Developing with Node.js, NPM and Visual Studio Code

Lab Time: 60 minutes

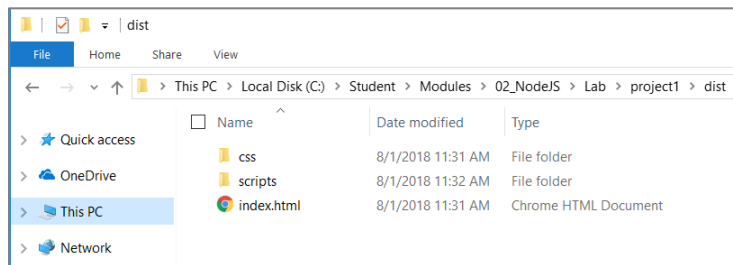**Lab Folder**: C:\Student\Modules\02_NodeJS\Lab

Lab Overview: As you work through this lab, you will get a healthy dose of hands-on experience working in the Node.js development environment and managing software projects using Node Package Manager (npm). You will learn how to use npm to initialize new projects and to install the NodeJS packages required to provide a local web server used for testing and debugging. You will also learn to configure TypeScript support and to write and execute custom developer tasks using gulp. In the final exercise, you will learn to use the webpack utility to bundle the source files for your project into a streamlined set of files for distribution.

Lab Prerequisite: This lab assumes you've already installed Node.js, GIT and Visual Studio Code as described in setup.pdf.

Exercise 1: Getting Started with Node.js, NPM and Visual Studio Code

In this exercise you will learn the fundamentals of initializing and managing projects using Node.js and npm. You will begin with a folder that contains an HTML file, a CSS file and a JavaScript file.

1. Inspect the project starter files in the folder named **project1**.

   a) Using Windows Explorer, open the folder at **c:\Student\Modules\02_NodeJS\Lab\StarterProjects\project1**.

   b) Make a copy of the project1 folder outside the **StarterFiles** project **c:\Student\Modules\02_NodeJS\Lab\project1**

   c) If you look inside the **project1** folder you just created, you will find a child folder named **Dist**.

   d) you look inside the **Dist** folder, you will find an **index.html** file at the root. There is also a CSS file named **app.css** inside the **css** folder and a JavaScript file named **app.js** inside the **scripts** folder. Also note that the **css** folder contains a child folder named **img** that contains an image file named **AppIcon.png**.


2. Open the **project1** folder with Visual Studio Code.

   a) Launch Visual Studio Code.

   b) Use the **File >> Open Folder** command to open the folder at **c:\Student\Modules\02_NodeJS\Lab\project1**.

   c) Use Visual Studio Code to open the **Dist/index.html** in an editor window. Take a moment to review the HTML code inside.
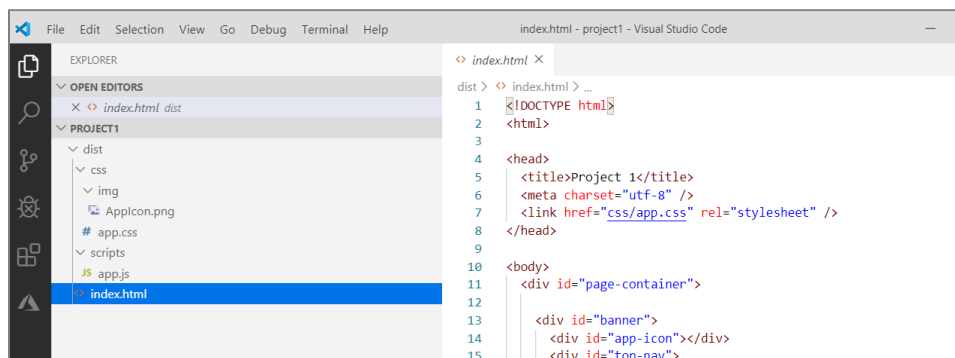

   d) Open the CCS file named **app.css** in the **Dist/css** folder and quickly review the CSS code that's inside.

   e) Open the JavaScript file named **app.js** in the **Dist/scripts** folder and quickly review the JavaScript code that's inside.

   f) After you have inspected each of these three source files, close all of them so that no editor windows are open.

3. Initialize the project1 folder as a Node.js project.

   a) Use the **View > Terminal** menu command to display the Integrated Terminal.
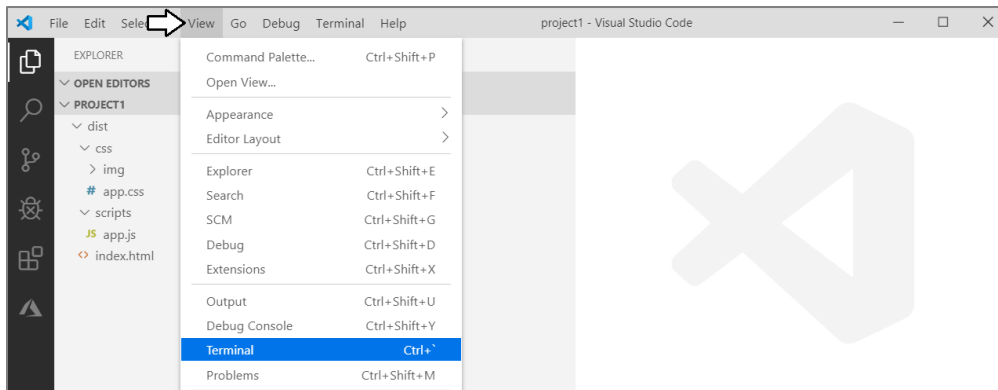
   b) You should see command line interface of the Terminal as shown in the following screenshot.
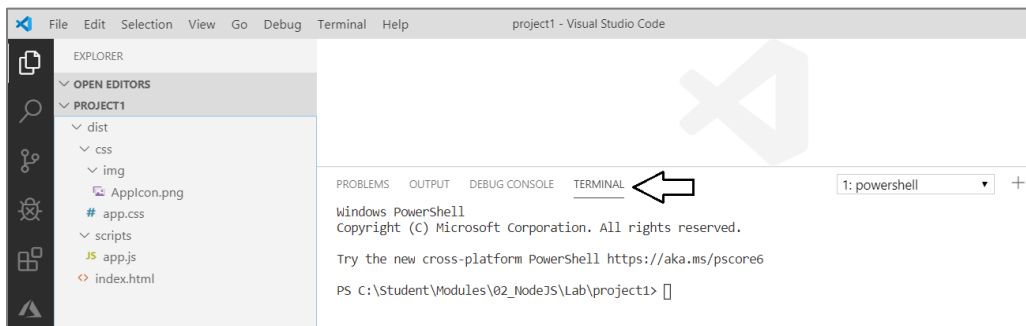
   c) From the command line of the Integrated Terminal, type the following command and then press **Enter** to execute it.

```
npm init –y
```

   d) After the command completes, you should see that a new file has been added to your project named **package.json**.
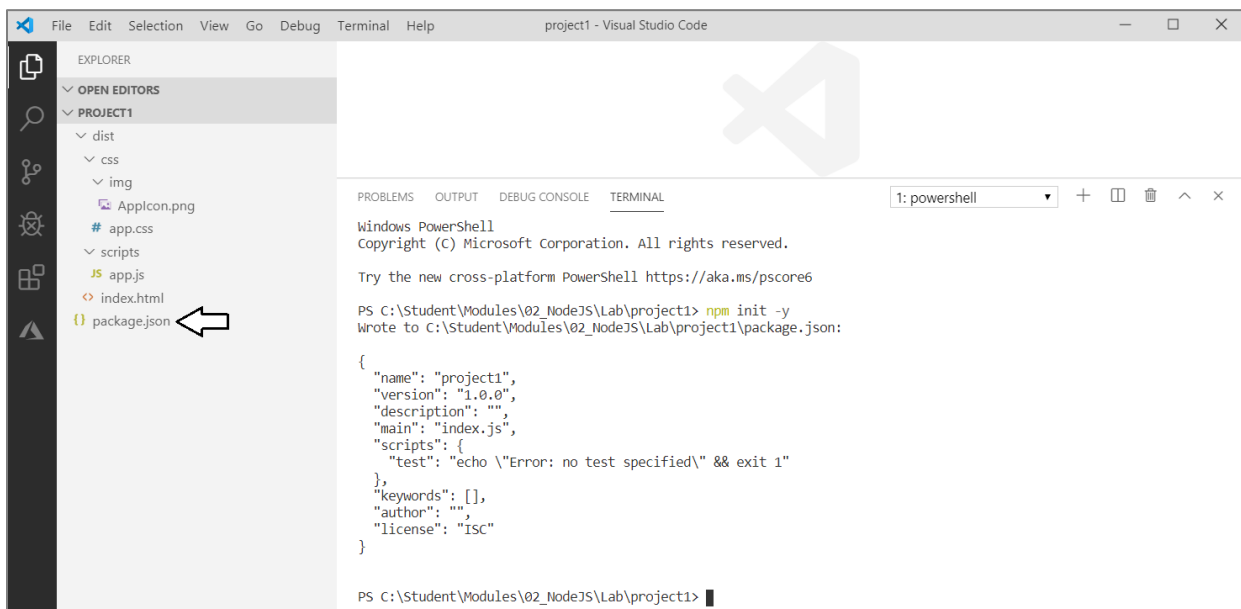
e) Open the **package.json** and see what's inside. There is no need for you to make any changes to this file.
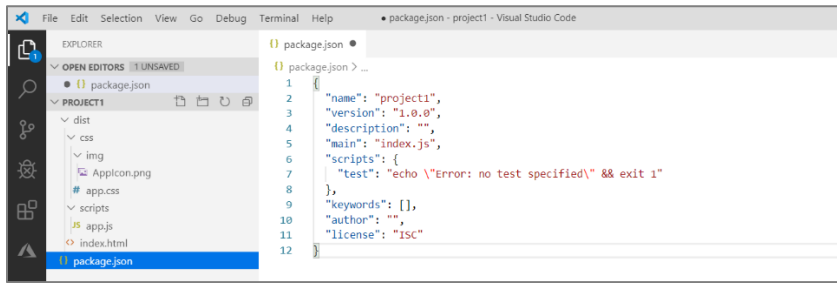
When you are done looking at the **package.json** file, leave it open. You will see that this file will be automatically updated by the npm utility when you install new packages.

