Developing with Node.JS, NPM and Visual Studio Code

Lab Time: 60 minutes

**Lab Folder**: C:\Student\Modules\13_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.docx.. f you are using the GSA2016 student from Critical Path Training, these prerequisites should already be installed.

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 NodeJS 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\09_NodeJS\Lab\StarterFiles\project1**.

    b) Make a copy of the project1 folder outside the StarterFiles project **c:\Student\Modules\09_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**.
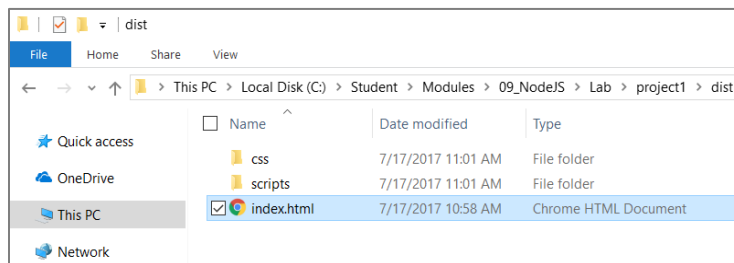

2. Open the **project1** folder with Visual Studio Code.

    a) Launch Visual Studio Code.

    b) Use the **File >> Open Folder** command to open the folder at **c:\Student\Modules\09_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.
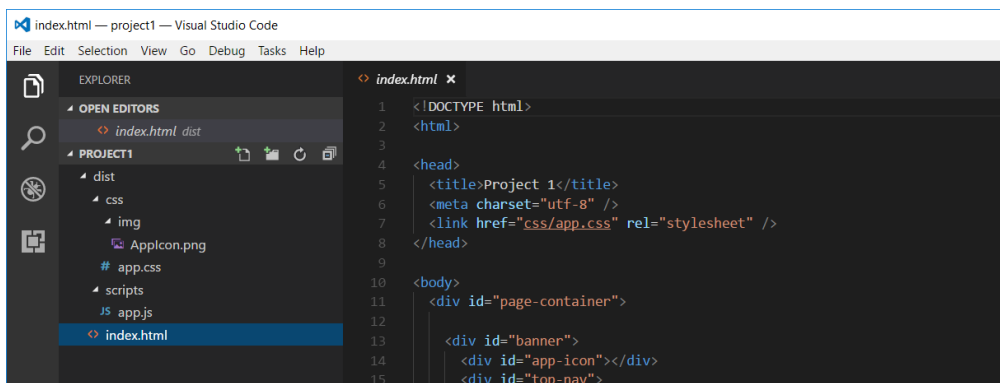

    d) Open the CCS file named **app.css** in the **Dist/css** folder and quickly review the CSS code that's inside.

    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.

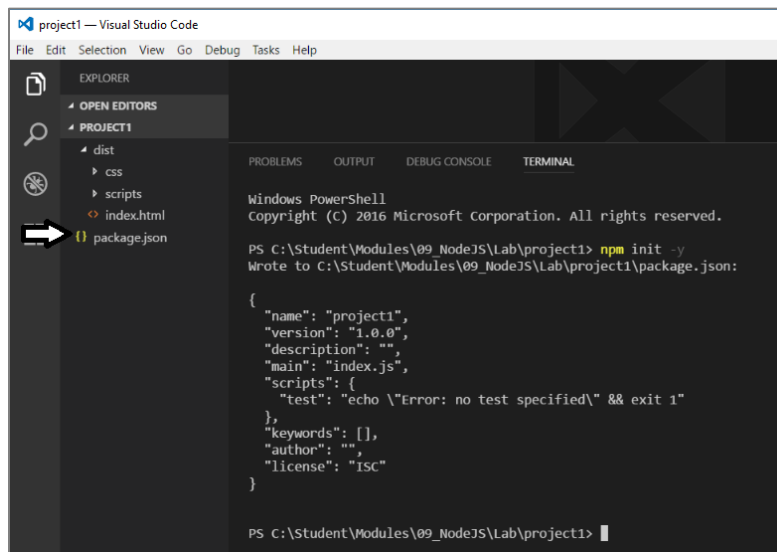   a)   Use the **View > Integrated Terminal** menu command to display the Integrated Terminal.
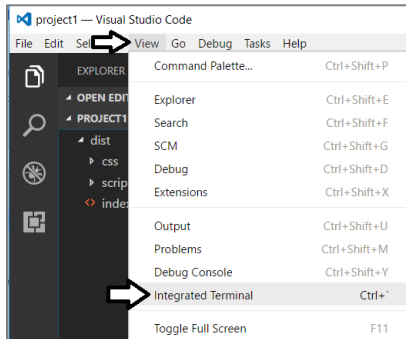
      b)   You should see command line interface of the Integrated Terminal as shown in the following screenshot.
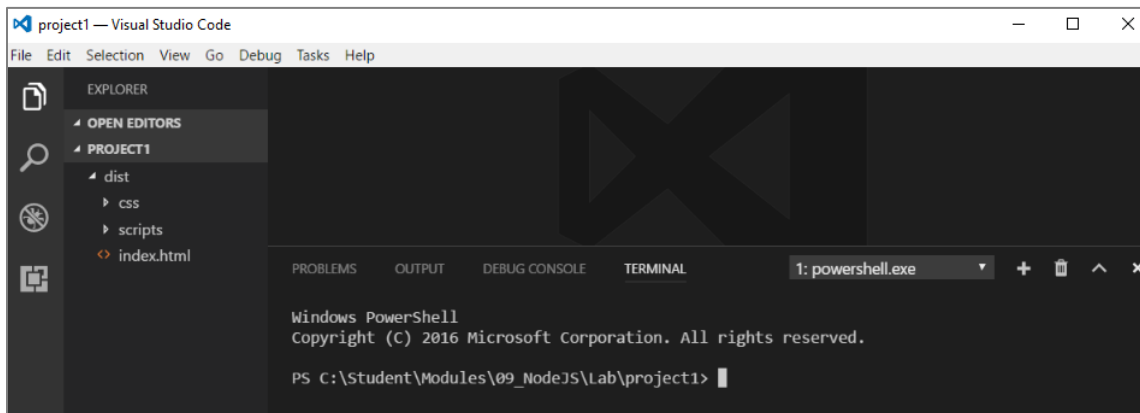
      c)   From the command line of the Integrated Terminal, type the following command and then press **Enter** to execute it.
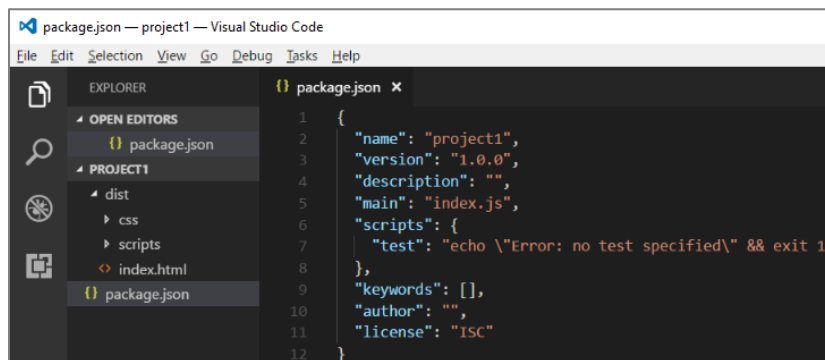
```
npm init -y
```

