# WEB422 Assignment 6

## Submission Deadline:

Friday, April 5th 2019 @ 11:59pm

## Assessment Weight:

9% of your final course Grade

## Objective:

For this final assignment, students will work with an existing Angular application (ie: Assignment 5) and update it to allow users to edit data as well as to practice building and deploying their App to the cloud (Heroku).

**Note**: **This assignment relies on the successful completion of Assignment 5. If you require a fresh copy of Assignment 5, please email your Professor.**

## Specification:

For our updated Assignment 5 application (now Assignment 6) we will use our knowledge of Template Driven Angular Forms to allow users to edit employee / position information as well as add some search/filtering to our Employees table to make it more useable. Lastly, we will deploy it on Heroku.

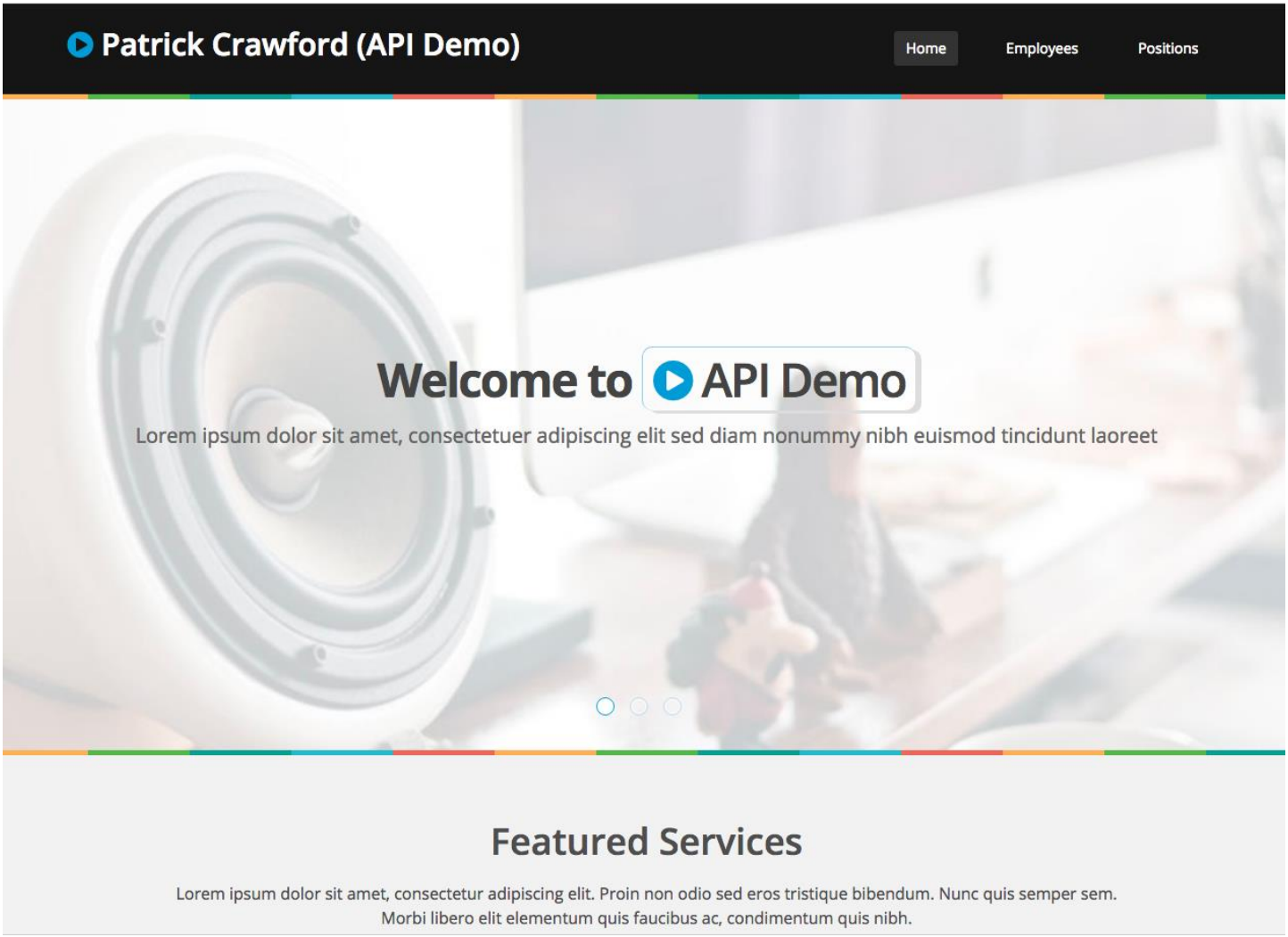
### Getting Started:

To begin this assignment, we will first need to make a copy of or current Assignment 5 and open it in Visual Studio Code. Once it is open, we can proceed to add new methods to our Employee & Position services (Part 1)

**NOTE**: For Assignment 5, we didn't address the issue of the background slider images, ie: the console errors that look like:

**images/slider/bg3.jpg Failed to load resource: the server responded with a status of 404 (Not Found)**

To fix these errors, simply open your home.component.html file and modify the 3 image paths (bg1.jpg, bg2.jpg & bg3.jpg) from: **images/slider/…** to **/assets/images/slider/…** This should give you your slider backgrounds back:




# Part 1 - New Functionality for Editing Employees / Positions

## Step 1 - Updating EmployeeService & PositionService

For this assignment, we will enable users to edit existing employees & services. To enable this functionality, we must add some additional properites & methods to our EmployeeService & PositionService:

### EmployeeService (employee.service.ts)

Before we can begin to update our EmployeeService, we must define a new class: **EmployeeRaw** in its own file (employeeRaw.ts) within the "data" folder. The EmployeeRaw class must have the following properties / types:

| **Property** | **Type** |
|---|---|
| _id | string |
| FirstName | string |
| LastName | string |
| AddressStreet | string |
| AddressState | string |

| | |
|---|---|
| AddressCity | string |
| AddressZip | string |
| PhoneNum | string |
| Extension | number |
| Position | string |
| HireDate | string |
| SalaryBonus | number |
| __v | number |

Next, we must import our new EmployeeRaw class in our EmployeeService. This will allow us to create the following additional 2 methods:

- **saveEmployee(employee: EmployeeRaw)**

  This method must make a "**put**" request (using the HTTPClient module) to your Teams API running on heroku with the path: "/employee/id" (where "id" matches the employee's "_id" property). It will return an Observable of type any, ie: Observable<any>. **NOTE**: with a "put" request, you can send the data in the second parameter, ie:

  - return this.http.put<any>("someURL/", **someData**);

- **getEmployee(id)**

  This method must make a "**get**" request (using the HTTPClient module) to your Teams API running on heroku with the path: "/employee-raw/id" (where "id" matches the "id" parameter to the function). It will return an Observable of type EmployeeRaw[], ie: Observable<EmployeeRaw[]>. Note, the API will return an **array** with **one element** (containing the "raw" Employee object)

### PositionService (position.service.ts)

This service needs to create the following additional 2 methods:

- **savePosition(position: Position)**

  This method must make a "**put**" request (using the HTTPClient module) to your Teams API running on heroku with the path: "/position/id" (where "id" matches the position's "_id" property). It will return an Observable of type any, ie: Observable<any>. **NOTE**: with a "put" request, you can send the data in the second parameter, ie:

  - return this.http.put<any>("someURL/", **someData**);

- **getPosition(id)**

  This method must make a "**get**" request (using the HTTPClient module) to your Teams API running on heroku with the path: "/position/id" (where "id" matches the "id" parameter to the function). It will return an Observable of type Position[], ie: Observable<Position[]>. Note, the API will return an **array** with **one element** (containing the Position object)

## Step 2 - Creating New Components (EmployeeComponent & PositionComponent)

Now that we have our new service methods to fetch a specific employee / position, we can create the components that will enable us to edit existing employees and positions. This involves creating two new components: **EmployeeComponent** & **PositionComponent**.

Use the @angular/cli (ie: ng generate) to create these components now.