4. Install a new npm package named **browser-sync** to provide a local web server for testing and debugging.

   a) Return to the command line of the Integrated Terminal

   b) From the command line, type the following **npm install** command and then execute it by pressing the **Enter** key.

   ```
   npm install browser-sync --save-dev
   ```

   c) When you execute this command, wait until it completes which should take about 20-30 seconds.

   ```
   PS C:\Student\Modules\02_NodeJS\Lab\project1> npm install browser-sync --save-dev          npm notice
    created a lockfile as package-lock.json. You should commit this file.
   npm WARN project1@1.0.0 No description
   npm WARN project1@1.0.0 No repository field.
   npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.9 (node_modules\fsevents):
   npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.9: wanted {"os":"darwin","arch":"any"} (
   current: {"os":"win32","arch":"x64"})

   + browser-sync@2.26.7
   added 331 packages from 262 contributors and audited 3498 packages in 28.912s
   found 0 vulnerabilities
   ```

   What the Heck! There is a message in the console that says over 200 packages were installed. And you thought you were just installing a single package. What's the story? Well, the browser-sync package has dependent packages that are also installed. And then the packages that browser-sync depends on have their own dependent packages as well and so on. The dependency tree for the browser-sync packages contains over 200 other packages.

   d) If you look in the **devDependencies** section inside the **package.json** file, you can see a new entry that tracks the minimum version number for the browser-sync package.


   e) You will notice that a new file has just been created named **package-lock.json**. If you look inside this file, you will see that it contains entries for all the packages that were installed including the actual version number for each package.
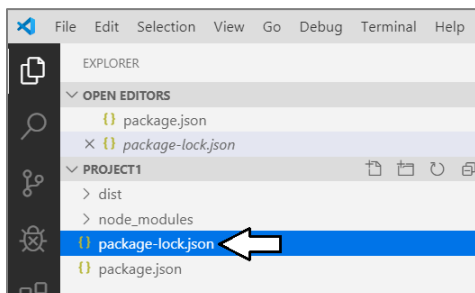
5. Execute the **npm shrinkwrap** command to convert the **package.lock.json** file into the **npm.shrinkwrap.json** file.

    a) Return to the command line of the Integrated Terminal.

    b) From the command line, type the following **npm shrinkwrap** command and then execute it by pressing the **Enter** key.

```
npm shrinkwrap
```

    c) You should see that executing this command replaces the **package.lock.json** file with the **npm.shrinkwrap.json** file
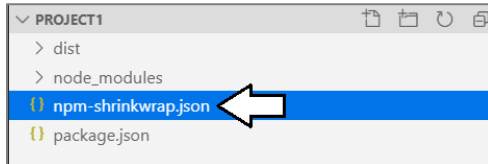
6. Examine the **node_modules** folder which contains the installed package files.

    a) Inside the root folder for **project1**, you should also notice that that new folder has been created named **node_modules**.
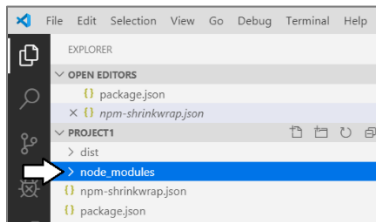
    b) If you expand the **node_modules** folder, you will see many child folders which Node.js packages that have been installed.
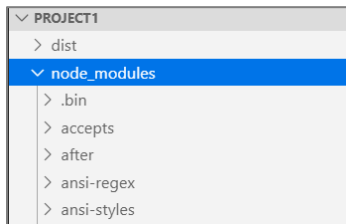

7. Install the **browser-sync** package globally to make it easier to call its command-line interface (CLI) from the console window.

    a) From the command line, type and execute the following **npm install** command to perform a global installation of the package.

```
npm install –g browser-sync
```

8. Use the web server support in browser-sync to run and test the application in the browser.

    a) Navigate back to the command line in the Integrated Terminal.

    b) Type the following command and then execute it by pressing the **Enter** key.
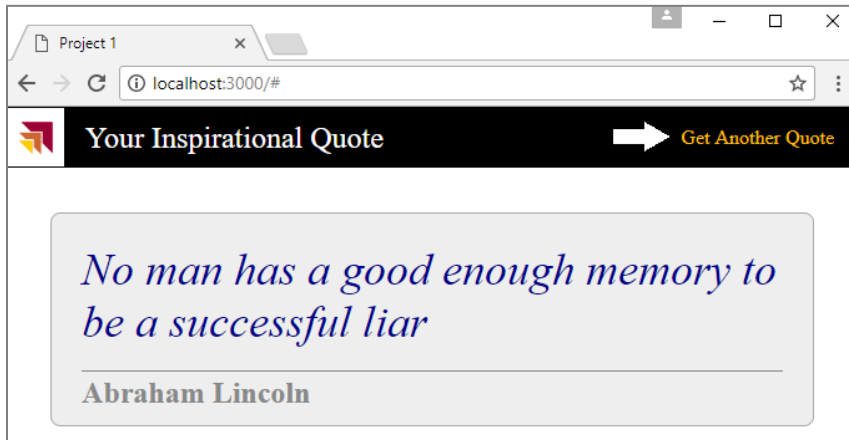
```
browser-sync start --server dist
```

    c) You should see the browser-sync web server process start and initialize in the console window.
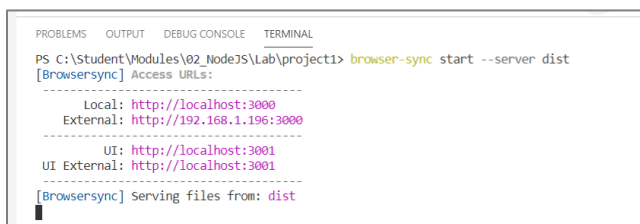
d) Next, browser-sync support should launch the browser and display the simple web application defined by **index.html**.
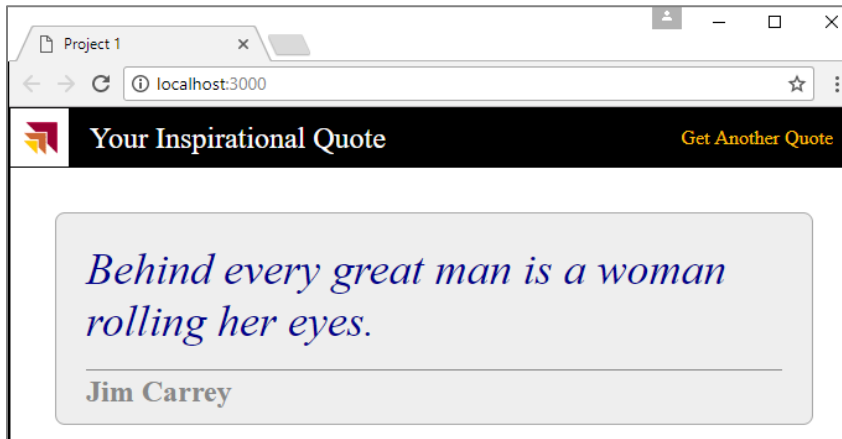
e) Test the application by clicking the **Get Another Quote** link.

As you can see, this application is very simple. It is built using an HTML file, a CSS file and a JavaScript file. The only outside library that this project depends on is jQuery. Note that index.html has a script link to download jQuery from a public CDN on the Internet.
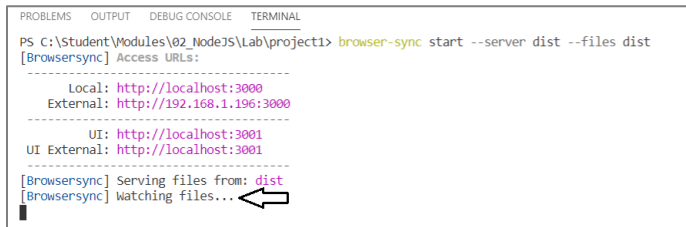
9. Stop the Node.js web server to terminate the current debugging session.

a) Return to the command line of the Integrated Terminal.

b) You should see that there is no command prompt because the console is blocking on the Node.js web server process.

c) Make sure the console window is the active window and then press the **Ctrl + C** keyboard combination.

d) If the console window responds by prompting you to terminate the current batch job.

e) If prompted to confirm, type **y** then then press the **Return** key to stop the web server and the current debugging session.

You have just learned how to stop the web server and terminate the current debugging session. You will be doing this on a regular basis because you often need to work at the command line and execute additional npm commands to install new packages. In the future, the lab instructions will just tell you to stop the web server debugging session and you must remember that this is accomplished using **CTRL+C**.

10. Run browser-sync one more time, except this time add file watch support to refresh the browser whenever you update source files.

a) Return to the Integrated Terminal in Visual Studio Code.

b) Type the following command and then press **Enter** to execute it.

```
browser-sync start --server dist --files dist
```

c) Note the output in the console as the web server process starts. You should see there is a file watch in effect.

d) Once the browser starts, place it side by side to Visual Studio Code.

e) Make an update to index.html and then save your changes. You should be able to see that the browser automatically refreshes to show you change whenever you save your changes.
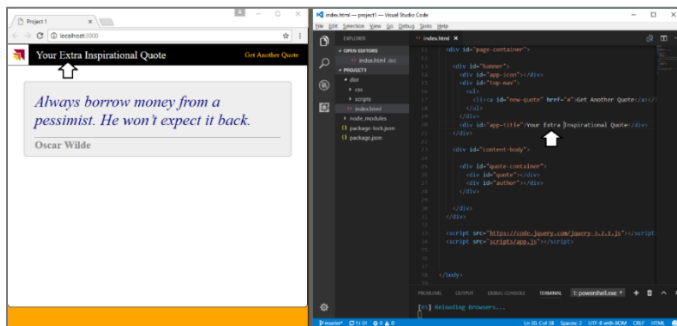
f) Return to the Integrated Terminal in Visual Studio Code and stop the web server debugging session.

Exercise 2: Moving from JavaScript to TypeScript

In this exercise, you will migrate the application's client-side code from JavaScript to TypeScript. This will involve creating a new TypeScript source file named app.ts and adding support to compile that TypeScript file into an output file named app.js.

