

The first step is to create the form with the desired fields using an EPS template, then route to the template using a *app.get()* method in the *app.ts* class.

Add a Weather Forecast

The form consists of the following fields and a button:

- Date:** A text input field with placeholder text 'DD/MM/YY'.
- Location:** A text input field with placeholder text 'e.g. 'Swansea''.
- Temp Low:** A text input field with placeholder text 'e.g. 9'.
- Temp High:** A text input field with placeholder text 'e.g. 18'.
- Conditions:** A dropdown menu with a downward arrow icon.
- Wind Direction:** A text input field with placeholder text 'e.g. WSW'.
- Wind Speed:** A text input field with placeholder text 'e.g. 6'.
- Add Forecast:** A green button with white text.

```
app.get('/add', (req, res) => {  
  res.render('./add-forecast');  
});
```

Also in the *app.ts* class, the *body-parser* modules are added to allow the *insertForecast* module in the *WeatherController* to read the values of the form and using the *req.body* property.

```
import bodyParser = require('body-parser');  
app.use(bodyParser.urlencoded({ extended: false }));  
app.use(bodyParser.json()); |
```

Then, the *insertForecast* module is modified to pass in the user inputted data from the form into the variables stated using the *req.body.variable* property.

```

export let insertForecast = (req: Request, res: Response) => {

  let date = req.body.date;
  let location = req.body.location;
  let temp_low = req.body.temp_low;
  let temp_high = req.body.temp_high;
  let conditions = req.body.conditions;
  let wind_dir = req.body.wind_dir;
  let wind_spd = req.body.wind_spd;

  let insertWeather = "INSERT INTO `weather` (`date`, `location`, `temp_low`, `t";
  db.query(insertWeather, (error:any, results:any) =>
  {
    if(error)
    {
      res.status(500).send(error);
      //res.send('There was an error querying the database!')
    }
    else
    {
      res.send('Record Inserted Successfully!');
      res.redirect('/admin');
    }
  });
}

```

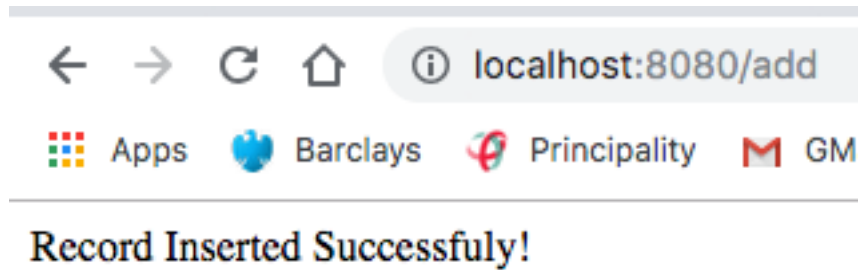
These values are passed into the query and the query is executed using the *db.query* function, as before. A message is then displayed to the user to signal success or failure and then the user is redirected back to the admin page to view the table.

To post the data to the server, the *app.post* method is called in the *app.ts* file on the route */add*. This means that when a GET request is made for */add* the form page *'add-forecast.ejs'* is displayed, but when there is a POST request on this route, the module containing the query is called and the data posted to the database. As the form is set to submit with no action specified it will always submit a POST request back to the same URL.

So, to test the insert module the following data is entered into the form:

Date	Location	Temp Low	Temp High
15/04/19	London	6	15
Conditions	Wind Direction	Wind Speed	
Dark Cloud	ESE	3	

[Add Forecast](#)



12	11/04/19	Mumbles	1	1	NW	2	Edit	Delete
13	15/04/19	London	6	15	ESE	3	Edit	Delete

And the data has been successfully entered to the database using a REST HTTP POST method.

3.3.3 Editing Data

Again, editing data has only been tested through a hardcoded module. To allow user to dynamically update weather records another EJS view template is created with the same form as in section 3.3.2.

Editing data, from the controller's point of view is slightly different. *WeatherController* will require two module functions, one to fetch the record by the id and pass the contents of the returned JSON array to the view, enabling the record to be displayed in the relevant entry field as a placeholder, and the second to perform the update query. The first module:

```

export let getForecastByID = (req: Request, res: Response) => {

  let weatherId = req.params.id;

  if (!weatherId)
  {
    return res.status(400).send('Please provide a weather ID');
  }

  db.query('SELECT * FROM weather WHERE id=?', weatherId, function (error:any, results:any) {
    if(error)
    {
      res.send('There was an error querying the database!')
    }
    else
    {
      console.log(results);
      return res.render('./edit-forecast', {
        // Pass results to view
        'weather': results[0]});
    }
  });
});
}

```

And the second module:

```

export let updateForecast = (req: Request, res: Response) => {

  let id = req.params.id;
  let date = req.body.date;
  let location = req.body.location;
  let temp_low = req.body.temp_low;
  let temp_high = req.body.temp_high;
  let conditions = req.body.conditions;
  let wind_dir = req.body.wind_dir;
  let wind_spd = req.body.wind_spd;

  let updateWeather = "UPDATE `weather` SET `date` = '" + date + "', `location` = '"
  db.query(updateWeather, (error:any, results:any) =>
  {
    if(error)
    {
      res.status(500).send(error);
    }
    else
    {
      res.redirect('./admin');
    }
  });
});
}

```

To route to each of these modules, like the insert statement there needs to be two separate HTTP method calls. One to GET the form and one to POST the form data back to the server to update the record. In the *app.ts* file this is done as shown below.

```
app.get('/edit-forecast/:id', WeatherController.getForecastByID);
app.post('/edit-forecast/:id', WeatherController.updateForecast);
```

As the data is passed to the *'edit-template.ejs'* template view it is possible to 'echo' out, to borrow a phrase from php, in the placeholder position using the EJS special syntax:

```
" placeholder="<%= weather.date %>" required>

value="" placeholder="<%= weather.location %>" required>

w" value="" placeholder="<%= weather.temp_low %>" required>
```

This syntax is also used on the edit button to pass the corresponding id to that row to the GET route.

```
<a href="./edit-forecast/<%= weather.id %>"
```

Now, when any row's edit button is clicked, the correct values are passed into the placeholders of the entry fields to allow users to know what they are editing.

Date	Location	Temp Low	Temp High
01/01/01	Town	1	1
Conditions	Wind Direction	Wind Speed	
Raining	NW	1	

[Edit Forecast](#)

13	01/01/01	Town	1	1	Raining	NW	1
----	----------	------	---	---	---------	----	---

The above screenshot shows record with id = 13 in the database and it echoed out into the web form. When the fields are updated:

Date04/04/19

LocationBedford

Temp Low3

Temp High19

ConditionsWindy

Wind DirectionSSW

Wind Speed34

Edit Forecast

13	04/04/19	Bedford	3	19	Windy	SSW	34
----	----------	---------	---	----	-------	-----	----

1304/04/19Bedford319SSW34

The form successfully updated the record in the database.