Once these components are created, add the following properties / methods according to the specifications outlined below:

### EmployeeComponent (employee.component.css)

To keep the look and feel of our app consistent, we must add the following css:

- .center{ margin-top:40px; }

### EmployeeComponent (employee.component.ts)

Properties:

- paramSubscription ( type any)
- employeeSubscription ( type any)
- getPositionsSubcription ( type any )
- saveEmployeeSubscription ( type any)
- employee ( type EmployeeRaw )
- positions( type Position[] )
- successMessage ( set to false )
- failMessage (set to false)

Methods:

- constructor()
  In this method, we must inject the following Services:
    - EmployeeService from employee.service
    - ActivatedRoute from @angular/router
    - PositionService from position.service
- ngOnInit()
  In this method, we will use Injected services / modules to perform the following tasks:
    - Determine what the value of the _id variable is in the Route parameter using the ActivatedRoute service (Note: a reference to the subscription should be stored using "paramSubscription" so that it can be disposed of later)
    - Use the value of _id to populate the "employee" property using the EmployeeService service (Note: a reference to the subscription should be stored using "employeeSubscription" so that it can be disposed of later)
    - populate the "positions" property using the PositionService service (Note: a reference to the subscription should be stored using "getPositionsSub" so that it can be disposed of later)

- onSubmit()
  In this method, we will use Injected services / modules to perform the following tasks:
    - Persist ("save") the "employee" property using the EmployeeService service (Note: a reference to the subscription should be stored using "saveEmployeeSubscription" so that it can be disposed of later)
    - If the subscription output the data **successfully** (ie, in the **first** callback):
        - Set the value of the successMessage property to **true**
        - Using the setTimeout() method, automatically set the successMessage property to **false** after 2500 ms
    - If the subscription **failed** to output the data (ie, in the **second** callback):
        - Set the value of the failMessage property to **true**
        - Using the setTimeout() method, automatically set the failMessage property to **false** after 2500 ms
- ngOnDestroy
  In this method we call the "unsubscribe()" methods on any saved subscriptions within the component (ie: paramSubscription, etc) - **Note**: we must make sure they are not "undefined" before we call "unsubscribe()"

### PositionComponent (position.component.css)

To keep the look and feel of our app consistent, we must add the following css:

- .center{ margin-top:40px; }

### PositionComponent (position.component.ts)

Properties:

- paramSubscription ( type any)
- positionSubscription ( type any)
- savePositionSubscription (type any )
- position( type Position )
- successMessage ( set to false )
- failMessage (set to false)

Methods:

- constructor()
  In this method, we must inject the following Services:
    - PositionService from position.service
    - ActivatedRoute from @angular/router
- ngOnInit()
  In this method, we will use Injected services / modules to perform the following tasks:

- Determine what the value of the _id variable is in the Route parameter using the ActivatedRoute service (Note: a reference to the subscription should be stored using "paramSubscription" so that it can be disposed of later)
- Use the value of _id to populate the "position" property using the PositionService service (Note: a reference to the subscription should be stored using "positionSubscription" so that it can be disposed of later)

- onSubmit()
  In this method, we will use Injected services / modules to perform the following tasks:
  - Persist ("save") the "position" property using the PositionService service (Note: a reference to the subscription should be stored using "savePositionSubscription" so that it can be disposed of later)
  - If the subscription output the data **successfully** (ie, in the **first** callback):
    - Set the value of the successMessage property to **true**
    - Using the setTimeout() method, automatically set the successMessage property to **false** after 2500 ms
  - If the subscription **failed** to output the data (ie, in the **second** callback):
    - Set the value of the failMessage property to **true**
    - Using the setTimeout() method, automatically set the failMessage property to **false** after 2500 ms

- ngOnDestroy
  In this method we call the "unsubscribe()" methods on any saved subscriptions within the component (ie: paramSubscription, etc) - Note: we must make sure they are not "undefined" before we call "unsubscribe()"

## Step 2 - Creating Routes for the Employees / Positions Components

Now that we have our new components in place and capable of reading route parameters, we must update app-routing-module.ts and our existing EmployeesComponent & PositionsComponent files so that we can successfully navigate to the new routes for a specific employee / position.