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

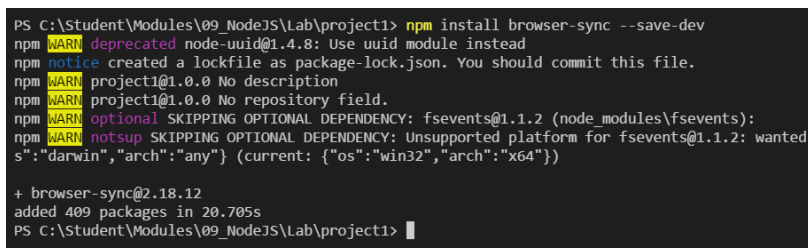e) Open the **package.json** and see what's inside. There is no need for you to make any changes to this file.

f) 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 Node.JS packaged named **browser-sync** to provide a 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.

What the Heck! There is a message in the console that says 409 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.

GSA2016: The Great SharePoint Adventure for Developers
Module 13 Lab: Developing with Node.JS, NPM and Visual Studio Code

Live Lab Version: Nov 6, 2017

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.

```
"devDependencies": {
    "browser-sync": "^2.18.12"
}
```

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 409 packages that were installed including the actual version number for each package.
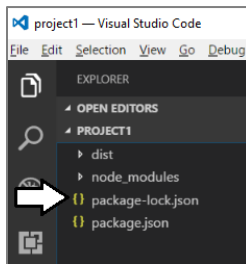
f) You should also notice that that new folder has been created named **node_modules**.
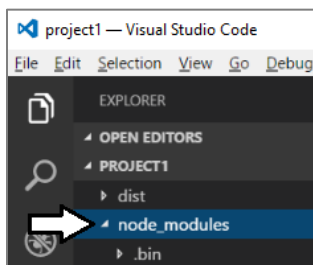
g) If you expand the **node_modules** folder, you will see many child folders which Node.JS packages that have been installed.
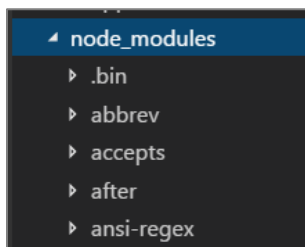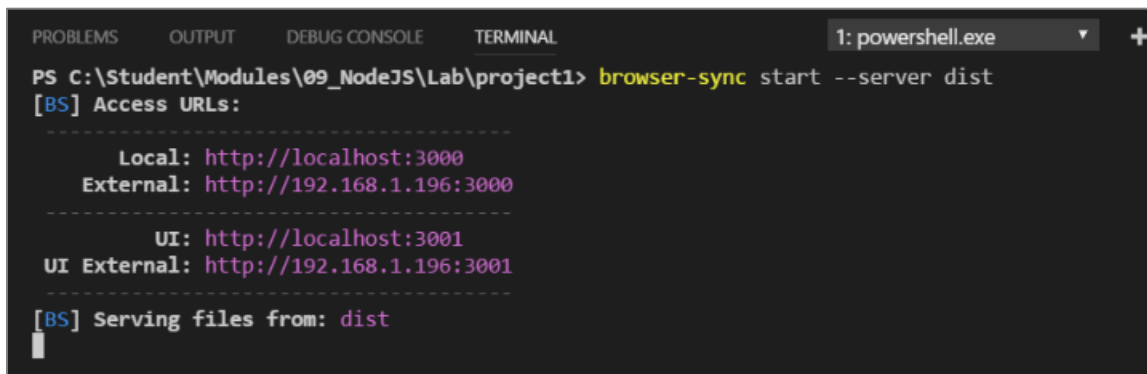
5. Use the web server support in browser-sync to run and test the application in the browser.

   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.

GSA2016: The Great SharePoint Adventure for Developers
Module 13 Lab: Developing with Node.JS, NPM and Visual Studio Code