3.3.4 Deleting a Record

Deleting a record is probably the simplest of the four operations. It requires the same query as shown in section two, however, rather than hard coding the id, the id will be passed in through the delete button using the EJS template syntax and stored using the *req.param* property in the *deleteForecast* module.

```
export let deleteForecast = (req: Request, res: Response) => {  
  let id = req.params.id;  
  
  let deleteWeather = "DELETE FROM `weather` WHERE `id` = '" + id + "'";  
  db.query(deleteWeather, (error:any, results:any) =>  
  {  
    if(error)  
    {  
      res.status(500).send(error);  
      //res.send('There was an error querying the database!')  
    }  
    else  
    {  
      return res.send('Record Deleted!');  
    }  
  });  
}
```

A route is created to GET the request to the server with the :id passed into the URI. There is no need to use POST for this request as there is no sensitive or large data to be passed through the URI.

```
app.get('/delete/:id', WeatherController.deleteForecast);
```

This is then linked to the delete button on the *table.ejs* template view.

```
<a href="/delete/<%= weather.id %>" class="btn btn-sm btn-danger">Delete</a>
```

Now, when record 13 is to be deleted, by simply clicking on the delete button, the record will be deleted.

13	04/04/19	Bedford	3	19		SSW	34	
10	08/02/19	Pembrokeshire	1	11		SSW	3	Edit Delete
12	11/04/19	Mumbles	1	1		NW	2	Edit Delete

3.3.5 Searching for Records

On the opening page of the application there is a search form which was intended to fetch results based on a search location, unfortunately this was unable to be implemented in the given time frame.

The query written does fetch the results from the database when manually entered into browser, for example, if the URI <http://localhost:8080/search/Swansea> is visited, it returns the correct results, as shown below:

```
{ "weather": [ { "id": 1, "date": "01/04/19", "location": "Swansea", "temp_low": 8, "temp_high": 19 }, { "id": 2, "date": "02/04/19", "location": "Swansea", "temp_low": 8, "temp_high": 19 }, { "id": 3, "date": "03/04/19", "location": "Swansea", "temp_low": 3, "temp_high": 19 }, { "id": 4, "date": "04/04/19", "location": "Swansea", "temp_low": 4, "temp_high": 19 } ] }
```

However, when trying to append the search results from the form using an input and submit button the URL is appended like: <http://localhost:8080/?location=Swansea> which will not hit the end point set in the *app.ts* file:

```
app.get('/search/:location', WeatherController.getForecastsBySearch);
```

Many combinations were tried to try and append the URL properly to allow the user to search the database. It is felt that this is a simple problem with a very simple solution, however, having worked on this application for too long, and especially this problem, it is felt some blindness to the solution has occurred.

```
export let getForecastsBySearch = (req: Request, res: Response) => {  
  
  let location = req.params.location;  
  
  if (!location)  
  {  
    return res.status(400).send('Please provide a weather location');  
  }  
  db.query('SELECT * FROM weather WHERE location LIKE ?', ['%'] + location +  
    if(error)  
    {  
      res.send('There was an error querying the database!')  
    }  
    else  
    {  
      // res.render('./search', {  
      //   'weather': results  
      // });  
      return res.send({'weather': results});  
    }  
  });  
}
```

3.4. Ajax Requests to an API

The EJS server-side templates used in the node application produced and demonstrated in section three do not allow for the use of external JavaScript files to be rendered on the client side. Research into the issue showed that because the EJS templates are running on the server side it cannot render the client-side JavaScript files, however, some users on a well-known Q&A [39] site offered the solution of including the following code in the *app.ts* file:

```
app.use(express.static(path.join(__dirname, 'src')));
```

This should let the express app know to look in the directory 'src' for any related files and the external JS files can then just be linked in the view as normal. In the case of the application produced, all the JavaScript files were located in this directory, however, when trying to link external JavaScript files in the views, the files would still not load.

Having spent many hours trying to fix this problem to no avail it was thought the best course of action would be to produce a new application which used static HTML files instead of server-side templates to demonstrate the use of Ajax requests to an API. This mini application, called 'Ajax EWT' will be included in the zipped code accompanying this report.

3.4.1 Simple API Example

To begin making Ajax requests to an API using jQuery, a new project was started with a simple file structure consisting of one *src* folder, one *AjaxWeatherRequest* class and an *index.html* page. To test an Ajax request to an API a number of tutorials [40] were looked at, as when referring back to class notes and exercises it was discovered that the solutions did not work due to a Typescript typing problem. Between the tutorials and the broken class solutions a simple Ajax request was created to an API from clicking a button.

The typescript class contains the following code:

```
class AjaxWeatherRequest
{
    // attributes

    // constructor
    constructor()
    {
        $.ajaxSetup({ cache: false }); // Turn off browser page cache
    }

    // properties

    // methods
    LoadApi(): void
    {
        $("#weatherApi").click(this.GetWeatherOnClick);
    }

    private GetWeatherOnClick(e: JQuery.Event): boolean
    {
        //window.alert("Hello");
        $(document).ready(function() {
            $.ajax({
                url: "http://rest-service.guides.spring.io/greeting"
            }).then(function(data) {
                $('.greeting-id').append(data.id);
                $('.greeting-content').append(data.content);
            });
        });
        e.preventDefault();
        return false;
    }
}

$(function(){
    let weather: AjaxWeatherRequest = new AjaxWeatherRequest();
    weather.LoadApi();
});
```

The constructor simply turns off the browser cache to stop the browser caching the requests to the API and displaying the same data. The first method gets the id of the button input element and calls the jQuery *onclick* function on it which in turn calls the *GetWeatherOnClick* which contains a jQuery Ajax function which makes a call to the API and appends it to two *<p>* tags with the corresponding classes, shown in the next screen shot. The *preventDefault* called on the event and returning false stops the default browser behaviour occurring (refresh) when making the Ajax call.

```
<html>
  <head>
    <title>Hello jQuery</title>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js"></script>
    <script src="src/AjaxWeatherRequest.js"></script>
  </head>
  <body>
    <form id="form1">
      <input id="weatherApi" type="button" value="Load" />
      <br>
      <br>
      <p class="greeting-id">The ID is </p>
      <p class="greeting-content">The content is </p>
    </form>
  </body>
</html>
```

Now when this HTML page is loaded through a local server (MAMP/WAMP) and the button is clicked, the API call is made, and the results appended to the html.

Load

The ID is

The content is

Load

The ID is 278

The content is Hello, World!

The original JSON array returned from this API:

```
{"id":279,"content":"Hello, World!"}
```

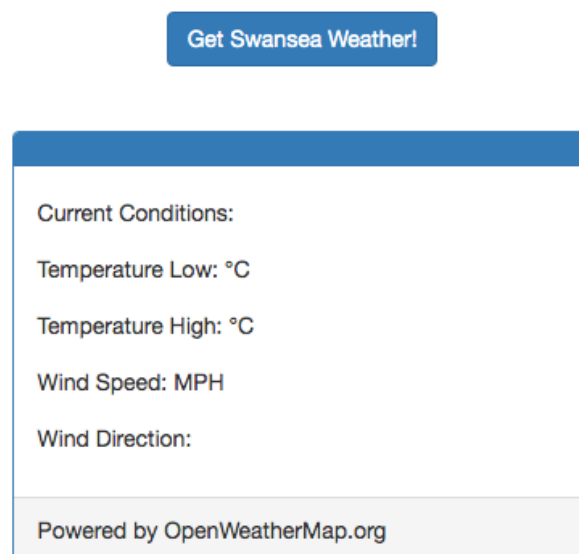
The ID increments each time a call is made; therefore, it differs in each screenshot.