1. Review the current structure of **project1**.

   a) Currently, the **app.js** file existing inside a folder named **scripts** which exists in the **dist** folder.
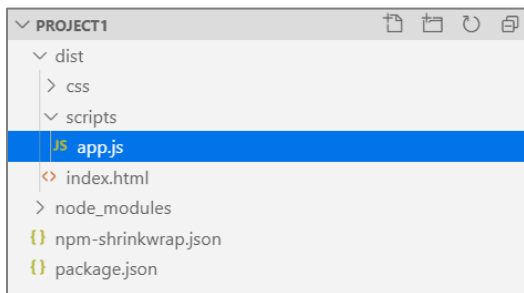
   b) Create a new top-level folder in the project named **src**.
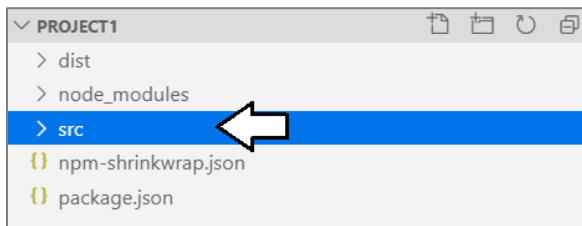
c) Using a drag-and-drop operation with the mouse, move the **scripts** folder (and the **app.js** file inside) from the **dist** folder over to the **src** folder.
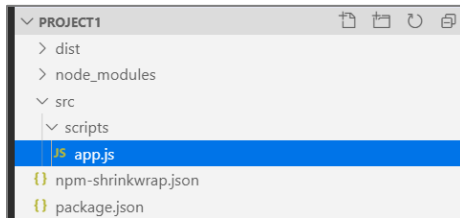
d) Change the file extension by renaming the file from **app.js** to **app.ts**.

Now your project has a TypeScript source file. However, the project doesn't yet have any support yet for compiling this TypeScript source file into JavaScript code for testing and for distribution. In the next step, you will install the **typescript** package which adds the TypeScript compiler named **tsc.exe**.

2. Install the **typescript** package into your project.

   a) Return to the Integrated Terminal in Visual Studio Code.

   b) Type the following command and execute by pressing Enter.

   ```
   npm install typescript --save-dev
   ```

   c) The command should install the latest version of the **typescript** package.

Once you've installed the **typescript** package, the TypeScript compiler (**tsc.exe**) is available from the Integrated Terminal console.

   d) Return to the console in the Integrated Terminal.

   e) Type and execute the following **npx** command to display the project-specific version of the installed **typescript** package.

   ```
   npx tsc --version
   ```

Note that you are using to **npx** command to execute the version of **tsc.exe** that exists within the current project folder. Note that some early versions of the **npx** utility return a warning message "Path must be a string. Received undefined" which you can ignore.

   f) Verify the **tsc** version number. The following screenshot shows a version number of **3.7.2** but yours may be more recent.

Now your project is configured to use the TypeScript compiler. However, you are not quite ready to compile the TypeScript code inside **app.ts**. First you need to install a new package which contains the Typed Definition file for the jQuery library.

3. Install the **@types/jquery** package to add the Typed Definition file for the jQuery library.

   a) Return to the console in the Integrated Terminal and run the following command.

   ```
   npm install @types/jquery --save-dev
   ```

   b) After running this command, look inside the **node_modules** folder and you should see a new folder **@types/jquery** which contains a file named **index.d.ts**. This is the Typed Definition file for the jQuery library.

   ```
   ▲ node_modules
     ▷ .bin
     ▲ @types
       ▲ jquery
         TS index.d.ts
         🍴 LICENSE
         {} package.json
         ⓘ README.md
   ```
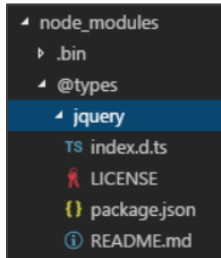
Now you are ready to compile your TypeScript source file into JavaScript code using the TypeScript compiler. The first step will be to generate a **tsconfig.json** file which will allow you to configure the input and output of the TypeScript compiler.

4. Create a **tsconfig.json** file in your project to configure how the **tsc** utility compiles your TypeScript into JavaScript.

   a) Run the following command from the console to generate a new **tsconfig.json** file at the root of your project.

   ```
   npx tsc --init
   ```

   b) After running this command, you should see that a new **tsconfig.json** file has been created at the root of your project.
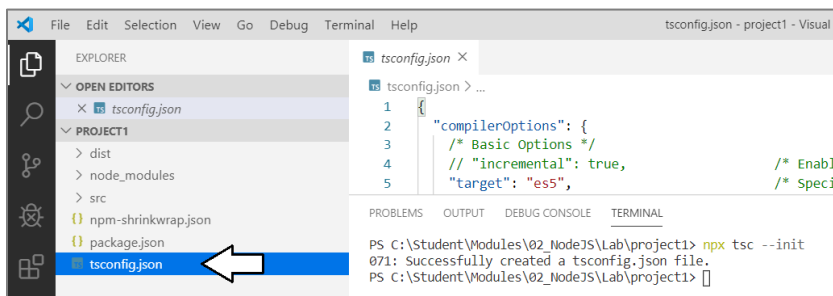
   c) Open the **tsconfig.json** file in an editor window.

   d) Replace the contents of the **tsconfig.json** file with the following JSON code.

   ```
   {
     "compilerOptions": {
       "noImplicitAny": true,
       "removeComments": true,
       "preserveConstEnums": true,
       "outFile": "./dist/scripts/app.js",
       "sourceMap": true,
       "lib": [ "dom", "es6" ]
     },
     "files": [
       "./src/scripts/app.ts"
     ],
     "exclude": [
       "node_modules"
     ]
   }
   ```

> This JSON code is also available a file named **tsconfig.json.txt** in the **StarterFiles** folder.

    e)   Save your changes to **tsconfig.json**.

    f)   Close **tsconfig.json**.

5. Run the TypeScript compiler to generate an output JavaScript file named **app.js**.

    a)   Return to the console in the Integrated Terminal.

    b)   Execute a command by calling **npx tsc** without passing any parameters.

```
npx tsc
```

    c)   After you run the command, you should be able to verify that the **app.js** file and a second debugging file named **app.js.map** have been created in the **dist/scripts** folder.

> Open the **app.js** that has been generated by the TypeScript compiler so you can see what the generated looks like. You will find that your TypeScript code has been compiled into JavaScript code that's EcmaScript5 compatible so it runs in all today's popular browsers.
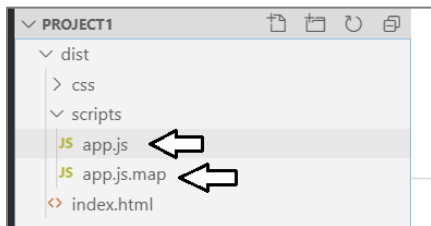
    d)   Close **app.js** after you have looked through the code inside.

6. Test out the application to make sure that the generated JavaScript code runs as you expect it to.

    a)   Return to the console in the Integrated Terminal.

    b)   Execute the following command to start the web server and launch the browser.

```
browser-sync start --server dist
```

    c)   Test the application to verify that it works as it did before.

> Keep in mind you didn't really do anything other than rename a source file from **app.js** to **app.ts**. This simple TypeScript migration approach works because any valid JavaScript code *(including the JavaScript code inside app.js)* is valid TypeScript code. However, your TypeScript code doesn't take advantage of any TypeScript language features such as classes and strongly-typed programming.

    d)   Cancel the debugging session from the Integrated terminal.

7. Replace the code in **app.ts** with a refactored design using TypeScript classes.

    a)   Find the file named **app.ts.txt** in the folder named **StarterFiles**.

    b)   Open **app.ts.txt** and copy its contents into the Windows clipboard.

    c)   Return to **project1** in Visual Studio Code and open the **app.ts** file in the **src** folder.

    d)   Delete the contents of **app.ts** and replace it by pasting in the contents of the Windows clipboard.

    e)   Save your changes to **app.ts**.

8. Inspect the TypeScript code that has been added to **app.ts**.

    a)   At the top of **app.ts**, there is a class named **Quote**.

```
class Quote {
    value: string;
    author: string;
    constructor(value: string, author: string){
        this.value = value;
        this.author = author;
    }
}
```

b) Next, you will see another class named **QuoteManager**.

```
class QuoteManager {
  // my list of quotes
  private  static quotes: Quote[] = [
    new Quote("Always borrow money from a pessimist. He won't expect it back.", "Oscar Wilde" ),
    new Quote("Behind every great man is a woman rolling her eyes.", "Jim Carrey" )
    // other quotes omited for brevity
  ];

  public static getQuote = () : Quote => {
    var index = Math.floor(Math.random()*QuoteManager.quotes.length);
    return QuoteManager.quotes[index];
  }

}
```

c) At the bottom of **app.ts**, there is code which uses the jQuery document ready event to display a quote and to wire up an event handle to respond to the user command for a new quote.

```
$( () => {

  var displayNewQuote = (): void => {
    var quote: Quote = QuoteManager.getQuote();
    $("#quote").text(quote.value);
    $("#author").text(quote.author);
  };

  $("#new-quote").click( ()=> {
    displayNewQuote();
  });

  displayNewQuote();

});
```

d) There is no need for you to modify any of the code in **app.ts**. However, you should take note that the TypeScript editor window in Visual Studio Code will provide IntelliSense and strongly-typed programming against the jQuery library. For example, Visual Studio Code automatically provides IntelliSense for the jQuery library as shown in the following screenshot.
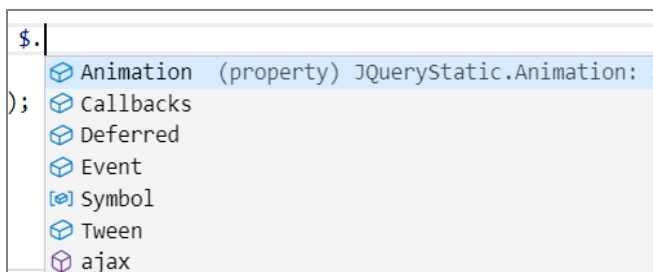
9. Compile the refactored TypeScript code.

a) the **npx tsc** command from the Integrated Terminal.

```
npx tsc
```

b) Rerun the application and make sure everything still works correctly.

c) Run this

```
browser-sync start --server dist
```

d) The app should run just like it did before.

e) Cancel the debugging session from the Integrated terminal.