Live Lab Version: Nov 6, 2017

d) Next, browser-sync support should launch the browser and display the simple web application defined by **index.html**.
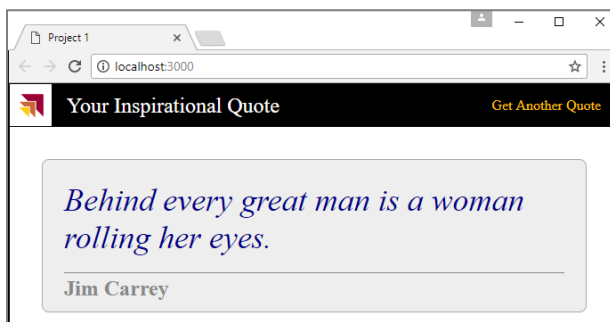
e) Test the application by clicking the **Get Another Quote** link.
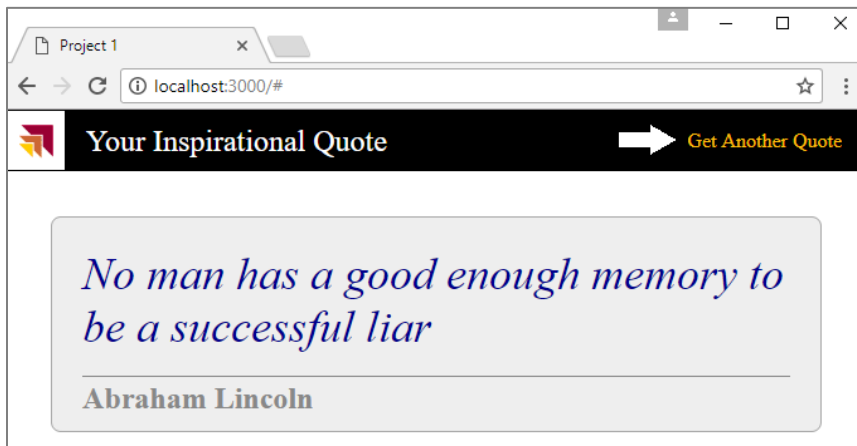
As you can see, this application is very simple. It is built using an HTML file, a CSS file and a JavaScript file. The only outside library that this project depends on is jQuery. Note that index.html has a script link to download jQuery from a public CDN on the Internet.

6. Stop the Node.JS web server and 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) The console window should respond by prompting you to terminate the current batch job.

e) 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** and then typing **y** and pressing **Enter**.

7. 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,

GSA2016: The Great SharePoint Adventure for Developers
Module 13 Lab: Developing with Node.JS, NPM and Visual Studio Code

Live Lab Version: Nov 6, 2017
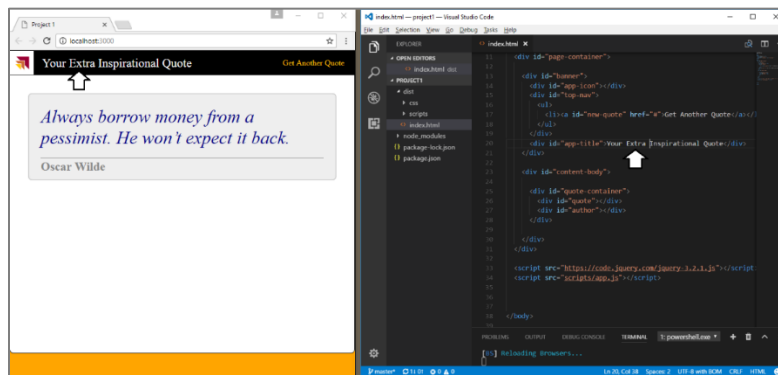
f) Return to the Integrated Terminal in Visual Studio Code and stop the web server debugging session.

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.
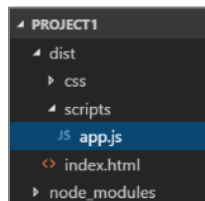
   b) Create a new top-level folder in the project named **src**.
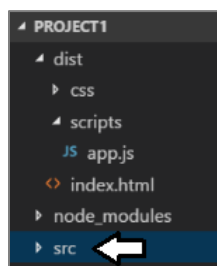
   c) Using a drag-and-drop operation with the mouse, move the **scripts** folder (and the **app.js** file inside) from the **dist** folder over to the **src** folder.
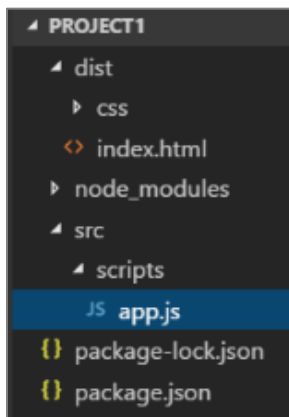
d) Change the file extension by renaming the file from **app.js** to **app.ts**.
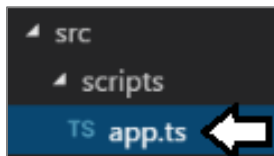
Now your project has a TypeScript source file. However, the project doesn't yet have any support yet for compiling this TypeScript source file into JavaScript code for testing and for distribution. In the next step, you will install the **typescript** package which adds the TypeScript compiler named **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 the following command and press **Enter** to display the version number of the installed version of the **typescript** package.

   ```
   tsc --version
   ```

   f) Verify that the command executed against the **tsc** command-line utility succeeded.

GSA2016: The Great SharePoint Adventure for Developers
Module 13 Lab: Developing with Node.JS, NPM and Visual Studio Code

Live Lab Version: Nov 6, 2017

```
PS C:\Student\Modules\09_NodeJS\Lab\project1> tsc --version
Version 2.4.1
PS C:\Student\Modules\09_NodeJS\Lab\project1>
```

> Now your project is configured to use the TypeScript compiler. However, you are not quite ready to compile the TypeScritp 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.
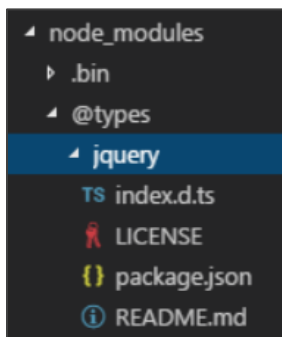
> Now you are ready to compile your TypeScript source file into JavaScript code using the TypeScript compiler. The first step will be to generate a **tsconfig.json** file which will allow you to configure the input and output of the TypeScript compiler.

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.