3.4.2 Weather API

To get a weather forecast using an API, OpenWeatherMap.org [41] was used as an account had previously set up for an assignment in year 2. It offers a free API endpoint for any location throughout the world.

To improve the interface for the weather forecast, the HTML was borrowed from the original Node.js application and modified to accept the results returned from the Open Weather endpoint.

```
<div class="panel-body">
  <p>Current Conditions: <b class="description"></b></p>
  <p>Temperature Low: <b class="temp_min"></b>&#176;C</p>
  <p>Temperature High: <b class="temp_max"></b>&#176;C</p>
  <p>Wind Speed: <b class="wind_spd"></b>MPH</p>
  <p>Wind Direction: <b class="wind_dir"></b></p>
</div>
<div class="panel-footer">Powered by OpenWeatherMap.org</div>
```



A new class was created called *OpenWeatherApi.ts* which contained the same logic as in section 4.1, however, the jQuery function has been modified slightly to deal with the larger JSON array object returned from the weather API. Furthermore, some conversions had to be made to display degrees in Celsius as opposed to Kelvin and the wind direction as a compass point rather than degrees.

```
private GetWeatherOnClick(e: JQuery.Event): boolean
{
    //window.alert("Hello");
    $(document).ready(function() {
        $.ajax({
            url: "http://api.openweathermap.org/data/2.5/weather?q=Swansea,uk&appi
        }).then(function(data) {
            // Convert from Kelvin to Celsius
            let tLow: number = data.main.temp_min - 273.15;
            let tHigh: number = data.main.temp_max - 273.15;
            // Round to whole number
            let rTlow = Math.round(tLow);
            let rThigh = Math.round(tHigh);
            // Cast to a string
            let celsiusLow: string = rTlow.toString();
            let celsiusHigh: string = rThigh.toString();
            // Convert Degrees to Wind Direction
            let degCon: number = Math.floor((data.wind.deg / 22.5) + .5);
            let directionValue: string[] = ["N", "NNE", "NE", "ENE", "E", "ESE", "SE",
            let value: any = directionValue[(degCon % 16)];
            // Append results to HTML
            $('.location').append(data.name);
            $('.description').append(data.weather[0].description);
            $('.temp_min').append(celsiusLow);
            $('.temp_max').append(celsiusHigh);
            $('.wind_spd').append(data.wind.speed);
            $('.wind_dir').append(value);
        });
    });
    e.preventDefault();
    return false;
}
```

The jQuery function now makes Ajax requests to a different endpoint from Open Weather and returns the JSON object into an object called *data*. This data object is then manipulated to make the appropriate conversions [41] learnt and then appended to the HTML when the *weatherApi* button is clicked. This results in the following:

Get Swansea Weather!

Swansea

Current Conditions: **light rain**

Temperature Low: **8°C**

Temperature High: **11°C**

Wind Speed: **5.1MPH**

Wind Direction: **SE**

Powered by OpenWeatherMap.org

3.4.3 Searching Weather API

In the previous Ajax request the Open Weather endpoint was hardcoded to only allow the weather for the current day in Swansea. To enable users to search for weather in different locations, first an input field was added to the form with an id of *searchInput* and the jQuery function was updated to get this value from the input box, store it in a variable called *searchLoc* and append it to the URL endpoint.

```
let searchLoc: any = $('#searchInput').val();  
//console.log(searchLoc)
```

```
g/data/2.5/weather?q=" + searchLoc + ",uk&appid=a3db25
```

Now, whatever location users type into the search box, within the UK, it will be appended to the URL and an endpoint relating to weather from that location will be returned and appended to the HTML. For example, if 'London' is entered as a search term, the following results are displayed:

London

Current Conditions: **scattered clouds**

Temperature Low: **11°C**

Temperature High: **14°C**

Wind Speed: **6.7MPH**

Wind Direction: **SE**

Powered by OpenWeatherMap.org

To clear the HTML to allow users to search for further requests, a new button was added to clear the results and linked to a jQuery *onClick* event which called a new function to append empty strings to the HTML.

Get Weather!

Clear Results

```
private ClearWeatherOnClick(e: JQuery.Event): boolean
{
    $(document).ready(function() {
        $('.location').html("");
        $('.description').html("");
        $('.temp_min').html("");
        $('.temp_max').html("");
        $('.wind_spd').html("");
        $('.wind_dir').html("");
    });

    e.preventDefault();
    return false;
}
```

It was then thought that calling a separate function on a separate button was neither an elegant solution, nor did provide good usability and give users a good user experience, therefore, the code to clear the form was moved to the *GetWeatherOnClick* function, before then API call, to clear the results before any new

call is made. This way users can interact with just one button on the interface to find new locations.

```
private GetWeatherOnClick(e: JQuery.Event): boolean
{
    $(document).ready(function() {
        $(document).ready(function() {
            $('.location').html("");
            $('.description').html("");
            $('.temp_min').html("");
            $('.temp_max').html("");
            $('.wind_spd').html("");
            $('.wind_dir').html("");
        });

        let searchLoc: any = $('#searchInput').val();
        //console.log(searchLoc)

        $.ajax({
            url: "http://api.openweathermap.org/data/2
        }, then(function(data) {
```

It was then considered to do error handling and return a message if the server responded with a 404 not found status code. The Open Weather API returns a JSON object as a 404, shown below:

```
{"cod": "404", "message": "city not found"}
```

Therefore, it was considered that this could be used in an *if* block to display a message if the location searched for was not found:

```
if (data.cod == 404)
{
    $('location').append(data.message);
}
else
{
    console.log(url)
    // Convert from Kelvin to Celsius
    let tLow: number = data.main.temp_min - 273.15;
    let tHigh: number = data.main.temp_max - 273.15;
    // Round to whole number
```

However, wrapping the code in an *if* block did not work. Unfortunately, the browser simply served up a 404-error message and would not allow access to the return JSON object and therefore the error handling did not work.

More research into error handling on HTTP status codes was done and it was discovered that the use of *.ajaxError(Handler)* [41] would allow for conditions on the HTTP response status code, however, with little time left before this assignment was due in, the feature was not implemented.

4. Evaluation

The application produced demonstrates a good use of a REST web service to CRUD a database, however, there are many parts of the application which need improving and would have been given more time.

Firstly, in-line styling has been used which is considered bad practice, however, it was used as rendering external JS files in EJS server-side templates proved tricky and could not be implemented. Secondly, the search feature on the home page does not work and provided great frustration, and it is understood that something simple and obvious is being missed!