The last thing you will do in this exercise is to add support for debugging TypeScript code inside Visual Studio Code.

10. Install the **Debugger for Chrome** extension for Visual Studio Code.

   a) Navigate to Visual Studio Code.

   b) Click on the **Extensions** tab in the left navigation

   c) Run a search for extensions using the text "Chrome".

   d) Find and install the **Debugger for Chrome** extension.
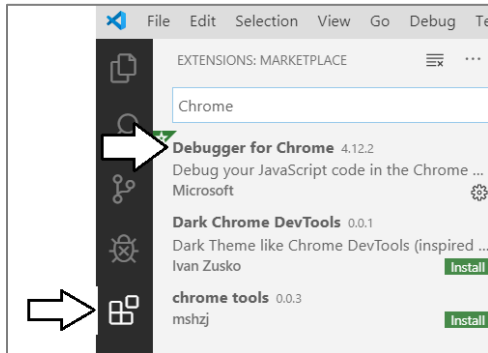
   e) You should be able to see the **Debugger for Chrome** in the **Enabled** section.
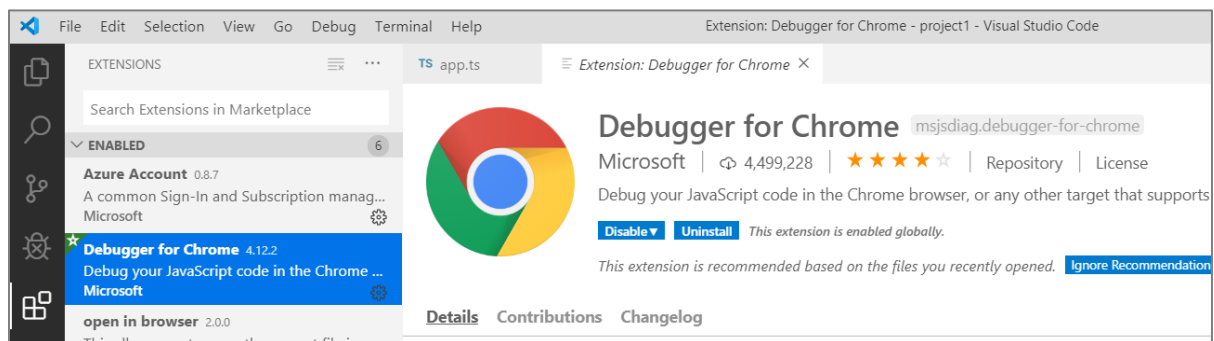
11. Configure support in Visual Studio Code for debugging client-side TypeScript code.

   a) Click on the **Debug** tab in the left navigation.
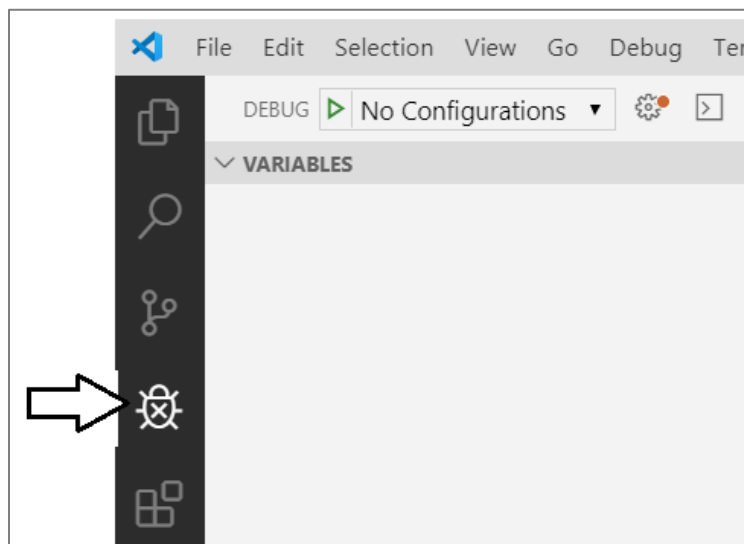
b) Drop down the **Configuration** menu and select **Add Configuration…**.
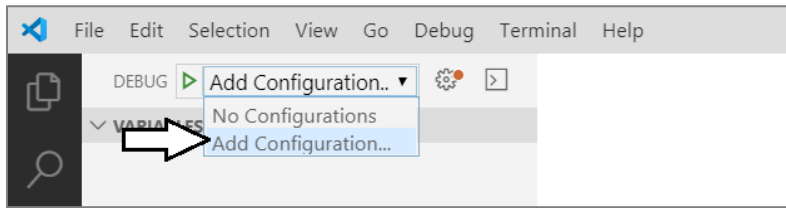
c) From the **Select Environment** menu, select **Chrome**.
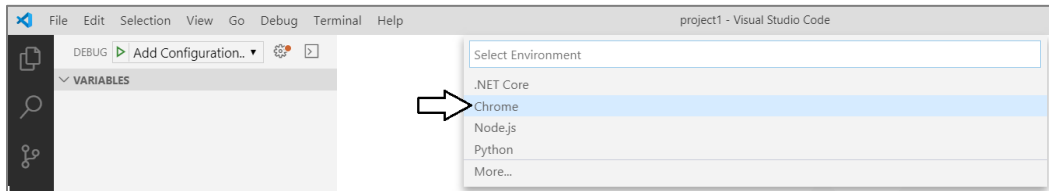
d) You will notice that a new file named **launch.json** has been added to the **.vscode** folder in project.

e) Replace with contents of launch.json with the following JSON code.

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Local Debugging Session",
            "type": "chrome",
            "request": "launch",
            "url": "http://localhost:3000/",
            "webRoot": "${workspaceRoot}/dist",
            "sourceMaps": true,
            "runtimeArgs": [
                "--remote-debugging-port=9222"
            ]
        }
    ]
}
```

This JSON code is also available a file named **launch.json.txt** in the **StarterFiles** folder.

f) Once you have added the JSON code to match the following screenshot, save and close **launch.json**.
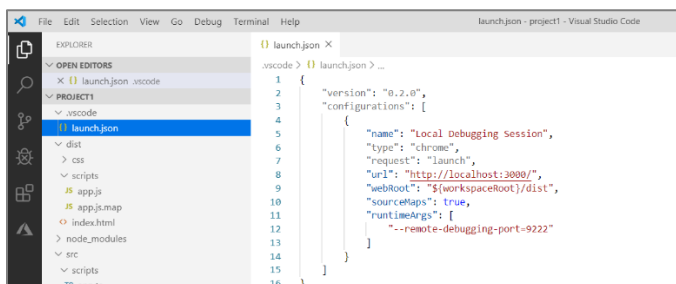
12. Start a debugging session with Visual Studio Code.

   a) Return to the console in the Integrated Terminal.

   b) Run the **browser-sync start** command using the **--no-open** argument to start the web server without launching the browser.

```
browser-sync start --server dist --no-open
```

   c) Click the Debug tab in the left navigation.

   d) Click the button with the green arrow to begin a debugging session.
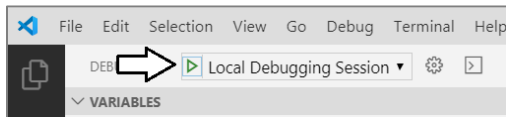
   e) The application should launch in the browser. Verify that is works like it did before.

Now you are going to set a breakpoint to test see if you can single step through your code using the Visual Studio Code debugger.

   f) Open the **app.ts** file from the **src/script** folder in a code editor window.
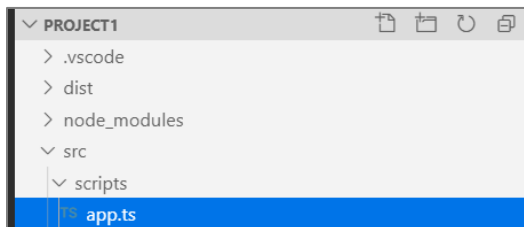
   g) Select the first line of code in the **displayNewQuote** method and set a breakpoint by pressing the **{F9}** key.
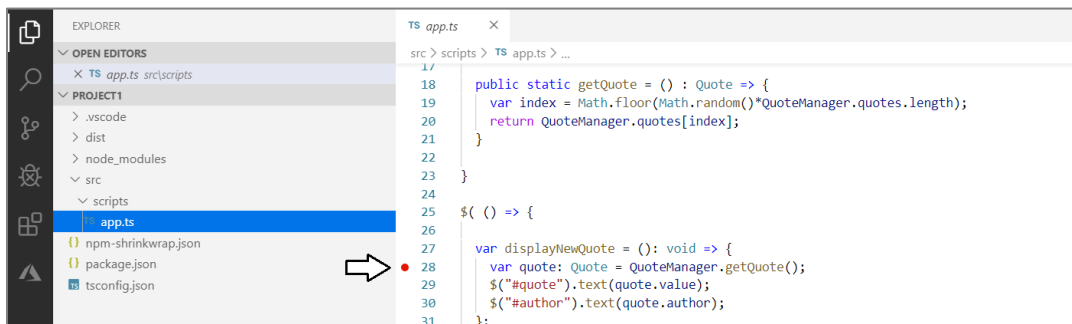
   h) Click on the **Get Another Quote** link to test your breakpoint. At this point, the Visual Studio Code Debugger should break at the line of code where you set the breakpoint.
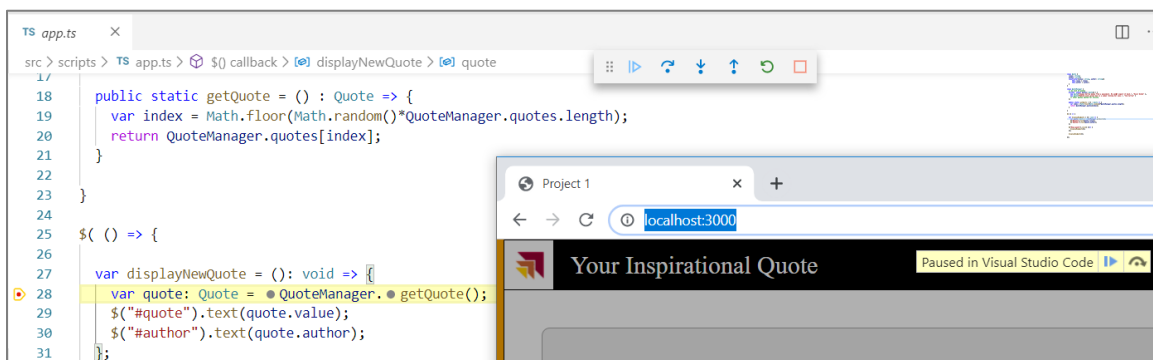
     i)   Click the **Step Into** button on the debugging toolbar as shown in the following screenshot to single step through your code. Also take note of all the debugging information available in the debugging tab while you are single-stepping through your code.