   ```
   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.
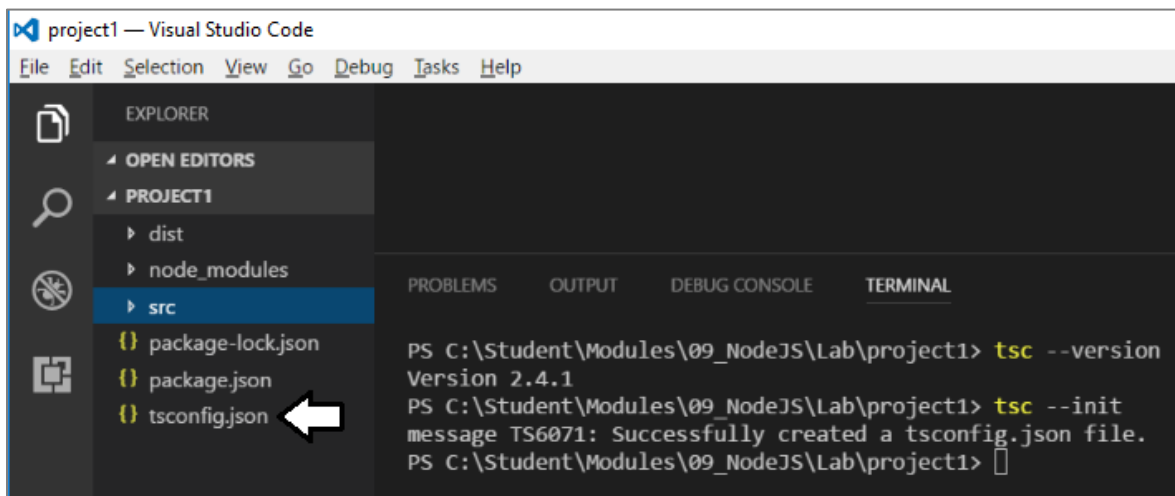
   c) Open the **tsconfig.json** file in an editor window.

GSA2016: The Great SharePoint Adventure for Developers
Module 13 Lab: Developing with Node.JS, NPM and Visual Studio Code

Live Lab Version: Nov 6, 2017

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 **tsc** without passing any parameters.

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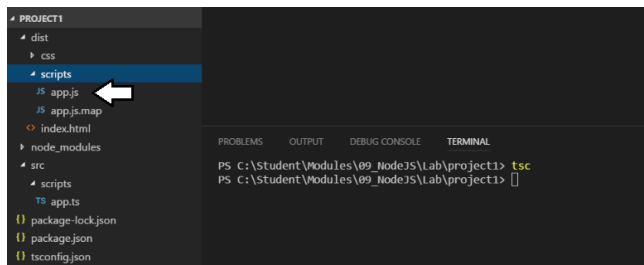
d) Open the **app.js** that has been generated by the TypeScript compiler so you can see what the generated looks like. You will find that your TypeScript code has been transpiled into JavaScript code that's EcmaScript5 compatible so that it runs in all today's popular browsers.

e) 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 works because the original JavaScript code in **app.js** is valid TypeScript code. However, the 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 {

  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.
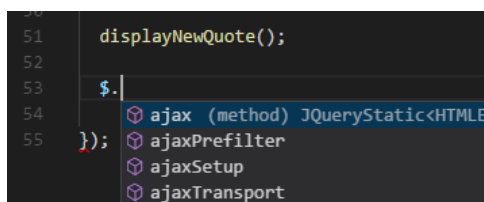
```
$( () => {

  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

GSA2016: The Great SharePoint Adventure for Developers
Module 13 Lab: Developing with Node.JS, NPM and Visual Studio Code

Live Lab Version: Nov 6, 2017

9. Compile the refactored TypeScript code.

    a) the tsc command from the Integrated Terminal.

```
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.
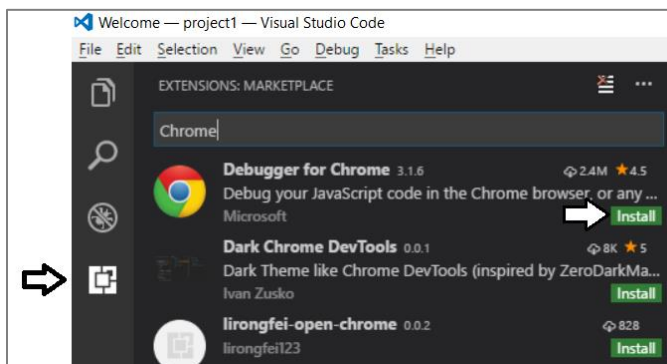
    e) Once it's installed, click the **Reload** button to restart the Visual Studio Code window.
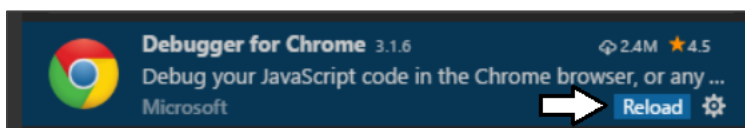
11. Configure support in Visual Studio Code for debugging client-side TypeScript code.

    a) Click on the **Debug** tab in the left navigation.

GSA2016: The Great SharePoint Adventure for Developers
Module 13 Lab: Developing with Node.JS, NPM and Visual Studio Code

Live Lab Version: Nov 6, 2017
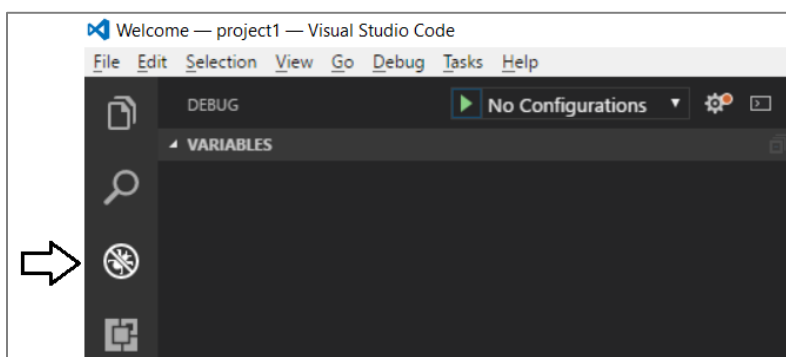
b) Drop down the **Configuration** menu and select **Add Configuration…**.
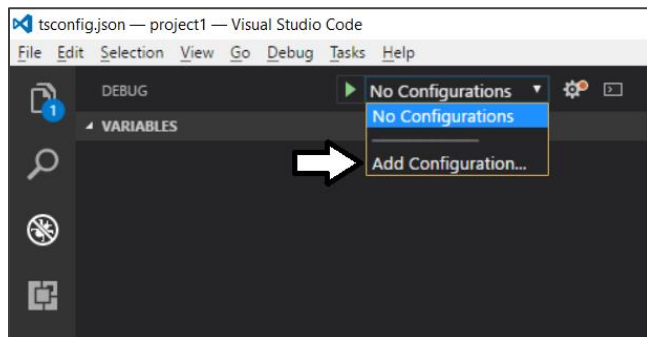
c) From the **Select Environment** menu, select **Chrome**.
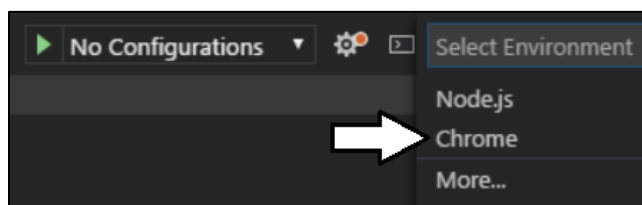
d) You will notice that a new file named **launch.json** has been added to the **.vscode** folder in project.
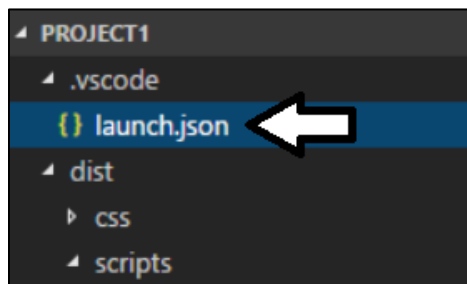
e) Replace with contents of launch.json with the following JSON code.