### app-routing-module.ts

Add the following routes:

| **Route Path** | **Component / Options** |
|---|---|
| employee/:_id | EmployeeComponent |
| position/:_id | PositionComponent |

### EmployeesComponent (employees.component.ts)

Our goal here is to allow users to click on a specific employee **row**, which will cause them to be sent to the corresponding employee view (using the /employee/:_id route)

To enable this, we must modify the EmployeesComponent according to the following specification

- Inject **Router** from @angular/router in the constructor parameters as "router"

- Add the method: **routeEmployee(id: string)**

    - In this method, we use the Injected router instance to "navigate" the to the /employee/id route (where id is the parameter passed to the function) - **HINT:** see **this.router.navigate** in the "Angular Services Intro" notes

### EmployeesComponent (employees.component.html)

With our new "routeEmployee(id)" method in place, we can now update our EmployeesComponent template (employees.component.html) to invoke the method by binding a "click" event to the same <tr> element that has our *ngFor directive. This click event will invoke the routeEmployee() method and pass in the value of the current employee's _id value, ie "routeEmployee(employee._id)"

By adding the click binding to the repeated <tr> element, we can ensure that every single <tr> element will have the correct event, and invoke the routeEmployee method for the matching employee _id value for that row in the table.

### PositionsComponent (positions.component.ts)

Similar to EmployeesComponent, our goal is to allow users to click on a specific position **row**, which will cause them to be sent to the corresponding position view (using the /position/:_id route)

To enable this, we must modify the PositionsComponent according to the following specification

- Inject **Router** from @angular/router in the constructor parameters as "router"

- Add the method: **routePosition(id: string)**

    - In this method, we use the Injected router instance to "navigate" the to the /position/id route (where id is the parameter passed to the function) - **HINT:** see **this.router.navigate** in the "Angular Services Intro" notes

### PositionsComponent (employees.component.html)

Again, this is the same process as our EmployeesComponent template, ie: we must update our PositionsComponent template (positions.component.html) to invoke the routePosition method by binding a "click" event to the same <tr> element that has our *ngFor directive. This click event will invoke the routePosition() method and pass in the value of the current positions's _id value, ie "routePosition(position._id)"

## Step 3 - Creating the Templates / Forms (EmployeeComponent & PositionComponent)

With our Component logic in place, we can now concentrate on creating the templates for our Employee & Position Components. However, before we start, we need to update our **assets/main.css** file, so that our forms render more clearly in the application:

### assets/main.css

- in the **.form-group . form-control** style (line 736), change the border-color property from: #f2f2f2 to #d2d2d2

- At the **bottom** of the file, **below** all the other styles, add:
    - .table-hover tr{ cursor: pointer; }

With this complete, we can focus on adding our template content / forms to our new Employee & Position Components.

**IMPORTANT NOTE: Before we can use Forms in Angular**, we must:

- Import **FormsModule** from **@angular/forms** in our **app.module.ts**
- Add **FormsModule** to the **imports: []** array in **app.module.ts**

## EmployeeComponent (employee.component.html)

To get you started, we have provided some boilerplate, static HTML that can be used to kick start the template:

https://scs.senecac.on.ca/~patrick.crawford/shared/winter-2019/web422/A6/employee.component.html.txt

Once you have copied the contents of the above .txt file into your employee.component.html file, you should have almost all the fields you will need to update an existing employee:

- With this HTML in place, we have a view that features a blank form and placeholder data. We need to update this view to use real data from the component (ie: employee and positions) for all form fields and any other placeholder locations (ie: in the "alert" elements at the bottom and the <h2>…</h2> element at the top).
- Regarding the form fields, remember to:
    - Add **(ngSubmit)="onSubmit()"** to your **<form>** tag
    - Use two-way binding syntax for form elements, ie: **[(ngModel)]="employee.FirstName"**
    - Use ***ngFor** to iterate over **<option>** elements in a <select> and always use the **[value]=""** syntax to specify a value
- We must also remember to:
    - **conditionally show** the "alert-success" alert only if the "successMessage" property is true
    - **conditionally show** the "alert-danger" alert only if the "failMessage" property is true
    - Use **routerLink** instead of **href** where applicable
    - Use **date:'longDate'** DatePipe ( from Assignment 5 - https://angular.io/api/common/DatePipe ) to format dates where applicable

Once you have completed updating the template, you should be able to modify any employee in the system!