```
TS app.ts      ×

⋮  ▷  ➡  ↓  ↑  ⟳  □     allback ⟩ [∅] displayNewQuote ⟩ [∅] quote

    25    $( () Step Into (F11)
    26
    27       var displayNewQuote = (): void => {
 ●  28          var quote: Quote = ● QuoteManager. ● getQuote();
    29        $("#quote").text(quote.value);
    30        $("#author").text(quote.author);
    31      };
```
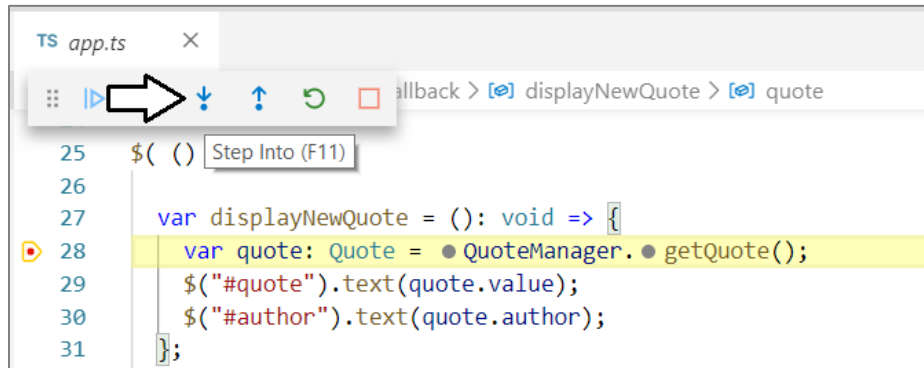
Once you have successfully gotten the Visual Studio Code debugger to single step through your code, you are done with this exercise.

Exercise 3: Integrating Gulp Tasks into a Project

In this exercise, you will install the **gulp** task runner utility and learn how to write and execute custom tasks. You will begin by restructuring the files within your project so all the source files are kept inside the **src** folder. Then you will write gulp tasks to build out the **dist** folder with all the files that need to distributed with your project.

1.   Restructure the project so that all editable source files are kept in the **src** folder.

    a)   Review the current project structure. You will see that several of the project's files of currently stored in the **dist** folder including **index.html**, **app.css** and **AppIcon.png**.

```
∨ PROJECT1                    🗋 🗂 ↻ ⧉
  ⟩ .vscode
  ∨ dist
    ∨ css
      ∨ img
        🖼 AppIcon.png
      # app.css
    ⟩ scripts
    ⟨⟩ index.html
```
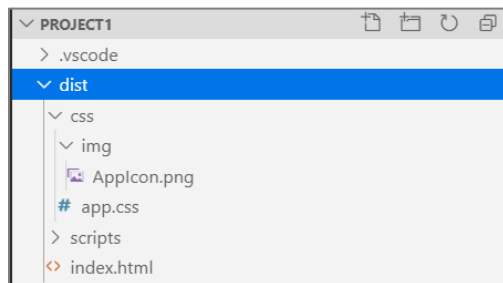
    b)   Using a drag-and-drop operation with the mouse, move the **index.html** file from the **dist** folder to the **src** folder.

    c)   Using a drag-and-drop operation with the mouse, move the **css** folder from the **dist** folder to the **src** folder.

    d)   The contents of the **src** folder in your project should now match the following screenshot.

```
∨ PROJECT1
  ⟩ .vscode
  ∨ dist
  ⟩ node_modules
  ∨ src
    ∨ css
      ∨ img
        🖼 AppIcon.png
      # app.css
    ∨ scripts
    TS app.ts
    ⟨⟩ index.html
```
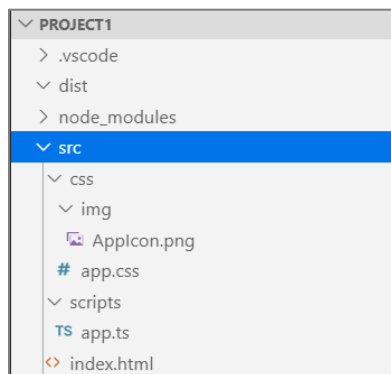
2. Install the **gulp** task runner utility using **npm**.

    a) Return to the console in the Integrated Terminal.

    b) Run the following **npm install** command to perform of project-specific installation of the **gulp** package version 3.

    ```
    npm install gulp@3 --save-dev
    ```

Note that there is a version 4 release of gulp, this lab will use the latest 3.x version of gulp. The SharePoint Framework as of version 1.9.1 still use version 3 and has not yet moved to gulp version 4. This lab is still using version 3 to stay consistent with the SharePoint Framework.

    c) Wait until the **npm install** command completes.

If your attempt to install gulp fails, try again using the **--force** parameter (e.g. **npm install gulp --save-dev --force**)

    d) Install the **gulp** version 3 package globally so you can directly call the **gulp** CLI from the console window.

    ```
    npm install -g gulp@3
    ```

    e) Try running the **gulp** command from the console without passing any arguments.

    ```
    gulp
    ```

    f) You should see that running this command causes an error because you have not yet created a special file named **gulpfile.js**.

3. Create a file named **gulpfile.js** and add the "hello world" gulp task.

    a) Create new file in the root folder of the project named **gulpfile.js**.
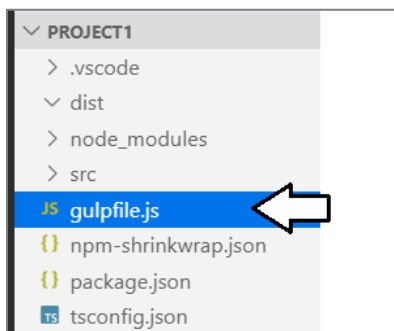
b) Add the following JavaScript code into **gulpfile.js** to create a simple gulp task named **default**.

```
var gulp = require('gulp');

gulp.task('default', function() {
   console.log("Running my very first gulp task")
});
```

c) Save your changes to **gulpfile.js**.

d) Return to the console in the Integrated Terminal and run the **gulp** command again without any arguments. You should be able to verify that the **default** task ran and logged the **"Running my very first gulp task"** message to the console window.

```
PS C:\Student\Modules\02_NodeJS\Lab\project1> gulp
[21:07:00] Using gulpfile C:\Student\Modules\02_NodeJS\Lab\project1\gulpfile.js
[21:07:00] Starting 'default'...
Running my very first gulp task
[21:07:00] Finished 'default' after 507 µs
PS C:\Student\Modules\02_NodeJS\Lab\project1>
```

e) Remove the **default** task you add to **gulpfile.js** file so the file contains only the following line of code.

```
var gulp = require('gulp');
```

4. Create a new gulp task named **clean** to delete any pre-existing files in the **dist** folder.

a) Return to the console in the Integrated Terminal and run the following command to install the **gulp-clean** package.

```
npm install gulp-clean --save-dev
```

b) Wait for the **npm install** command to complete.

```
PS C:\Student\Modules\02_NodeJS\Lab\project1> npm install gulp-clean --save-dev
npm WARN project1@1.0.0 No description
npm WARN project1@1.0.0 No repository field.
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.9 (node_modules\fse
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@

+ gulp-clean@0.4.0
added 18 packages from 30 contributors and audited 4691 packages in 4.158s
```

c) Return to the editor window with **gulpfile.js**.

d) Add a new line to create variable named **clean** and initialize it with a call to **require('gulp-clean')**.

```
var gulp = require('gulp');
var clean = require('gulp-clean');
```

e) Move down below in **gulpfile.js** and add the following code to create a new gulp task named **clean**.

```
gulp.task('clean', function() {
   console.log("Running clean task");
   return gulp.src('dist/', {read: false})
      .pipe(clean());
});
```

f) Save your changes to **gulpfile.js**.

g) Test out the new **clean** task by executing the following command from the Integrated Terminal.

```
gulp clean
```

h) You should be able to verify that the **dist** folder has been deleted from your project's root folder.

```
PS C:\Student\Modules\02_NodeJS\Lab\project1> gulp clean
[21:12:35] Using gulpfile C:\Student\Modules\02_NodeJS\Lab\project1\gulpfile.js
[21:12:35] Starting 'clean'...
Running clean task
[21:12:35] Finished 'clean' after 16 ms
```

5. Install the gulp packages required to compile Typescript code into JavaScript and to generate source map files for debugging.

a) Return to the console in the Integrated Terminal and run the following command to install the **gulp-typescript** package.

```
npm install gulp-typescript --save-dev
```

b) Wait until the **npm install** command completes.

c) Install the **gulp-sourcemaps** package.

```
npm install gulp-sourcemaps --save-dev
```

d) Wait for the **npm install** command to complete.

e) At the top of **gulpfile.js**, add to more lines to create variables named **ts** and **tsProject** using the following code.

```
var gulp = require('gulp');
var clean = require('gulp-clean');
var ts = require("gulp-typescript");
var tsProject = ts.createProject("tsconfig.json");
var sourcemaps = require('gulp-sourcemaps');
```

f) At the bottom of **gulpfile.js** under the **clean** task, add a new task named **build**.