```json
{
    "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**.

GSA2016: The Great SharePoint Adventure for Developers
Module 13 Lab: Developing with Node.JS, NPM and Visual Studio Code

Live Lab Version: Nov 6, 2017

```
{} launch.json ●
1    {
2        "version": "0.2.0",
3        "configurations": [
4            {
5                "name": "Local Debugging Session",
6                "type": "chrome",
7                "request": "launch",
8                "url": "http://localhost:3000/",
9                "webRoot": "${workspaceRoot}/dist",
10               "sourceMaps": true,
11               "runtimeArgs": [
12                   "--remote-debugging-port=9222"
13               ]
14           }
15       ]
16   }
```

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.
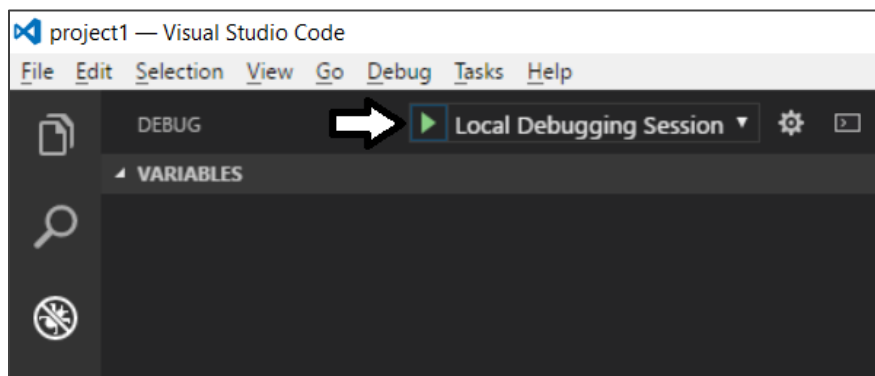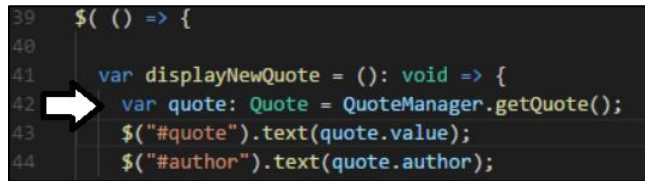
   e) The application should launch in the browser. Verify that is works like it did before.

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.

GSA2016: The Great SharePoint Adventure for Developers
Module 13 Lab: Developing with Node.JS, NPM and Visual Studio Code

Live Lab Version: Nov 6, 2017
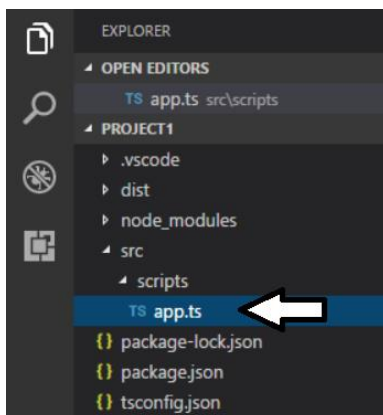
g) Select the first line of code in the **displayNewQuote** method and set a breakpoint by pressing the **{F9}** key.

h) If the debugging session has been created properly, you should see a red dot indicating an active breakpoint.