## PositionComponent (position.component.html)

To get you started, we have provided some boilerplate, static HTML that can be used to kick start the template:

https://scs.senecac.on.ca/~patrick.crawford/shared/winter-2019/web422/A6/position.component.html.txt

Once you have copied the contents of the above .txt file into your position.component.html file, you should have almost all the fields you will need to update an existing position:

- With this HTML in place, we have a view that features a blank form and placeholder data. We need to update this view to use real data from the component (ie: position) for all form fields and any other placeholder locations (ie: in the "alert" elements at the bottom and the <h2>…</h2> element at the top).
- Regarding the form fields, remember to:
    - Add **(ngSubmit)="onSubmit()"** to your **<form>** tag
    - Use two-way binding syntax for form elements, ie: **[(ngModel)]="position.PositionName"**
- We must also remember to:
    - **conditionally show** the "alert-success" alert only if the "successMessage" property is true
    - **conditionally show** the "alert-danger" alert only if the "failMessage" property is true
    - Use **routerLink** instead of **href** where applicable

Once you have completed updating the template, you should be able to modify any position in the system!

## Step 4 - Adding Validation

While we can detect errors when we execute code to update an employee or position (ie: try modifying an employee by changing their extension to "abc" - we will see our "alert-danger" element), Angular has a very robust method of handling errors on the **client side**, before a flawed "put" request is even sent (see: Week 10 - Angular Forms Intro).

To enable "client side" validation in our app and show errors to the user as they update the form (as well as prevent them from "submitting" the form if the form has any errors), we must update the forms in the employee.component.html & position.component.html files, according to the following specification:

1. Add the CSS from here: https://scs.senecac.on.ca/~patrick.crawford/shared/winter-2019/web422/A6/validation.css.txt to the end of the **src/styles.css** file (this will ensure that the elements are correctly highlighted
2. Add the class "control-label" to all <label> elements within a "form-group"
3. Add "Template Reference Variables" to all editable form control elements except "Position" (we will need these to check for validation errors)
4. Add the "Template Reference Variable": "**f**" to the <form> element (we will need this to disable/enable the "submit" button
5. Ensure the following fields are validated (Using HTML5 validation attributes) according to the following specification:

### Employee Form

| Input Name | Type of Validation |
|---|---|
| First Name | Required Field |
| Last Name | Required Field |
| Salary Bonus | Required Field (Input Type: Number) |
| Address (Street) | Required Field |
| Address (City) | Required Field |
| Address (State) | Required Field |
| Address (Zip) | Required Field |
| Phone Number | Must Match the following Regular Expression (used for American phone numbers): **\+?[ ]*[1-9]?[ ]*\-?[ ]*\(?[ ]*[1-9][ ]*(\d[ ]*){2}\)?[ ]*\-?[ ]*(\d[ ]*){3}-[ ]*(\d[ ]*){4}** |
| Extension | Required Field (Input Type: Number) |

### Position Form

| Input Name | Type of Validation |
|---|---|
| Position Name | Required Field |
| Position Description | Required Field |

6. Update all form elements so that when they're in an "error" state, they will become highlighted and show a relevant error to the user. **Note**: you can reference the following example showing what a simplified Employee "FirstName" field (ie: no Template Reference Variable or validation attributes) looks like when it's **not in error** vs. **when it's in an error state** (the differences are highlighted in blue). You will need to update all elements so that they behave in this manner.

   **NOTE**: To achieve this transformation conditionally (ie, only when the form element is truly in error), we will need to use the following techniques: class binding, *ngIf & the "**errors**" property of a "Template Reference Variable" for a form element (see the "Angular Forms Intro" notes)

   #### "First Name" *not in* Error:

```
<div class="form-group">
    <label class="control-label" for="FirstName">First Name:</label>
    <input class="form-control" id="FirstName" type="text" name="FirstName" required />
</div>
```

### "First Name" *in* Error (ie: it's identified as "required", but currently empty)

```
<div class="form-group has-error">
    <label class="control-label" for="FirstName">First Name:</label>
    <input class="form-control" id="FirstName" type="text" name="FirstName" required />
    <span class="help-block">First Name is Required</span>
</div>
```


Notice how we add the "has-error" class to the "form-group", as well as add a special "span" element to the "form-group"

7. Lastly, add the following "disabled" property binding to the "submit" button: **[disabled]="!f.valid" -** this uses your Template Reference Variable from your <form> element (ie: "**f**") to conditionally disable the submit button if the form as a whole is invalid