In addition to the CSS files not being rendered, the external JavaScript files could also not be rendered which means the submission for this assignment now has two separate applications, it would have been nicer to implement the external Open Weather API into the Node.js application. Furthermore, a superior routing system was attempted but it wasn't able to be implemented due to a lack of understanding.

The form handling could also be improved through validation of the fields, and better success/error messaging. Furthermore, the discussed redirects after insert/update/delete's have ceased working for reasons unknown. It appears the forms are resubmitting on redirect causes the app to break, therefore, after an insert/update/delete, users must navigate using the browser's back button or reenter the URL.

It is also felt that although Typescript has been used, a lot of the code could be counted as vanilla JavaScript, which is not what has been asked, however, the Ajax application does use pure Typescript.

Despite all the problems with application it has provided a good insight into a new technology; *Node.js* and its associated technologies like *express* and *ejs* server-side templates as well as the *node package manager*. It is certainly a technology which

given enough practice would provide a simple and effective solution to building server-side & client-side applications using JavaScript & Typescript.

5. References

- [1] "AJAX Introduction," *W3 Schools*. [Online]. Available: https://www.w3schools.com/xml/ajax_intro.asp. [Accessed: 27-Mar-2019].
- [2] Microsoft, "Windows Communication Foundation."
- [3] R. Fielding, "Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST)," Irvine, 2000.
- [4] "Welcome to the Blazor preview docs website!" [Online]. Available: <https://blazor.net/>. [Accessed: 04-Mar-2019].
- [5] "What is Blazor? :: Learn Blazor," *Learn Blazor*, 2018. [Online]. Available: <https://learn-blazor.com/getting-started/what-is-blazor/>. [Accessed: 27-Mar-2019].
- [6] D. Roth, "Blazor Releases," *GitHub*, 2019. [Online]. Available: <https://github.com/aspnet/Blazor/releases>.
- [7] S. Sanderson, "Blazor: a technical introduction," *Steve Sanderson's Blog*, 2018. [Online]. Available: <http://blog.stevensanderson.com/2018/02/06/blazor-intro/>. [Accessed: 28-Mar-2019].
- [8] J. D. Garcia, "An Overview of Blazor - .NET in the Browser," *DNC Magazine*, 2018.
- [9] D. Roth and L. Latham, "Introduction to Blazor | Microsoft Docs," *Microsoft Develop Network*, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/aspnet/core/client-side/spa/blazor/?view=aspnetcore-3.0>. [Accessed: 27-Mar-2019].
- [10] P. Himschoot, *Blazor Revealed*. 2019.
- [11] A. Neagoie, "Want to be a Web Developer? Learn Node.js not PHP – zero to mastery – Medium," *Medium*, 2018. [Online]. Available: <https://medium.com/zerotomastery/want-to-be-a-web-developer-learn-node-js-not-php-dc298154fafd>. [Accessed: 27-Mar-2019].
- [12] "What is .NET?" [Online]. Available: <https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet>. [Accessed: 27-Mar-2019].
- [13] N. Bray, "Web Assembly."
- [14] "Home | Mono." [Online]. Available: <https://www.mono-project.com/>. [Accessed: 27-Mar-2019].
- [15] "WebAssembly." [Online]. Available: <https://webassembly.org/>. [Accessed: 27-Mar-2019].
- [16] D. Glick, "Dave Glick - Blazor, Razor, WebAssembly, and Mono," *Dave Glick*, 2018. [Online]. Available: <https://daveaglick.com/posts/blazor-razor-webassembly-and-mono>. [Accessed: 27-Mar-2019].
- [17] L. Clark, "A cartoon intro to WebAssembly – Mozilla Hacks – the Web developer blog," *Hacks*, 2017. [Online]. Available: <https://hacks.mozilla.org/2017/02/a-cartoon-intro-to-webassembly/>. [Accessed: 27-Mar-2019].
- [18] D. Roth and L. Latham, "Get started with Blazor | Microsoft Docs," *Microsoft Developer Network*, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/aspnet/core/client-side/spa/blazor/get-started?view=aspnetcore-3.0&tabs=visual-studio>. [Accessed: 27-Mar-2019].

- [19] P. J. Sadalage and M. Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison Wesley, 2012.
- [20] R. Irfan Imaduddin, "Nosql Database Technologies," *J. Int. Technol. Inf. Manag.*, 2015.
- [21] F. Chang *et al.*, "Bigtable: A Distributed Storage System for Structured Data."
- [22] G. Vaish, *Getting Started with NoSQL*, 1st ed. Birmingham, Mumbai: Packt Publishing, 2013.
- [23] N. Marz, *Big Data Principles and best practices of scalable real-time data systems*. Shelter Island: Manning, 2015.
- [24] H. Jasim Hadi, A. Hameed Shnain, S. Hadishaheed, and A. Haji Ahmad, "BIG DATA AND FIVE V'S CHARACTERISTICS," 2015.
- [25] D. Sullivan, *NoSQL for Mere Mortals*, 1st ed. Addison-Wesley, 2014.
- [26] M. Shilov, "Where is NoSQL practically used?," *Web Scraping*, 2014. [Online]. Available: <http://scraping.pro/where-nosql-practically-used/>. [Accessed: 28-Mar-2019].
- [27] J. Hurwitz, A. Nugent, F. Halper, and M. Kaufman, *Big Data For Dummies*. O'Reilly, 2013.
- [28] D. Pritchett, "Base: An Acid Alternative," *ACM Queue*, 2008.
- [29] N. Leavitt, "Will NoSQL Databases Live Up to Their Promise?," *Computer (Long. Beach. Calif.)*, vol. 43, no. 2, pp. 12–14, 2010.
- [30] N. Ameya, P. Anil, and P. Dikshay, "Type of NOSQL databases and its comparison with relational databases.," *Int. J. Appl. Inf. Syst.*, vol. 5, no. 4, pp. 16–19, 2013.
- [31] "Git Client - Glo Boards | GitKraken." [Online]. Available: <https://www.gitkraken.com/>. [Accessed: 18-Jan-2019].
- [32] "Node.js," *Node.js*. [Online]. Available: <https://nodejs.org/en/>. [Accessed: 01-Apr-2019].
- [33] "Express - Node.js web application framework." [Online]. Available: <https://expressjs.com/>. [Accessed: 01-Apr-2019].
- [34] Microsoft, "Visual Studio Code - Code Editing. Redefined," *Visual Studio Code*. [Online]. Available: <https://code.visualstudio.com/>. [Accessed: 13-Feb-2019].
- [35] R. T. Fielding and J. Reschke, *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC Editor, 2014.
- [36] "Postman | Download Postman App." [Online]. Available: <https://www.getpostman.com/downloads/>. [Accessed: 03-Apr-2019].
- [37] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.
- [38] "Tryit Editor v3.6 - Bootstrap Table," *W3 Schools*. [Online]. Available: https://www.w3schools.com/bootstrap/tryit.asp?filename=trybs_ref_table-responsive&stacked=h. [Accessed: 25-Mar-2019].
- [39] "Stack Overflow - Where Developers Learn, Share, & Build Careers." [Online]. Available: <https://stackoverflow.com/>. [Accessed: 05-Apr-2019].
- [40] "Getting Started · Consuming a RESTful Web Service with jQuery." [Online]. Available: <https://spring.io/guides/gs/consuming-rest-jquery/>. [Accessed: 02-Apr-2019].
- [41] "Current weather data - OpenWeatherMap." [Online]. Available: <https://openweathermap.org/current>. [Accessed: 05-Apr-2019].