```
gulp.task('build', function() {

});
```

g) In the call to **gulp.task**, add **['clean']** as a parameter so that the **build** task is created with a dependency on the **clean** task.

```
gulp.task('build', ['clean'], function() {

});
```

h) Add the following line to the top of the **build** task to log to the console whenever this task is executed.

```
gulp.task('build', ['clean'], function() {
   console.log("Running build task");

});
```

i) Underneath the call to **console.log** add the following code to copy any html files in the **src** folder into the **dist** folder.

```
console.log("Running build task");

gulp.src('src/**/*.html').pipe(gulp.dest('dist'));
```

j) Next, add the following code to copy any CSS files in the **src/css** folder into the **dist/css** folder.

```
gulp.src('src/css/**/*.css').pipe(gulp.dest('dist/css'));
```

k) Next, add the following code to copy any PNG files in the **src/css/img** folder into the **dist/css/img** folder.

```
gulp.src('src/css/img/**/*.png').pipe(gulp.dest('dist/css/img'));
```

l) Finally, add the following code to run the TypeScript compiler to generate the **app.js** and **app.js.map** folder in the **dist** folder.

```
return tsProject.src()
   .pipe(sourcemaps.init())
   .pipe(tsProject())
   .pipe(sourcemaps.write('.', { sourceRoot: './', includeContent: false }))
   .pipe(gulp.dest("."));
```

m) At this point, the **build** task you have written should match the following code listing.

```
gulp.task('build', ['clean'], function () {
  console.log("Running build task");

  gulp.src('src/**/*.html').pipe(gulp.dest('dist'));
  gulp.src('src/css/**/*.css').pipe(gulp.dest('dist/css'));
  gulp.src('src/css/img/**/*.png').pipe(gulp.dest('dist/css/img'));

  return tsProject.src()
    .pipe(sourcemaps.init())
    .pipe(tsProject())
    .pipe(sourcemaps.write('.', { sourceRoot: './', includeContent: false }))
    .pipe(gulp.dest("."));

});
```

n) Save your changes to **gulpfile.js**.

o) Test out the new **build** task by executing the following command from the Integrated Terminal.

```
gulp build
```

p) You should be able to verify that the **dist** folder has been built out with all the files needed for distribution.

```
∨ PROJECT1
  > .vscode
  ∨ dist
    ∨ css
      ∨ img
        🖼 AppIcon.png
      # app.css
    ∨ scripts
    JS app.js
    JS app.js.map
    <> index.html
```
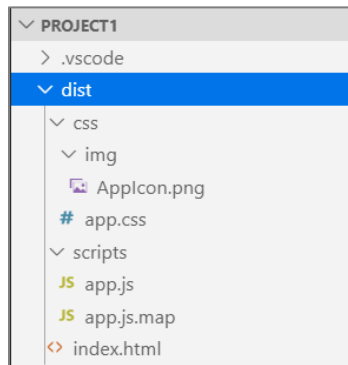
Once again, it's time to test out your application so you can verify that the **build** task has correctly generated all the files that are required in the **dist** folder of the project. However, you will not start as you did in previous steps using the **browser-sync** command from the command line. Instead, you will create a new gulp task named **start** and you will implement this task to start up the web server provided by browser-sync in an automated fashion.

6. Create a new gulp task named **start** to start up a browser-sync web server session and launch the browser.

a) Add a line at the top of **gulpfile.js** to create a variable named **browserSync** and initialize it using **require('browser-sync')**.

```
var gulp = require('gulp');
var clean = require('gulp-clean');
var ts = require("gulp-typescript");
var tsProject = ts.createProject("tsconfig.json");
var sourcemaps = require('gulp-sourcemaps');
var browserSync = require('browser-sync');
```

Note that you are not required to install a new npm package. That's because all the required gulp integration support was added when you installed the original package named **browser-sync**. You can say that **browser-sync** package is a gulp-friendly utility.

b) At the bottom of **gulpfile.js**, create a new gulp task named **start** which has a dependency on the **build** task. Also add a call to **console.log** to log to the console whenever this task is executed.

```
gulp.task('start', ['build'], function() {
  console.log("Running start  task");

});
```

c) Implement the **start** task with a call to **browserSync.init** as shown in the following code listing.

```
gulp.task('start', ['build'], function() {
  console.log("Running start  task");
  return browserSync.init( {server: {baseDir: 'dist'} } );
});
```

d) Save your changes to **gulpfile.js**.

e) Test out the new **start** task by executing the following command from the Integrated Terminal.

```
gulp start
```

f) You should be able to verify that running this task starts the web server and launches the browse to load the application.

7. Create a new gulp task named **refresh** and implement this task to refresh the browser during a debugging session.

a) At the bottom of **gulpfile.js**, create a new task named **refresh** with a dependency on the **build** task. Implement this task using the following code.

```
gulp.task('refresh', ['build'], function(){
  console.log("Running refresh task");
  return browserSync.reload();
});
```

8. Create a new gulp task named **serve** to start a debugging session with a file watch and automatic refreshing capabilities.

a) At the bottom of **gulpfile.js**, create a new task named **serve** with a dependency on the **start** task. Implement this task using the following code.

```
gulp.task('serve', ['start'], function() {
  console.log("Running serve task");
  gulp.watch("src/**/*", ['refresh']);

});
```

b) Save your changes to **gulpfile.js**.

c) Test out the new **serve** task by executing the following command from the Integrated Terminal.