i) Click on the **Get Another Quote** link to test your breakpoint.
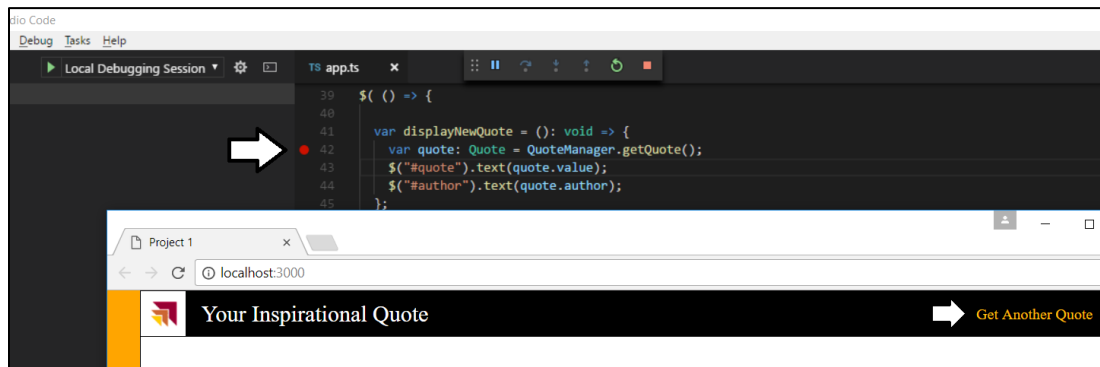
j) At this point, the Visual Studio Code Debugger should break at the line of code where you set the breakpoint.
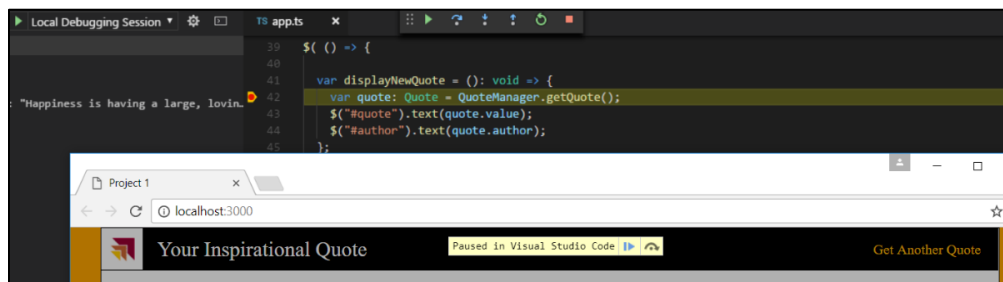
k) Click the Step Over button on the debugging toolbar to single step through your code.
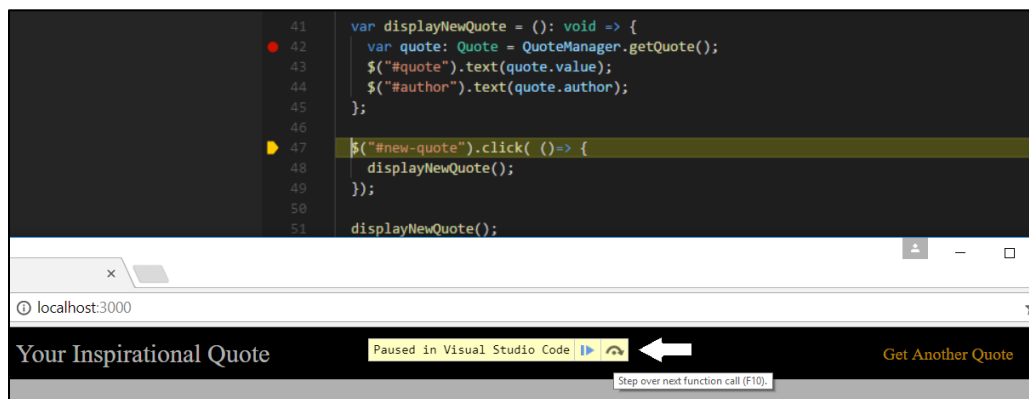
Once you have successfully gotten the Visual Studio Code debugger to single step through your code, you are done with this exercise.

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**.
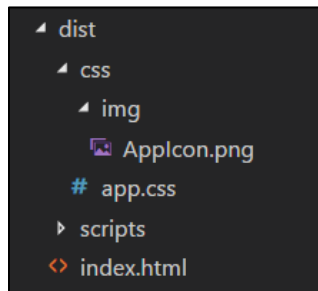
    b) Using a drag-and-drop operation with the mouse, move the **index.html** file from the **dist** folder to the **src** folder.

    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.
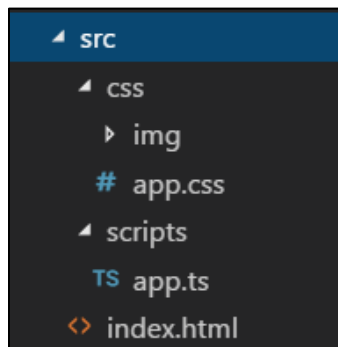
2. Install the **gulp** task runner utility suing **npm**.

    a) Return to the console in the Integrated Terminal,

    b) Run the following npm install command to install gulp.

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

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

Once you have installed gulp, you can directly call the **gulp** command-line utility from the console window.

GSA2016: The Great SharePoint Adventure for Developers
Module 13 Lab: Developing with Node.JS, NPM and Visual Studio Code

Live Lab Version: Nov 6, 2017

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

```
gulp
```

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

```
PS C:\Student\Modules\09_NodeJS\Lab\project1> gulp
[20:36:09] No gulpfile found
PS C:\Student\Modules\09_NodeJS\Lab\project1>
```

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**.

   ```
   ▲ PROJECT1
     ▶ dist
     ▶ node_modules
     ▶ src
     JS gulpfile.js  ⟵
     {} package-lock.json
     {} package.json
     {} tsconfig.json
   ```
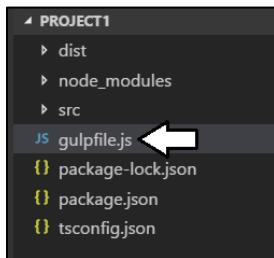
   b) Add the following JavaScript code into **gulpfile.js** to create a simple gulp task named **default**.

   ```javascript
   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\09_NodeJS\Lab\project1> gulp
   [22:16:56] Using gulpfile C:\Student\Modules\09_NodeJS\Lab\project1\gulpfile.js
   [22:16:56] Starting 'default'...
   Running my very first gulp task
   [22:16:56] Finished 'default' after 904 µs
   PS C:\Student\Modules\09_NodeJS\Lab\project1>
   ```

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

   ```javascript
   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\09_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@1.1.2 (node_modules\fsevents):
   npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.1.2: want
   64"})

   + gulp-clean@0.3.2
   added 115 packages and updated 1 package in 21.316s
   PS C:\Student\Modules\09_NodeJS\Lab\project1>
   ```

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.

5.   Create a new gulp task named **build** to generate the required files in the **dist** folder using source files inside the **src** folder.

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.

```
PS C:\Student\Modules\09_NodeJS\Lab\project1> npm install gulp-typescript --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@1.1.2 (node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.1.2: want
64"})

+ gulp-typescript@3.2.1
added 151 packages in 26.999s
PS C:\Student\Modules\09_NodeJS\Lab\project1> █
```

c)   At the top of gulpfile.json, 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");
```

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

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

});
```

e)   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() {

});
```

f)   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");


});
```

g) 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'));
```

h) 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'));
```

i) 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'));
```

j) 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(tsProject())
    .js.pipe(gulp.dest("."));
```

k) 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(tsProject())
        .js.pipe(gulp.dest("."));

});
```

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

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

```
gulp build
```

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

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 the browser-sync web server 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 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();
})
```

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

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.
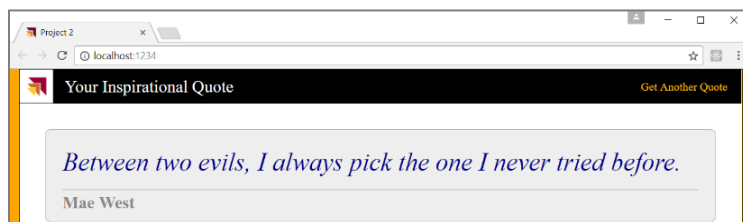
You have just spent the last three exercises working with **project1** and learning how to use utilities such as **npm**, **tsc** and **gulp**. In the next exercise, you will move on to begin working a new project named **project2** so you can begin learning how to use the **webpack** utility. As you will see, working with **webpack** changes fundamentally changes the way that you structure and test your project.