6. Bibliography

- Cattell, R. (n.d.). *Scalable SQL and NoSQL Data Stores*. Retrieved from http://delivery.acm.org/10.1145/1980000/1978919/p12-cattell.pdf?ip=212.219.61.254&id=1978919&acc=ACTIVE_SERVICE&key=BF07A2EE685417C5.D3B0B0D06D8A4908.4D4702B0C3E38B35.4D4702B0C3E38B35&__acm__=1553791774_1d9dc310f4872968b251fdf90bf14466
- Miller, J. C. (2019). Web Development - Full Stack C# with Blazor. Retrieved March 4, 2019, from <https://msdn.microsoft.com/en-us/magazine/mt833288.aspx>
- AJAX Introduction. (n.d.). Retrieved March 27, 2019, from https://www.w3schools.com/xml/ajax_intro.asp
- Top 5 Considerations When Evaluating NoSQL Databases*. (2018). Retrieved from https://webassets.mongodb.com/_com_assets/collateral/10gen_Top_5_NoSQL_Considerations.pdf?_ga=2.61953839.225168861.1552921740-606178624.1552921740&_gac=1.19121676.1552921740.EAlalQobChMIzfiKmvYL4QIVhvdRCh30Rw1-EAAYASABEgLABvD_BwE
- Sharma, A. (2018). Single Page Application Using Server-Side Blazor - Ankit Sharma's Blog. Retrieved March 12, 2019, from <https://ankitsharmablogs.com/single-page-application-using-server-side-blazor/>
- Freeman, A. (2016). *Pro ASP. NET Core MVC. Pro ASP. NET MVC 5 Platform*.
- Neagoie, A. (2018). Want to be a Web Developer? Learn Node.js not PHP – zero to mastery – Medium. Retrieved March 27, 2019, from <https://medium.com/zerotomastery/want-to-be-a-web-developer-learn-node-js-not-php-dc298154fafd>
- Glick, D. (2018). Dave Glick - Blazor, Razor, WebAssembly, and Mono. Retrieved March 27, 2019, from <https://daveaglick.com/posts/blazor-razor-webassembly-and-mono>
- Clark, L. (2017). A cartoon intro to WebAssembly – Mozilla Hacks – the Web developer blog. Retrieved March 27, 2019, from <https://hacks.mozilla.org/2017/02/a-cartoon-intro-to-webassembly/>
- Schmitz, H. (2018). Build a Basic CRUD App with Angular and Node | Okta Developer. Retrieved March 29, 2019, from <https://developer.okta.com/blog/2018/10/30/basic-crud-angular-and-node>

- Creating a RESTful API with Express js, Node js/MySQL - Arjun. (n.d.). Retrieved April 2, 2019, from <https://arjunphp.com/creating-restful-api-express-js-node-js-mysql/>
- Simplest and Easiest way to Build a RESTful API using Node.js - opencodez. (n.d.). Retrieved April 2, 2019, from <https://www.opencodez.com/javascript/build-restful-api-using-node-js.htm>
- Forbes, E. (n.d.). Building a RESTful API using Express and Typescript | TutorialEdge.net. Retrieved April 2, 2019, from <https://tutorialedge.net/typescript/creating-rest-api-express-typescript/#package-json>
- Nemeth, G. (n.d.). Building a Node.js App with TypeScript Tutorial | @RisingStack. Retrieved April 2, 2019, from <https://blog.risingstack.com/building-a-node-js-app-with-typescript-tutorial/>
- Building RESTful Web APIs with Node.js, Express, MongoDB and TypeScript — Part 2. (n.d.). Retrieved April 2, 2019, from <https://itnext.io/building-restful-web-apis-with-node-js-express-mongodb-and-typescript-part-2-98c34e3513a2>
- Getting Started · Consuming a RESTful Web Service with jQuery. (n.d.). Retrieved April 2, 2019, from <https://spring.io/guides/gs/consuming-rest-jquery/>
- Express 4.x - API-Referenz. (n.d.). Retrieved April 2, 2019, from <http://expressjs.com/de/api.html>
- Joshi, B. (2013). Using TypeScript to Perform CRUD Operations. Retrieved April 1, 2019, from <https://www.codeguru.com/csharp/.net/using-typescript-to-perform-crud-operations.htm>
- Barre, G. (2018). Starting a TypeScript project with Visual Studio Code - Meziantou's blog. Retrieved April 1, 2019, from <https://www.meziantou.net/2018/02/01/starting-a-typescript-project-with-visual-studio-code>
- Argon 2016 - MySQL Historical Weather Database System. (n.d.). Retrieved April 1, 2019, from <http://www.weathergraphics.com/argon/>
- Sevilleja, C. (n.d.). Use EJS to Template Your Node Application — Scotch.io. Retrieved April 3, 2019, from <https://scotch.io/tutorials/use-ejs-to-template-your-node-application>

7. Appendix

7.1 Node.js Application

Weather Controller

```
import { Request, Response, response } from 'express';
import db from '../database';
import { read } from 'fs';

// GET ALL FORECASTS
export let getAllForecasts = (req: Request, res: Response) => {

  db.query('SELECT * FROM weather', function (error:any, results:any) {
    if(error)
    {
      res.send('There was an error querying the database!')
      // Redirect back home if there is an error.
      res.redirect('/');
    }
    else
    {
      // Pass results to view
      res.render('./table', {
        'weatherReport': results
      });
    }
  });
}

// GET FORECASTS WITH ID
export let getForecastByID = (req: Request, res: Response) => {

  let weatherId = req.params.id;

  if (!weatherId)
  {
    return res.status(400).send('Please provide a weather ID');
  }

  db.query('SELECT * FROM weather WHERE id=?', weatherId, function
(error:any, results:any) {
    if(error)
    {
      res.send('There was an error querying the database!')
    }
    else
    {
      //console.log(results);
    }
  })
}
```

```

        return res.render('./edit-forecast', {
            // Pass results to view
            'weather': results[0]
        });
    }
});
}

// UPDATE RECORD
export let updateForecast = (req: Request, res: Response) => {

    let id = req.params.id;
    let date = req.body.date;
    let location = req.body.location;
    let temp_low = req.body.temp_low;
    let temp_high = req.body.temp_high;
    let conditions = req.body.conditions;
    let wind_dir = req.body.wind_dir;
    let wind_spd = req.body.wind_spd;

    let updateWeather = "UPDATE `weather` SET `date` = '" + date + "',
`location` = '" + location + "', `temp_low` = '" + temp_low + "',
`temp_high` = '" + temp_high + "', `conditions` = '" + conditions + "',
`wind_dir` = '" + wind_dir + "', `wind_spd` = '" + wind_spd + "' WHERE `id`
= '" + id + "'";
    db.query(updateWeather, (error:any, results:any) =>
    {
        if(error)
        {
            res.status(500).send(error);
        }
        else
        {
            res.send('Record Successfully Updated');
            //res.redirect('./admin');
        }
    });
}

// GET FORECASTS BY LOCATION
export let getForecastsBySearch = (req: Request, res: Response) => {

    let location = req.params.location;

    if (!location)
    {
        return res.status(400).send('Please provide a weather location');
    }
    db.query('SELECT * FROM weather WHERE location LIKE ?', ['%'] + location
+ ['%'], function (error:any, results:any) {
        if(error)

