CODING STANDARDS

INTRODUCTION

This document outlines a series of guidelines for how our project will be coded. Each developer will try to adhere to these guidelines in order to make the code more readable and consistent.

EDITOR SETTINGS

- Code editors should be configured to make use of 4-space indentation.
- Linters should be used to maintain consistent coding standards

GENERAL STANDARDS

- JavaScript/TypeScript strings should be coded to use string interpolation. This is done using the quote character ` instead of " or '. Ideally string interpolation should be used at all times.
- Use of async/await, promises and observables where useful.
- Lower-camel for all variable and function names. Example: myFunc, myVariable, myReallyLongFunc.
- Underscored lower-camel for all private variables. Example: _myPrivateVariable.
- Upper-camel for class names. Example: ExampleClass.

Commenting

Code should be commented as shown below. Sometimes, commenting is not needed as simplistic functions are inherently clear what their purpose is. Where appropriate, code shall be commented clearly using the format shown below:

```
/**
 * This is an example of a comment
 * @param parameters used within the function
 * @author sometimes the author of a specific function will be listed, this is rare though
 * @return the expected returned values will be listed in certain circumstances where this is
 * clear by reading the code
 **/
public function() {
      Do something;
}
```

ASYNC/AWAIT, PROMISES, OBSERVABLES

Observables are basically a publish subscribe system. You create an observable object and other parts of your application subscribe to it, then when the data of the observable changes all parts of the program that are subscribed get the new data. There are some good tutorials on this relating to Angular 4/5 which meant that

Async/await and promises are all related to asynchronous code execution. I mainly use promises but from my understanding it goes like this:

- Promises have a resolve and reject state. When you've finished executing your code and have something to return you call resolve() and when you want to error you call reject(). What ever called your promise captures these in a .then() and .catch() call.
- Async/await is, from what I understand, an easier way to do async code than promises. You basically have a function you declare async then when you call it you use await and the call will wait for the async task to finish, but I don't think it hangs the main thread.

Example promise:

```
public callPromise() {
```

```
    myPromise(true)
        .then((result) => {
            console.log(`Promise returned value of ${result}`);
        })
        .catch((error) => {
            console.log(error);
        });
}

public myPromise(fail:boolean): Promise<boolean> {
    return new Promise<boolean>(resolve, reject) => {
        if(fail) {
            reject(`Some error.`);
        } else {
            resolve(true);
        }
    }
}
```

USING VARIABLE AND FUNCTION MODIFIERS

Ideally all variable and function declarations should have public/private/protected/abstract/async modifiers:

```
public count:number = 0;
public static sum:number = 0;

public myFunc():number {
    return 0;
}

public async myFunc():number {
    return 0;
}
```

Functions that don't return anything don't require a return type.

STRONG TYPES

Ideally all variable declarations should be strongly typed, and all functions should have strongly typed return types indicated with a colon followed by the type. For example:

```
public abc:number = 0;
public def:string = ``;
public ghi:boolean[] = [];
```

USE OF GET/SET OVER TRUE FUNCTIONS

Ideally functions should be used when executing a lot of code or performing some kind of calculation. These can return an object or not, doesn't matter. But if you're just doing basic calculation or returning basic values it would be better to use get/set. Example:

```
public get length():number {
    return this._length;
}
```

```
public get lengthDoubled():number {
    return this._length * 2;
}

public set length(value:number) {
    this._length = value;
}
```

Git UNIX/Windows OS Formatting and Whitespace
Included as a clear explanation of the issue. We are working across both Unix and Windows systems, hence the need for consistent standards around formatting and whitespace.

Formatting and whitespace issues are some of the more frustrating and subtle problems that many developers encounter when collaborating, especially cross-platform. It's very easy for patches or other collaborated work to introduce subtle whitespace changes because editors silently introduce them, and if your files ever touch a Windows system, their line endings might be replaced. Git has a few configuration options to help with these issues.

`Core.autocrlf`

If you're programming on Windows and working with people who are not (or vice-versa), you'll probably run into line-ending issues at some point. This is because Windows uses both a carriage-return character and a linefeed character for newlines in its files, whereas Mac and Linux systems use only the linefeed character. This is a subtle but incredibly annoying fact of cross-platform work; many editors on Windows silently replace existing LF-style line endings with CRLF, or insert both line-ending characters when the user hits the enter key.

Git can handle this by auto-converting CRLF line endings into LF when you add a file to the index, and vice versa when it checks out code onto your filesystem. You can turn on this functionality with the core.autocrlf setting. If you're on a Windows machine, set it to true – this converts LF endings into CRLF when you check out code:

```
$ git config --global core.autocrlf true
```

If you're on a Linux or Mac system that uses LF line endings, then you don't want Git to automatically convert them when you check out files; however, if a file with CRLF endings accidentally gets introduced, then you may want Git to fix it. You can tell Git to convert CRLF to LF on commit but not the other way around by setting core.autocrlf to input:

```
$ git config --global core.autocrlf input
```

This setup should leave you with CRLF endings in Windows checkouts, but LF endings on Mac and Linux systems and in the repository.

If you're a Windows programmer doing a Windows-only project, then you can turn off this functionality, recording the carriage returns in the repository by setting the config value to false:

```
$ git config --global core.autocrlf false
```

MongoDB
Development
The MongoDB server is hosted on a DigitalOcean Droplet. Database connection happens automatically when the file is installed according to instructions in the readme.md file.

Production
Production shall make use of the free sandbox environment online with mLab to host the MongoDB database. No Redundancy/Backup with this option so consideration needs to be taken on how we will populate the database.

Unfortunately, specific ports are needed to access mLab from within the University campus - thus rendering it’s usage impossible for the development stages of this process.