GSA2016: The Great SharePoint Adventure for Developers
Module 13 Lab: Developing with Node.JS, NPM and Visual Studio Code

Live Lab Version: Nov 6, 2017

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 in bundles 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\09_NodeJS\Lab\StarterFiles\project2**.

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

    ```
    c:\Student\Modules\09_NodeJS\Lab\project2
    ```

    c) Using Windows Explorer, look inside the **src** folder and you will find an HTML file named **index.html**. The **src** folder also contains a child folder named **css** folder 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**.
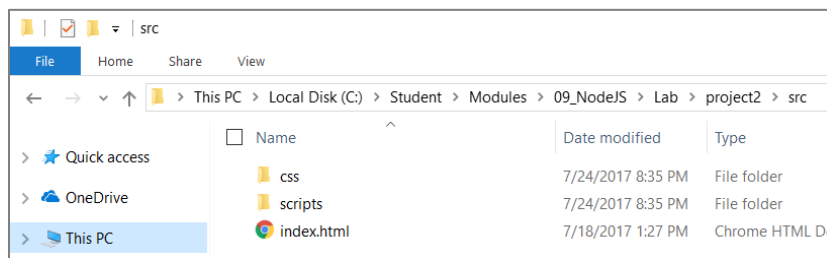
    d) Look in the **scripts** folder and you should see three TypeScript source files named **app.ts**, **quote.ts** and **quote-manager.ts**.
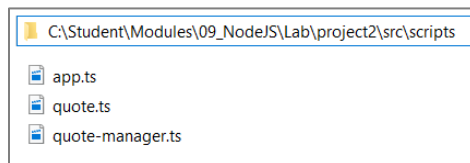
Note that **project2** uses the exact same TypeScript code that you worked with in **project1**. The only difference is that the TypeScript code in **project1** lived inside a single TypeScript file named **app.ts**. The difference is that **project2** contains the same code that is spread across three TypeScript source files that have been written to run inside an environment that supports 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\09_NodeJS\Lab\project2
    ```

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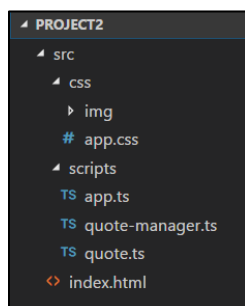
3. Take a moment to review the code inside the three TypeScript source files.

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

```
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**.

```
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.

```
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.

    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 **typescript** package.

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

    b) 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
    ```

    c) Type and execute the following **npm** command to install **webpack** and the TypeScript loader named **ts-loader**.

    ```
    npm install webpack ts-loader --save-dev
    ```

    d) Inspect **package.json** file and you should see the **devDependencies** section includes the following developer dependencies.

    ```
    "devDependencies": {
      "@types/jquery": "^3.2.10",
      "jquery": "^3.2.1",
      "ts-loader": "^2.3.2",
      "typescript": "^2.4.2",
      "webpack": "^3.4.1"
    }
    ```

6. 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 **tsc** command to generate a new **tsconfig.json** file at the root of your project.

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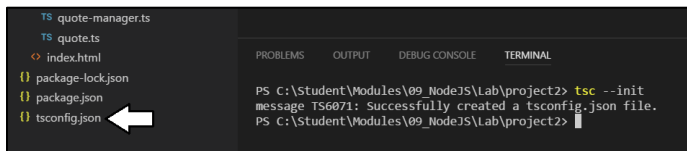
    d) Replace the contents of **tsconfig.json** with the following code.

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

7. 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.
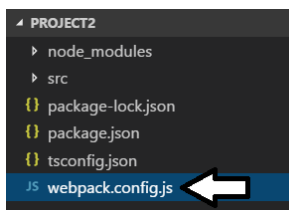
b) Copy and paste the following code into **webpack.config.js**.

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

module.exports = {
  entry: {
    'app': './src/scripts/app.ts'
  },
  output: {
    path: __dirname + '/dist',
    filename: './scripts/bundle.js'
  },
  resolve: {
    extensions: ['.ts', '.js']
  },
  module: {
    rules: [{
      test: /\.ts$/,
      use: 'ts-loader'
    }]
  }
};
```

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

8. 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.

```
{
  "name": "project2",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "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\09_NodeJS\Lab\project2> npm run build

> project2@1.0.0 build C:\Student\Modules\09_NodeJS\Lab\project2
> webpack

ts-loader: Using typescript@2.4.2 and C:\Student\Modules\09_NodeJS\Lab\project2\tsconfig.json
Hash: d8ae4e980bdaf7bbc65b
Version: webpack 3.4.1
Time: 1220ms
              Asset    Size  Chunks                    Chunk Names
./scripts/bundle.js  274 kB       0  [emitted]  [big]  app
   [0] ./src/scripts/app.ts 475 bytes {0} [built]
   [2] ./src/scripts/quote-manager.ts 2.42 kB {0} [built]
   [3] ./src/scripts/quote.ts 261 bytes {0} [built]
    + 1 hidden module
PS C:\Student\Modules\09_NodeJS\Lab\project2>
```

g) You should be able to verify that **webpack** has created a new file inside the **dist/scripts** folder named **bundle.js**.
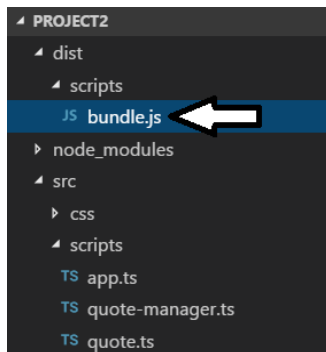
h) Open **bundle,js** and inspect the code inside. You should be able to see that it contains the JavaScript code for the jQuery library in addition to the JavaScript code generator from the project's three TypeScript source files.

i) Close **bundle.js** without saving any changes.