## Part 2 - Filtering Employees

Currently, our assignment allows us to perform edit/update operations on employees fairly easily. Unfortunately, finding a specific employee is not especially user-friendly. The user needs to do a ctrl/cmd + F to search for employees. It would be better if we added a text field in our employees view to allow users to filter the list of employees

### Step 1: Adding the Search Field

In your employees.component.tml file, add the following element immediately **above** the … element:

```
<input class="form-control" id="EmployeeSearch" name="EmployeeSearch" type="text" placeholder="Search Employees by Full Name or Position" (keyup)="onEmployeeSearchKeyUP($event)" /><br />
```

### Step 2: Add a filteredEmployees Property & Update our EmployeesComponent View

To display a subset of the employees (ie: "filtered" employees), we need to add an additional property to our EmployeesComponent, ie:

- filteredEmployees: Employee[]

We must also be sure to populate the filteredEmployees array with all returned employees from our **getEmployees()** service call (in **ngOnInit()**)

Lastly, instead of iterating over all **employees** in our **employees.component.html** template (using *ngFor), we must instead iterate over all **filteredEmployees**. This will allow us to modify the **filteredEmployees** array to display a subset of the returned **employees** array.

### Step 2: Add the Event Handler (onEmployeeSearchKeyUP(event:any)

In this method, we can get the value of the related input element (that invoked this method on "keyup") using **event.target.value.** It's this value that we can use to filter the full **employees** array into the filtered **filteredEmployees** array property. In this case, we want to filter the **employees** array to **only** include employees that:

- Have a FirstName value included in the search string (**event.target.value**), **or**
- Have a LastName value included in the search string (**event.target.value**), **or**
- Have a Position.PositionName value included in the search string (**event.target.value**)

Note: This comparison must be **case-insensitive** (**Hint:** You can normalize the case using "toLowerCase()")

**Hint**: you can make use of the this.filteredEmployees = this.employees.filter(…) method (see this link for reference)

## Part 3 - Publishing the Application

### Step 1: Preparing the code to be "built" / building the app for Production

Unfortunately, if we try to do a production build (ie: --prod See week 11 Notes: Angular Deployment Intro) of our code right now, we may see some errors, ie:

ERROR in src/app/employee/employee.component.html(2,7): : Property 'employee' is private and only accessible within class 'EmployeeComponent'.

To remedy this and the countless other errors that may occur like it, we must ensure that any variable declared in a Component's **class**, that's also used in the Component's **template** is **not private**. To fix the error above, all we need to do is open the **employee.component.ts** file and change the line: **private employee: EmployeeRaw;** to simply **employee: EmployeeRaw** and the error will disappear the next time you try to build the app for production.

### Step 2: Writing a Static Server & pushing to Heroku

Finally, the app has built and we have a fresh "dist" folder containing the files. The final task is to write a "static server" in a new "server" directory using Node.js / Express to serve a "public" folder containing the contents of the newly created "dist" folder.

For instructions on how to create this server and place it on Heroku, refer to the "Angular Deployment Introduction" notes and/or the "Getting Started with Heroku" guide from WEB322.

## Assignment Submission:

- Add the following declaration at the top of your app.component.ts file:

```
/*********************************************************************************
* WEB422 – Assignment 06
* I declare that this assignment is my own work in accordance with Seneca  Academic Policy.  No part of this
* assignment has been copied manually or electronically from any other source (including web sites) or
* distributed to other students.
*
* Name: _____ Student ID: _____ Date: _____
*
* Heroku Link: _____
*
*********************************************************************************/
```

- Compress (.zip) the all files in your Visual Studio code folder **EXCEPT the node_modules folder** (this will just make your submission unnecessarily large, and all your module dependencies should be in your package.json file anyway).
- Submit your compressed file (without the node_modules folder) to My.Seneca under **Assignments** -> **Assignment** 6

## Important Note:

- **NO LATE SUBMISSIONS** for assignments. Late assignment submissions will not be accepted and will receive a **grade of zero (0)**.
- After the end (11:59PM) of the due date, the assignment submission link on My.Seneca will no longer be available.
- Submitted assignments must run locally, ie: start up errors causing the assignment/app to fail on startup will result in a **grade of zero (0)** for the assignment.