```
gulp serve
```

d) The application should launch in the browser and allow you to test it as you did before. However, now you should be able to make updates to source files such as **index.html**, **app.css** and **app.ts** and when you save your changes, the project should automatically run the build task and then refresh the browser.
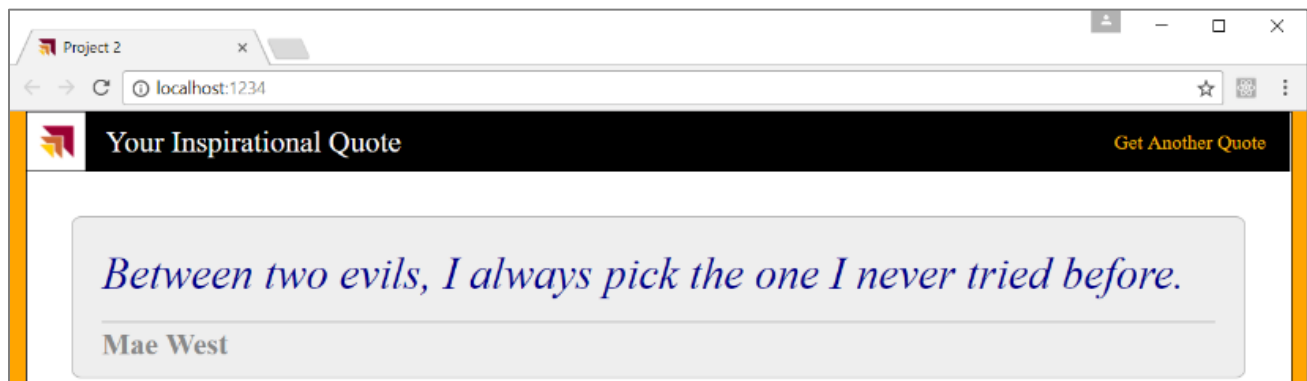
You have just spent the last three exercises working with **project1** and learning how to use utilities such as **npm**, **tsc** and **gulp**. In the next exercise, you will move on to begin working a new project named **project2** so you can begin learning how to use the **webpack** utility. As you will see, working with **webpack** changes fundamentally changes the way that you structure and test your project.

Exercise 4: Using WebPack to Bundle Your Project Files for Distribution

In this exercise, you will begin your work by copying a simple folder that contains an HTML files, a CSS file and three TypeScript files. You will then use **npm** to initialize the folder as a Node.js project and then you will learn to use **webpack** to compile the project's TypeScript files into single bundle for distribution. Along the way, you will also learn to install the **webpack-dev-server** package and to configure its developer-oriented features which make it possible to test and debug your project's code using the browser.

1. Inspect the project starter files in the folder named **project2**.

    a) Using Windows Explorer, open the folder at **c:\Student\Modules\02_NodeJS\Lab\StarterFiles\project2**.

    b) Make a copy of the **project2** folder outside the **StarterFiles** project at the following path.

    ```
    c:\Student\Modules\02_NodeJS\Lab\project2
    ```

    c) Using Windows Explorer, look inside the **src** folder and you will find an HTML file named **index.html** and an icon file named **favicon.ico**. The **src** folder also contains a child folder named **css** which contains a CSS file named **app.css**. The **css** folder also contains child folder named **img** that contains an image file named **AppIcon.png**.
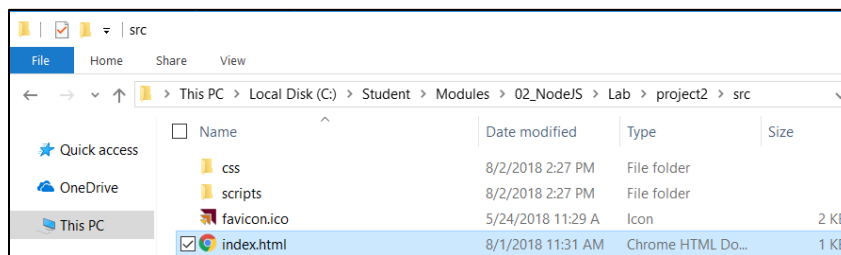
    d) Look in the **scripts** folder and you should see three TypeScript source files named **app.ts**, **quote.ts** and **quote-manager.ts**.
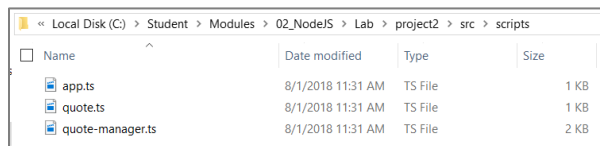
Note that **project2** uses the exact same TypeScript code that you worked with in **project1**. The only difference is that the TypeScript code in **project1** lived inside a single TypeScript file named **app.ts**. The new project named **project2** contains the exact same code except that it's spread across three TypeScript source files written to run inside an environment that supports dynamic module loading.

2. Open the **project2** folder with Visual Studio Code.

    a) Launch Visual Studio Code.

    b) Use the **File > Open Folder** command to open the project at the following path.

    ```
    c:\Student\Modules\02_NodeJS\Lab\project2
    ```

    c) Examine the structure of the folder and files within the project.
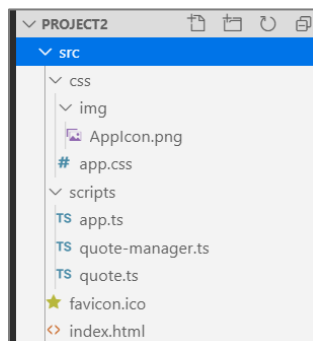
3. Take a moment to review the code inside the three TypeScript source files.

    a) Open **quote.ts** and examine the code inside. This file contains a single TypeScript class named **Quote**.

    ```typescript
    export class Quote {
        value: string;
        author: string;
        constructor(value: string, author: string){
            this.value = value;
            this.author = author;
        }
    }
    ```

    b) Open **quote-manager.ts** and examine the code inside. You can see this file contains an **import** statement to load the **Quote** class defined inside **quote.ts** in addition to defining a class named **QuoteManager**.

    ```typescript
    import { Quote } from './quote';

    export class QuoteManager {

      private  static quotes: Quote[] = [
        new Quote("Always borrow money from a pessimist. He won't expect it back.", "Oscar Wilde" ),
        new Quote("Behind every great man is a woman rolling her eyes.", "Jim Carrey" )
        // other quotes removed for brevity
      ];

      public static getQuote = () : Quote => {
          var index = Math.floor(Math.random()*QuoteManager.quotes.length);
          return QuoteManager.quotes[index];
      }

    }
    ```

    c) Open **app.ts** and examine the code inside. You should see an **import** statement to load the jQuery library and then two other **import** statements to load the **Quote** class and the **QuoteManager** class.

    ```typescript
    import * as $ from "jquery"

    import { Quote } from './quote';
    import { QuoteManager } from './quote-manager';

    $( () => {

      var displayNewQuote = (): void => {
        var quote: Quote = QuoteManager.getQuote();
        $("#quote").text(quote.value);
        $("#author").text(quote.author);
      };

      $("#new-quote").click( ()=> {
        displayNewQuote();
      });

      displayNewQuote();

    });
    ```

    d) Close **quote.ts**, **quote-manager.ts** and **app.ts** without saving any changes.

4. Initialize the **project2** folder as a Node.js project by executing the **npm init** command.

    a) Use the **View > Integrated Terminal** menu command to display the Integrated Terminal.

    b) From the console of the Integrated Terminal, type the following command and then press **Enter** to execute it.

    ```
    npm init –y
    ```

    c) After the command completes, you should see that a new file has been added to your project named **package.json**.

    d) Open the **package.json** and see what's inside.

    e) When you are done looking at the **package.json** file, leave this file open.

5. Install the initial set of Node.js packages your project needs to get started with **webpack**.

   a) Type and execute the following **npm** command to install the packages for **webpack** and the **webpack-cli**.

   ```
   npm install webpack webpack-cli --save-dev
   ```

   b) Type and execute the following **npm** command to install three webpack plugins that you will use in your project.

   ```
   npm install clean-webpack-plugin copy-webpack-plugin html-webpack-plugin --save-dev
   ```

   c) Type and execute the following **npm** command to install three webpack loaders that you will use in your project.

   ```
   npm install style-loader css-loader url-loader --save-dev
   ```

   d) Type and execute the following **npm** command to install the **typescript** package and **awesome-typescript-loader** package.

   ```
   npm install typescript awesome-typescript-loader --save-dev
   ```

   e) Type and execute the following **npm** command to install the jQuery library along with its typed definition files.

   ```
   npm install jquery @types/jquery --save-dev
   ```

   Now that you have installed the packages you need, it's time to call the **npm shrinkwrap** command to track in the package versions

6. Execute the **npm shrinkwrap** command to convert the **package.lock.json** file into the **npm.shrinkwrap.json** file.

   a) From the command line, type the following **npm shrinkwrap** command and then execute it by pressing the **Enter** key.

   ```
   npm shrinkwrap
   ```

   b) You should see that executing this command replaces the **package.lock.json** file with the **npm.shrinkwrap.json** file

7. Inspect **package.json** file to see what packages have been installed for the current project.

   a) When you examine **package.json**, you should see the **devDependencies** section includes the following dependencies.

   ```
   "devDependencies": {
     "@types/jquery": "^3.3.31",
     "awesome-typescript-loader": "^5.2.1",
     "clean-webpack-plugin": "^3.0.0",
     "copy-webpack-plugin": "^5.0.5",
     "css-loader": "^3.2.1",
     "html-webpack-plugin": "^3.2.0",
     "jquery": "^3.4.1",
     "style-loader": "^1.0.1",
     "typescript": "^3.7.2",
     "url-loader": "^3.0.0",
     "webpack": "^4.41.2",
     "webpack-cli": "^3.3.10"
   }
   ```

   Note that the package version number in your project may be more recent than those version numbers in the code listing above.

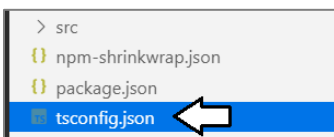8. Create and configure a **tsconfig.json** file for your project.

   a) Return to the console of the Integrated Terminal.

   b) Type and execute the following **npx tsc** command to generate a new **tsconfig.json** file at the root of your project.

   ```
   npx tsc --init
   ```

   c) After running this command, you should see that a new **tsconfig.json** file has been created at the root of your project.

d) Replace the contents of **tsconfig.json** with the following code.

```
{
  "compilerOptions": {
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "module": "commonjs",
    "sourceMap": true,
    "target": "es5",
    "lib": [ "dom", "es6" ]
  },
  "exclude": [
    "node_modules"
  ]
}
```

9. Create a new file named **webpack.config.js** to configure the **webpack** build process.

a) Create a new file named **webpack.config.js** at the root of the project.
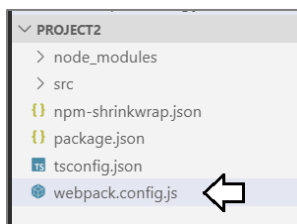
b) Copy and paste the following code to provide a starting point for the **webpack.config.js** file.

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

const HtmlWebpackPlugin = require('html-webpack-plugin');
const CopyWebpackPlugin = require('copy-webpack-plugin');
const { CleanWebpackPlugin } = require('clean-webpack-plugin');

module.exports = {}
```

c) Add the following code into the **module.exports** object the provide a starting point to initialize the object.

```
module.exports = {
  entry: './src/scripts/app.ts',
  output: {},
  resolve: {},
  plugins: [],
  module: {},
  mode: "development",
  devtool: 'source-map'
};
```

d) Replace the **output** property with the following code.

```
output: {
  filename: 'scripts/bundle.js',
  path: path.resolve(__dirname, 'dist'),
},
```

The **output** property determines the name and path of the JavaScript file that will be generated from all the project's TypeScript code. By the end of this exercise, you will also see how webpack is also able to add CSS styles and image files serialized into base64 encoded strings into the **bundle.js** file for distribution.

e) Replace the **resolve** property with the following code.

```
resolve: { extensions: ['.js', '.ts'] },
```

The **resolve** property determines which types of file will be examined during the webpack build process.

f) Replace the **plugins** property with the following code.

```
plugins: [
  new CleanWebpackPlugin({ cleanOnceBeforeBuildPatterns: './dist/*' }),
  new HtmlWebpackPlugin({ template: path.join(__dirname, 'src', 'index.html') }),
  new CopyWebpackPlugin([{ from: './src/favicon.ico', to: 'favicon.ico' }])
],
```

The **plugins** property determines what plugins will be loaded and how they will be initialized during the webpack build process. The **CleanWebpackPlugin** is used to delete any existing content inside the **dist** folder. The **HtmlWebpackPlugin** is used to copy **index.html** into the **dist** folder. Note that the **HtmlWebpackPlugin** also adds a script link into **index.jhtml** to load the main output file named **bundle.js**. The **CopyWebpackPlugin** is used to copy the favicon file named **favicon.ico** into the root of the **dist** folder.

g) Replace the **module** property with the following code.

```
module: {
  rules: [
    { test: /\.(ts)$/, loader: 'awesome-typescript-loader' },
    { test: /\.css$/, use: ['style-loader', 'css-loader'] },
    { test: /\.(png|jpg|gif)$/, use: [{ loader: 'url-loader', options: { limit: 8192 } }] }
  ],
},
```

The **module** property determines which types of files will be processed by specific modules during the webpack build process. Note that you can configure the **loader** property as in the case of the first module which loads the **awesome-typescript-loader**. The next two lines load modules using the **use** property instead of the **loader** property. You can use whichever style you like best.

h) At this point, the content of the **webpack.config.js** file should match the following code listing.

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

const HtmlWebpackPlugin = require('html-webpack-plugin');
const CopyWebpackPlugin = require('copy-webpack-plugin');
const { CleanWebpackPlugin } = require('clean-webpack-plugin');

module.exports = {
  entry: './src/scripts/app.ts',
  output: {
    filename: 'scripts/bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
  resolve: {
    extensions: ['.js', '.ts']
  },
  plugins: [
    new CleanWebpackPlugin({ cleanOnceBeforeBuildPatterns: './dist/*' }),
    new HtmlWebpackPlugin({ template: path.join(__dirname, 'src', 'index.html') }),
    new CopyWebpackPlugin([{ from: './src/favicon.ico', to: 'favicon.ico' }])
  ],
  module: {
    rules: [
      { test: /\.(ts)$/, loader: 'awesome-typescript-loader' },
      { test: /\.css$/, use: ['style-loader', 'css-loader'] },
      { test: /\.(png|jpg|gif)$/, use: [{ loader: 'url-loader', options: { limit: 8192 } }] }
    ],
  },
  mode: "development",
  devtool: 'source-map'
};
```

i) Save your changes to **webpack.config.js** and leave this file open.

This CSS import capabilities of webpack provide a valuable strategy for reducing the number of files that need to be distributed along with your applications. Because you have configured **project2** with the **style-loader** module, you can now import CSS files directly into a TypeScript file such as **app.ts**. This is what you will do in the next step.

10. Import the CSS styles from **app.css** into **app.ts**.

    a) Open **app.ts** from the **src/scripts** folder and add the following import statement to import the CSS styles from **app.css**.