```

```

    {
      res.send('There was an error querying the database!')
    }
    else
    {
      // res.render('./search', {
      //   'weather': results
      // });
      return res.send({'weather': results});
    }
  });
}

// export let getAllLocations = (req: Request, res: Response) => {

//   db.query('SELECT location FROM weather', function (error:any,
// results:any) {
//     if(error)
//     {
//       res.send('There was an error querying the database!')
//       // Redirect back home if there is an error.
//       res.redirect('/');
//     }
//     else
//     {
//       console.log(results);
//       // Pass results to view
//       return res.render('./', {
//         // Pass results to view
//         'locations': results
//       });
//     }
//   });
// }

// }

// INSERT A WEATHER RECORD
export let insertForecast = (req: Request, res: Response) => {

  let date = req.body.date;
  let location = req.body.location;
  let temp_low = req.body.temp_low;
  let temp_high = req.body.temp_high;
  let conditions = req.body.conditions;
  let wind_dir = req.body.wind_dir;
  let wind_spd = req.body.wind_spd;

  let insertWeather = "INSERT INTO `weather` (`date`, `location`,
`temp_low`, `temp_high`, `conditions`, `wind_dir`, `wind_spd`) VALUES ('" +
date + "', '" + location + "', '" + temp_low + "', '" + temp_high + "', '" +
conditions + "', '" + wind_dir + "', '" + wind_spd + "')";
  db.query(insertWeather, (error:any, results:any) => {

```

```

    {
      if(error)
      {
        res.status(500).send(error);
        //res.send('There was an error querying the database!')
      }
      else
      {
        res.send('Record Successfully Inserted!');
        //res.redirect('/admin');
        // REDIRECT IS CRASHING THE APP???? FIND OUT HOW TO REDIRECT
        AFTER DISPLAYING MESSAGE.
      }
    });
  }
}

// DELETE RECORD
export let deleteForecast = (req: Request, res: Response) => {

  let id = req.params.id;

  let deleteWeather = "DELETE FROM `weather` WHERE `id` = '" + id + "'";
  db.query(deleteWeather, (error:any, results:any) =>
  {
    if(error)
    {
      res.status(500).send(error);
      //res.send('There was an error querying the database!')
    }
    else
    {
      return res.send('Record Deleted!');
      //res.redirect('./admin');
    }
  }

  });
}

/* TRIED TO DO ROUTES A DIFFERENT WAY ASK MIKE!! */
// import { JsonController, Param, Body, Get, Post, Put, Delete, Req, Res }
from 'routing-controllers';
// import { Request, Response } from 'express';
// //import express = require('express');
// import db from '../database';
// import { request } from 'http';
// //import app from '../app';

```

```
// @JsonController()
// export class WeatherController {

//     @Get("/swansea")
//     public GetWeather(@Req() request: any, @Res() response: any) : any
//     {
//         db.query('SELECT * FROM weather', function (error:any,
// results:any)
//         {
//             if(error)
//             {
//                 response.send('There was an error querying the
// database!')
//             }
//             else
//             {
//                 return response.send({weather: results});
//             }
//         });
//     }
// }
```

App.ts

```
import express = require('express');
import db from './database';
const app = express();
import * as WeatherController from './controllers/WeatherController';
import bodyParser = require('body-parser');
const path = require('path');
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
app.use(express.static(path.join(__dirname, 'src')));

//SET LOCAL SERVER PORT
app.set('port', process.env.PORT || 8080);
// SET TEMPLATE ENGINE
app.set('view engine', 'ejs');

// SET DEFAULT VIEW TO INDEX IN VIEWS FOLDER
app.get('/', (req, res) => {
    res.render('./index');
});

app.get('/add', (req, res) => {
```

```

    res.render('./add-forecast');
  });

app.get('/admin', WeatherController.getAllForecasts);
//app.get('/', WeatherController.getAllLocations);
app.get('/search/:location', WeatherController.getForecastsBySearch);
app.get('/edit-forecast/:id', WeatherController.getForecastByID);
app.post('/edit-forecast/:id', WeatherController.updateForecast);

app.post('/add', WeatherController.insertForecast);

app.get('/delete/:id', WeatherController.deleteForecast);

export default app;

/* TRIED TO DO ROUTES A DIFFERENT WAY */
// import 'reflect-metadata';
// import {createExpressServer} from 'routing-controllers';
// import {WeatherController} from './controllers/WeatherController';

// const app = createExpressServer({
//   controllers: [WeatherController]
// });

// app.set('port', process.env.PORT || 8080);

// export default app;

```

Database.ts

```

const mysql = require('mysql');

// MySQL Connection Config WAMP
// const db = mysql.createConnection({
//   host: 'localhost',
//   user: 'root',
//   password: 'p079868',
//   database: 'ewt_weather'
// });
// MySQL Conn Info MAMP /** REMOVE socketPath WHEN CONNECTING WITH WAMP **/
const db = mysql.createConnection({
  socketPath: '/Applications/MAMP/tmp/mysql/mysql.sock',
  host: 'localhost',
  user: 'root',

```

```

    password: 'root',
    database: 'ewt_weather'
  });

// Connect To Database
db.connect((err:any) => {

  if (err) {
    console.log('Error connecting to Database!');
    return;
  } else {
    console.log('Connected to Weather Database!');
  }
});

export default db;

```

Server.ts

```

import app from './app';

const server = app.listen(app.get('port'), () => {
  console.log('App is running on http://localhost:%d in %s mode',
    app.get('port'),
    app.get('env'))
});

export default server;

```

Add-forecast.ejs

```

<!DOCTYPE html>
<html lang="en">

  <% include header.ejs %>
  <style>
    button.btn.btn-success.float-right {
      float: right;
    }

    form.add-player-form {
      margin-top: 50px;
    }
  </style>

<body>