```
import * as $ from "jquery"

import { Quote } from './quote';
import { QuoteManager } from './quote-manager';

import './../css/app.css'
```

    b) Save and close **app.ts**.

11. Edit **index.html** to remove the stylesheet link to **app.css**.

    a) Open the **index.html** file from inside the **src** folder and remove the link to **app.css**.

```
src > <> index.html > ⊘ html > ⊘ head > ⊘ link
  1    <!DOCTYPE html>
  2    <html>
  3
  4    <head>
  5      <title>Project 2</title>
  6      <meta charset="utf-8" />
  7      <link href="css/app.css" rel="stylesheet" />
  8    </head>
```

    b) Save your changes and close **index.html**.

At this point, you no longer need to distribute any CSS files with this application because the styles defined inside **src/css/app.css** will now be bundled into **app.js** for distribution. Also note that image file named **AppIcon.png** is automatically serialized by the **url-loader** and added to the CSS styles in **bundle.js** so there is no need to distribute that image file with the application.

12. Use **webpack** to build your project's TypeScript files into a single JavaScript file named **bundle.js**.

    a) Return to the file named **package.json**.

    b) Remove the existing command from the **scripts** section and replace it with the build command as shown in the following code.

```
"scripts": {
    "build": "webpack"
}
```

    c) Save your changes to **package.json**.

    d) Return to the console in the Integrated Terminal.

    e) Type and execute the following **npm** command run the **build** command.

```
npm run build
```

    f) You should see output in the console as **webpack** compiles your TypeScript source files into a single file named **bundle.js**.

```
PS C:\Student\Modules\02_NodeJS\Lab\project2> npm run build

> project2@1.0.0 build C:\Student\Modules\02_NodeJS\Lab\project2
> webpack

i ｜atl｜: Using typescript@3.7.2 from typescript
i ｜atl｜: Using tsconfig.json from C:/Student/Modules/02_NodeJS/Lab/project2/tsconfig.json
i ｜atl｜: Checking started in a separate process...
i ｜atl｜: Time: 668ms
Hash: 595a07e5d9d322477c27
Version: webpack 4.41.2
Time: 2913ms
Built at: 12/03/2019 10:25:28 PM
          Asset       Size  Chunks            Chunk Names
     favicon.ico   1.12 KiB          [emitted]
      index.html  702 bytes          [emitted]
 scripts/bundle.js   298 KiB    main  [emitted]       main
 scripts/bundle.js.map   384 KiB    main  [emitted] [dev]  main
Entrypoint main = scripts/bundle.js scripts/bundle.js.map
[./node_modules/css-loader/dist/cjs.js!./src/css/app.css] 2.13 KiB {main} [built]
[./src/css/app.css] 407 bytes {main} [built]
[./src/css/img/AppIcon.png] 979 bytes {main} [built]
[./src/scripts/app.ts] 505 bytes {main} [built]
[./src/scripts/quote-manager.ts] 2.38 KiB {main} [built]
[./src/scripts/quote.ts] 275 bytes {main} [built]
    + 4 hidden modules
Child html-webpack-plugin for "index.html":
    1 asset
    Entrypoint undefined = index.html
    [./node_modules/html-webpack-plugin/lib/loader.js!./src/index.html] 902 bytes {0} [built]
    [./node_modules/webpack/buildin/global.js] (webpack)/buildin/global.js 472 bytes {0} [built]
    [./node_modules/webpack/buildin/module.js] (webpack)/buildin/module.js 497 bytes {0} [built]
        + 1 hidden module
PS C:\Student\Modules\02_NodeJS\Lab\project2>
```

g) You should be able to verify that **webpack** has created three new files inside the **dist** folder. The two files named **favicon.ico** and **index.html** have been added to the root of the **dist** folder while **bundle.js** has been added the child **scripts** folder.
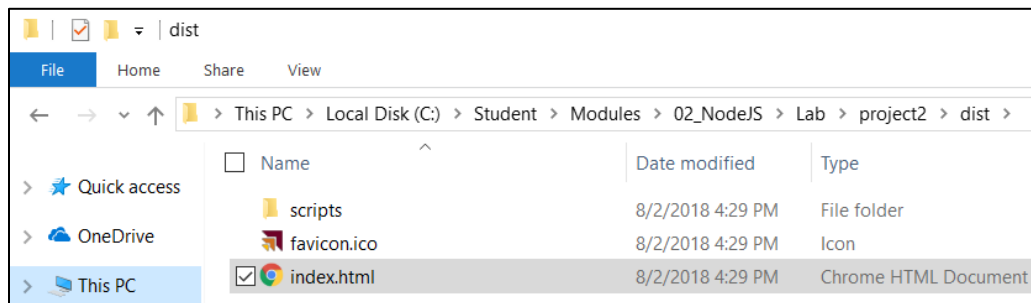
> ∨ PROJECT2
> ∨ dist
>   ∨ scripts
>   JS bundle.js
>   JS bundle.js.map
>   ★ favicon.ico
>   <> index.html
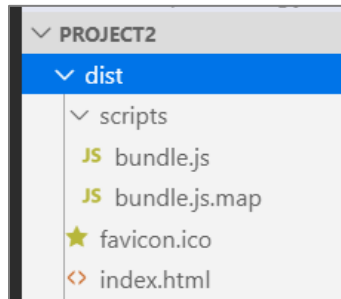
h) Open **bundle,js** and inspect the code inside.

While the **bundle.js** file may be hard to read, this file contains the JavaScript code for the jQuery library in addition to the JavaScript code generator from the project's three TypeScript source files. The file also contains your project's CSS styles and the image from **AppIcon.png** as a serialized base64 encoded string.
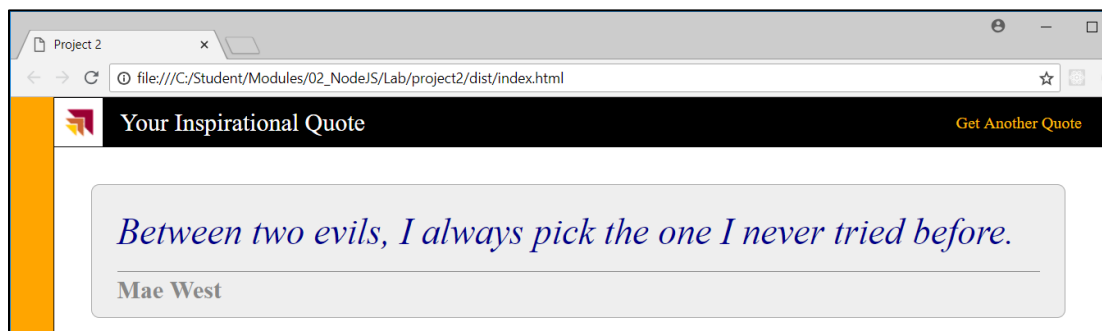
i) Close **bundle.js** without saving any changes.

13. Try running the application by launching **index.html** directly from the Windows file system.

a) Using Windows Explorer, navigate to the folder at **C:\Student\Modules\02_NodeJS\Lab\project2\dist**.

b) Double click on **index.html** to launch the application in the browser.

		dist

File   Home   Share   View

← → ∨ ↑ 📁 > This PC > Local Disk (C:) > Student > Modules > 02_NodeJS > Lab > project2 > dist >

	Name	Date modified	Type
⭐ Quick access			
☁ OneDrive	📁 scripts	8/2/2018 4:29 PM	File folder
	🔲 favicon.ico	8/2/2018 4:29 PM	Icon
💻 This PC	☑ 🌐 index.html	8/2/2018 4:29 PM	Chrome HTML Document

c) The application should continue to run just like it has before.

Project 2   ✕

← → C   ⓘ file:///C:/Student/Modules/02_NodeJS/Lab/project2/dist/index.html   ☆

🔺 **Your Inspirational Quote**   Get Another Quote

Between two evils, I always pick the one I never tried before.

Mae West

The application runs just as before, but now it only requires three distribution files. That's because webpack has been able to add the contents of **app.css** into **bundle.js**. Even the **AppIcon.png** file from inside **src/css/img** folder has been serialized as a base64 encoded string by the **url-loader** module and added to **bundle.js**. The only reason you cannot do the same with the **favicon.ico** file is that the browser needs to see a physical file to set the favicon for the current web page.

At this point, you are done configuring the webpack build process. Now it's time to install the **webpack-dev-server** package so you can test and debug the code in your project.

14. Install and configure the **webpack-dev-server** package to add debugging support to your project.

    a) Run the following command from the Integrated Terminal to install the **webpack-dev-server** package version 3.2.

```
npm install webpack-dev-server --save-dev
```

15. Add a new script command to **package.json** to start a debugging session.

    a) Return to the code editor window for **package.json**.

    b) Update the **scripts** section by adding the following **start** command.

```
"scripts": {
  "build": "webpack",
  "start": "webpack-dev-server --open"
}
```

    c) Save your changes to **package.json**.

16. Start and test your project using the debugging support of the **webpack-dev-server** package.

    a) Return to the console of the Integrated Terminal.

    b) Run the following command to start the **webpack-dev-server** package web server and launch the application in the browser.

```
npm run start
```

    c) The application should launch in the browser using the **localhost** and the default port of the webpack dev server which is **8080**. The application should continue to run just as before.

17. Test out the file watching behavior of the webpack dev server.

    a) While the application is running, try to update source files inside the **src** folder such as **index.html** or **app.css**.

    b) When you save your changes, the webpack file watch support will automatically rebuild the project and refresh the browser.

Congratulations. You have now learned the basics skills of working in a Node.js environment with developer utilities such as **npm**, **tsc**, **gulp** and **webpack**. Your experience and familiarity with these tools will be important as you begin to develop with libraries and frameworks that build on top of Node.js such as **SharePoint Framework**, **React** and **Angular2**.