```

```

<div class="container">
  <h1>Add a Weather Forecast</h1>

  <form class="add-player-form" action="" method="post">
    <div class="form-row">
      <div class="form-group col-md-3">
        <label for="first-name">Date</label>
        <input type="text" class="form-control" name="date"
id="name" value="" placeholder="DD/MM/YY" required>
      </div>
      <div class="form-group col-md-3">
        <label for="last-name">Location</label>
        <input type="text" class="form-control"
name="location" id="location" value="" placeholder="e.g. 'Swansea'"
required>
      </div>
      <div class="form-group col-md-3">
        <label for="username">Temp Low</label>
        <input type="number" class="form-control"
name="temp_low" id="temp_low" value="" placeholder="e.g. 9" required>
      </div>
      <div class="form-group col-md-3">
        <label for="username">Temp High</label>
        <input type="number" class="form-control"
name="temp_high" id="temp_high" value="" placeholder="e.g. 18" required>
      </div>
    </div>
    <div class="form-row">
      <div class="form-group col-md-4">
        <label for="conditions">Conditions</label>
        <select id="conditions" name="conditions"
class="form-control" required>
          <option selected></option>
          <option>Sunny</option>
          <option>Sunny Intervals</option>
          <option>Raining</option>
          <option>Heavy Rain</option>
          <option>Light Rain</option>
          <option>Rain Showers</option>
          <option>Cloudy</option>
          <option>Light Cloud</option>
          <option>Dark Cloud</option>
          <option>Snowing</option>
          <option>Windy</option>
        </select>
      </div>
      <div class="form-group col-md-4">
        <label for="number">Wind Direction</label>
        <input type="text" class="form-control"
name="wind_dir" id="wind_dir" value="" placeholder="e.g. WSW" required>

```

```

        </div>
        <div class="form-group col-md-4">
            <label for="number">Wind Speed</label>
            <input type="number" class="form-control"
name="wind_spd" id="wind_spd" value="" placeholder="e.g. 6" required>
        </div>
    </div>
    <button type="submit" class="btn btn-success float-
right">Add Forecast</button>
</form>
</div>
</body>

</html>

```

Edit-forecast.ejs

```

<!DOCTYPE html>
<html lang="en">

    <% include header.ejs %>
    <style>
        button.btn.btn-success.float-right {
            float: right;
        }

        form.add-player-form {
            margin-top: 50px;
        }
    </style>

    <body>

        <div class="container">
            <h1>Edit Weather Forecast</h1>

            <form class="add-player-form" action="" method="post">
                <div class="form-row">
                    <div class="form-group col-md-3">
                        <label for="first-name">Date</label>
                        <input type="text" class="form-control" name="date"
id="name" value="" placeholder="<%= weather.date %>" required>
                    </div>
                    <div class="form-group col-md-3">
                        <label for="last-name">Location</label>

```

```

        <input type="text" class="form-control"
name="location" id="location" value="" placeholder="<%= weather.location %>"
required>
    </div>
    <div class="form-group col-md-3">
        <label for="username">Temp Low</label>
        <input type="number" class="form-control"
name="temp_low" id="temp_low" value="" placeholder="<%= weather.temp_low %>"
required>
    </div>
    <div class="form-group col-md-3">
        <label for="username">Temp High</label>
        <input type="number" class="form-control"
name="temp_high" id="temp_high" value="" placeholder="<%= weather.temp_high
%>" required>
    </div>
</div>
<div class="form-row">
    <div class="form-group col-md-4">
        <label for="conditions">Conditions</label>
        <select id="conditions" name="conditions"
class="form-control" required>
            <option value="" selected><%=
weather.conditions %></option>
            <option>Sunny</option>
            <option>Sunny Intervals</option>
            <option>Raining</option>
            <option>Heavy Rain</option>
            <option>Light Rain</option>
            <option>Rain Showers</option>
            <option>Cloudy</option>
            <option>Light Cloud</option>
            <option>Dark Cloud</option>
            <option>Snowing</option>
            <option>Windy</option>
        </select>
    </div>
    <div class="form-group col-md-4">
        <label for="number">Wind Direction</label>
        <input type="text" class="form-control"
name="wind_dir" id="wind_dir" value="" placeholder="<%= weather.wind_dir %>"
required>
    </div>
    <div class="form-group col-md-4">
        <label for="number">Wind Speed</label>
        <input type="number" class="form-control"
name="wind_spd" id="wind_spd" value="" placeholder="<%= weather.wind_spd %>"
required>
    </div>
</div>

```

```

        <button type="submit" class="btn btn-success float-
right">Edit Forecast</button>
    </form>
    <p><%= weather.location %></p>
</div>

</body>

</html>

```

Header.ejs

```

<head>
  <title>EWT Weather</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.0/css/bootstrap.min.css"
>
  <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></scr
ipt>
  <script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.0/js/bootstrap.min.js"></
script>
  <style>
    /* Remove the navbar's default rounded borders and increase the
bottom margin */
    .navbar {
      margin-bottom: 50px;
      border-radius: 0;
    }

    /* Remove the jumbotron's default bottom margin */
    .jumbotron {
      margin-bottom: 0;
      padding: 0;
    }

    /* Add a gray background color and some padding to the footer */
    footer {
      background-color: #f2f2f2;
      padding: 25px;
    }
  </style>

  <div class="jumbotron">

```

```

        <div class="container text-center">
            <h1>EWT Weather</h1>
            <p>Location, Wind & Conditions</p>
        </div>
    </div>

    <nav class="navbar navbar-inverse">
        <div class="container-fluid">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-
toggle="collapse" data-target="#myNavbar">
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
            </div>
            <div class="container">
                <div class="collapse navbar-collapse" id="myNavbar">
                    <ul class="nav navbar-nav">
                        <li class="active"><a href="#">Home</a></li>
                    </ul>
                    <ul class="nav navbar-nav navbar-right">
                        <li><a href="/admin"><span class="glyphicon
glyphicon-user"></span> Admin</a></li>
                    </ul>
                </div>
            </div>
        </div>
    </nav>

</head>

```

Index.ejs

```

<!DOCTYPE html>
<html lang="en">

    <% include header.ejs %>

    <style>
        form.form-inline {
            margin-bottom: 40px;
        }
    </style>

<body>

    <section class="container-fluid text-center">

```

```

        <form method="" action="search/" class="form-inline">
            <input id="searchWeather" name="location" id="location"
type="text" class="form-control" size="50" placeholder="Enter a place...
e.g. Swansea">
            <input type="submit" class="btn btn-sm btn-success"
value="Search!"/>
        </form>
</section>

<!-- <script type="text/javascript" src="/js/AjaxRequest.js"></script> -->

<!-- <script>
"use strict";
var AjaxRequest = /** @class */ (function () {
    // attributes
    // constructor
    function AjaxRequest() {
        $.ajaxSetup({ cache: false }); // Turn off browser page cache
    }
    // properties
    // methods
    // PageLoad(): void
    // {
    //     $("#searchWeather").click(this.ButtonClick);
    // }
    // ButtonClick(): void
    // {
    //     $("#searchWeather").click(this.ButtonClick);
    //     window.alert('HELP!!');
    // }
    AjaxRequest.prototype.ButtonClick = function (e) {
        alert('HELP!');
    };
    return AjaxRequest;
})();
$(function () {
    var page = new AjaxRequest();
    // page.PageLoad();
    // //window.alert("PageLoad");
    // page.ButtonClick();
});

</script> -->
</body>
</html>

```

Table.ejs

```
<!doctype html>
<html lang="en">
<% include header.ejs %>

<body>
<div class="page-wrapper container">
  <style>
    .button {
      float: right;
      margin-bottom: 10px;
    }
  </style>
  <div class="table-wrapper">
    <a href="/add" class="btn btn-sm btn-primary button">Add
Record</a>
    <table class="table table-hovered">
      <thead class="thead-dark">
        <tr>
          <th scope="col">ID</th>
          <th scope="col">Date</th>
          <th scope="col">Location</th>
          <th scope="col">Temp Low</th>
          <th scope="col">Temp High</th>
          <th scope="col">Conditons</th>
          <th scope="col">Wind Direction</th>
          <th scope="col">Wind Speed</th>
        </tr>
      </thead>
      <tbody>
        <% weatherReport.forEach((weather, index) => { %>
          <tr scope="row">
            <td><%= weather.id %></td>
            <td><%= weather.date %></td>
            <td><%= weather.location %></td>
            <td><%= weather.temp_low %></td>
            <td><%= weather.temp_high %></td>
            <td><%= weather.conditions %></td>
            <td><%= weather.wind_dir %></td>
            <td><%= weather.wind_spd %></td>
            <td>
              <a href="/edit-forecast/<%= weather.id
%>" class="btn btn-sm btn-success">Edit</a>
              <a href="/delete/<%= weather.id %>"
class="btn btn-sm btn-danger">Delete</a>
            </td>
          </tr>
        <% }) %>
      </tbody>
    </table>
  </div>
</div>
</body>
</html>
```

```

    </div>
</div>

</body>
</html>

```

7.2 Ajax & JQuery Application

Index.html

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <title>EWT Weather</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.0/css/bootstrap.min.css"
>
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></scr
ipt>
    <script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.0/js/bootstrap.min.js"></
script>
    <script src="src/OpenWeatherApi.js"></script>
    <style>
      .navbar {
        margin-bottom: 50px;
        border-radius: 0;
      }
      .jumbotron {
        margin-bottom: 0;
        padding: 0;
      }
      footer {
        background-color: #f2f2f2;
        padding: 25px;
      }
      form.form-inline {
        margin-bottom: 40px;
      }
      .col-sm-4 {
        position: relative;
        left: 33.3%;
      }
      p.location {
        font-size: 30px;
        font-weight: bold;

```

```

        margin-bottom: 0;
    }
    b {
        font-size: 18px;
    }
</style>

<div class="jumbotron">
    <div class="container text-center">
        <h1>EWT Weather Open Weather API</h1>
        <p>Location, Wind & Conditions</p>
    </div>
</div>

<nav class="navbar navbar-inverse">
    <div class="container-fluid">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle" data-
toggle="collapse" data-target="#myNavbar">
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
        </div>
        <div class="container">
            <div class="collapse navbar-collapse" id="myNavbar">
                <ul class="nav navbar-nav">
                    <li class="active"><a href="#">Home</a></li>
                </ul>
                <!-- ADMIN -->
            </div>
        </div>
    </div>
</nav>

</head>
<style>

</style>

<body>
<section class="container-fluid text-center">
    <form method="POST" class="form-inline">
        <input type="text" class="form-control" id="searchInput"
size="50" placeholder="Enter a place... e.g. Swansea">
        <input type="button" value="Get Weather!" id="weatherApi"
class="btn btn-primary"></input>
        <!-- <input type="button" value="Clear Results" id="clearApi"
class="btn btn-success"></input> -->
    </form>
</section>

```

```

<div class="container">
  <div class="row">
    <div class="col-sm-4 box-center">
      <div class="panel panel-primary">
        <div class="panel-heading">
          <p class="location"></p>
        </div>
        <div class="panel-body">
          <p>Current Conditions: <b class="description"></b></p>
          <p>Temperature Low: <b class="temp_min"></b>&#176;C</p>
          <p>Temperature High: <b class="temp_max"></b>&#176;C</p>
          <p>Wind Speed: <b class="wind_spd"></b>MPH</p>
          <p>Wind Direction: <b class="wind_dir"></b></p>
        </div>
        <div class="panel-footer">Powered by OpenWeatherMap.org</div>
      </div>
    </div>
  </div><br><br>

</body>
</html>

```

OpenWeatherApi.ts

```

class OpenWeatherApi
{
  // attributes

  // constructor
  constructor()
  {
    $.ajaxSetup({ cache: false }); // Turn off browser page cache
  }

  // properties

  // methods
  LoadApi(): void
  {
    $("#weatherApi").click(this.GetWeatherOnClick);
    //$("#clearApi").click(this.ClearWeatherOnClick);
  }

  private GetWeatherOnClick(e: JQuery.Event): boolean
  {
    $(document).ready(function() {

```

```

$(document).ready(function() {
    $('.location').html("");
    $('.description').html("");
    $('.temp_min').html("");
    $('.temp_max').html("");
    $('.wind_spd').html("");
    $('.wind_dir').html("");
});

let searchLoc: any = $('#searchInput').val();
//console.log(searchLoc)

$.ajax({
    url: "http://api.openweathermap.org/data/2.5/weather?q=" +
searchLoc + ",uk&appid=a3db25317180374db1158527d5337ea7"
}).then(function(data) {

    let url: string =
"http://api.openweathermap.org/data/2.5/weather?q=" + searchLoc +
",uk&appid=a3db25317180374db1158527d5337ea7"
    console.log(data.cod)
    if (data.cod == 404)
    {
        $('.location').append(data.message);
    }
    else
    {
        console.log(url)
        // Convert from Kelvin to Celsius
        let tLow: number = data.main.temp_min - 273.15;
        let tHigh: number = data.main.temp_max - 273.15;
        // Round to whole number
        let rTlow = Math.round(tLow);
        let rThigh = Math.round(tHigh);
        // Cast to a string
        let celsiusLow: string = rTlow.toString();
        let celsiusHigh: string = rThigh.toString();
        // Convert Degrees to Wind Direction
        let degCon: number = Math.floor((data.wind.deg / 22.5) +
.5);

        let directionValue: string[] =
["N", "NNE", "NE", "ENE", "E", "ESE", "SE",
"SSE", "S", "SSW", "SW", "WSW", "W", "WNW", "NW", "NNW"];
        let value: any = directionValue[(degCon % 16)];
        // Append results to HTML
        $('.location').append(data.name);
        $('.description').append(data.weather[0].description);
        $('.temp_min').append(celsiusLow);
        $('.temp_max').append(celsiusHigh);
        $('.wind_spd').append(data.wind.speed);
    }
});

```

```

        $('.wind_dir').append(value);
    }

    });
    e.preventDefault();
    return false;
}

// private ClearWeatherOnClick(e: JQuery.Event): boolean
// {
//     $(document).ready(function() {
//         $('.location').html("");
//         $('.description').html("");
//         $('.temp_min').html("");
//         $('.temp_max').html("");
//         $('.wind_spd').html("");
//         $('.wind_dir').html("");
//     });

//     e.preventDefault();
//     return false;
// }

}

$(function(){
    let weather: OpenWeatherApi = new OpenWeatherApi();
    weather.LoadApi();
});

```