9. Split out the JavaScript code for the jQuery library into a separate bundle file named **vendor.ts**.

a) Add a new TypeScript source file named **vendor.ts** in the **src/scripts** folder.
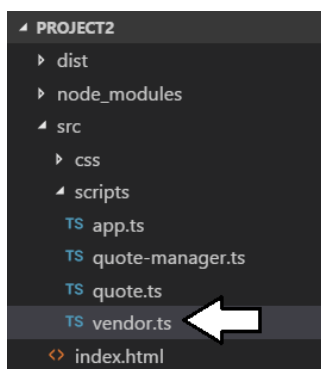
b) Add the following line of code to the top of **vendor.ts**.

```
import 'jquery/dist/jquery';
```

c) Save and close **vendor.ts**.

d) Navigate back to the editor window for **webpack.config.js**.

e) Beneath line that defines the **path** variable, add a new variable named **CommonsChunkPlugin** using the following code.

```
const path = require('path');
const CommonsChunkPlugin = require('webpack/lib/optimize/CommonsChunkPlugin');
```

CommonsChunkPlugin is built into the core webpack package so you don't need to install any additional packages to use it.

f) Move down in **webpack.config.js** to the **entry** section.

```
entry: {
  'app': './src/scripts/app.ts'
}
```

g) Add a second **entry** for the **vendor** bundle as shown in the following code listing.

```
entry: {
  'app': './src/scripts/app.ts',
  'vendor': './src/scripts/vendor.ts'
}
```

h) Move down in **webpack.config.js** and place your cursor after the **output** section but before the **resolve** section.

i) Add a new section named plugins using the following code listing.

```
plugins: [
  new CommonsChunkPlugin({
    name: 'vendor',
    filename: './scripts/vendor.bundle.js'
  })
]
```

j) At this point, your **webpack.config.js** file should match the following code listing.

```
const path = require('path');
const CommonsChunkPlugin = require('webpack/lib/optimize/CommonsChunkPlugin');

module.exports = {
  entry: {
    'app': './src/scripts/app.ts',
    'vendor': './src/scripts/vendor.ts'
  },
  output: {
    path: __dirname + '/dist',
    filename: './scripts/bundle.js'
  },
  plugins: [
    new CommonsChunkPlugin({
      name: 'vendor',
      filename: './scripts/vendor.bundle.js'
    })
  ],
  resolve: {
    extensions: ['.ts', '.js']
  },
  module: {
    rules: [{
      test: /\.ts$/,
      use: 'ts-loader'
    }]
  }
};
```

k) Save your changes to **webpack.config.js** and leave the file open.

10. Run the npm script command named **build** to rebuild the project using webpack.

a) Return to the console of the Integrated Terminal and execute the following **npm** command.

```
npm run build
```

b) As the **build** command executes, you should see output in the console displaying the details of the webpack build process.

```
PS C:\Student\Modules\09_NodeJS\Lab\project2> npm run build

> project2@1.0.0 build C:\Student\Modules\09_NodeJS\Lab\project2
> webpack

ts-loader: Using typescript@2.4.2 and C:\Student\Modules\09_NodeJS\Lab\project2\tsconfig.json
Hash: 2620a7c45f47a0cc754e
Version: webpack 2.7.0
Time: 1573ms
                    Asset      Size  Chunks                    Chunk Names
        ./scripts/bundle.js   3.43 kB       0  [emitted]         app
./scripts/vendor.bundle.js    275 kB       1  [emitted]  [big]  vendor
   [0] ./~/jquery/dist/jquery.js 268 kB {1} [built]
   [1] ./src/scripts/quote-manager.ts 2.42 kB {0} [built]
   [2] ./src/scripts/app.ts 475 bytes {0} [built]
   [3] ./src/scripts/quote.ts 261 bytes {0} [built]
   [4] ./src/scripts/vendor.ts 111 bytes {1} [built]
PS C:\Student\Modules\09_NodeJS\Lab\project2>
```

You should be able to verify that the webpack build process has generated two output files named **bundle.js** and **vender.bundle.js**. You should also note that the **vendor.bundle.js** file is displayed in yellow with **[big]** flag.

c) Look inside the **dist/scripts** folder and verify you can see the two files named **bundle.js** and **vendor.bundle.js**.

```
▲ PROJECT2
  ▲ dist
    ▲ scripts
      JS bundle.js
      JS vendor.bundle.js
  ▷ node_modules
  ▷ src
```
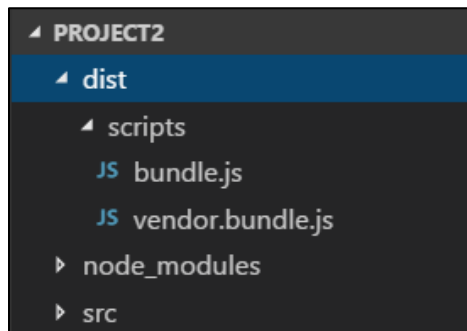
11. Configure the webpack build process to generate source map files to add debugging support.

   a) Return to the editor window with **webpack.config.js**.

   b) Move to the bottom of the file.

   c) Add a new section named **devtool** just under **module** section using the following code.

   ```
   devtool: 'source-map'
   ```

   d) The bottom of the **webpack.config.js** file should now match the following screenshot.

   ```
   module: {
     rules: [{
       test: /\.ts$/,
       use: 'ts-loader'
     }]
   },
   devtool: 'source-map'
   };
   ```

   e) Save your changes to **webpack.config.js**.

   f) Return to the console of the Integrated Terminal and execute the following **npm** command.

   ```
   npm run build
   ```

   g) Once the **build** command completes, you should be able to look inside **dist/scripts** and see that map files have been created.

   ```
   ▲ PROJECT2
     ▲ dist
       ▲ scripts
         JS bundle.js
         JS bundle.js.map
         JS vendor.bundle.js
         JS vendor.bundle.js.map
   ```
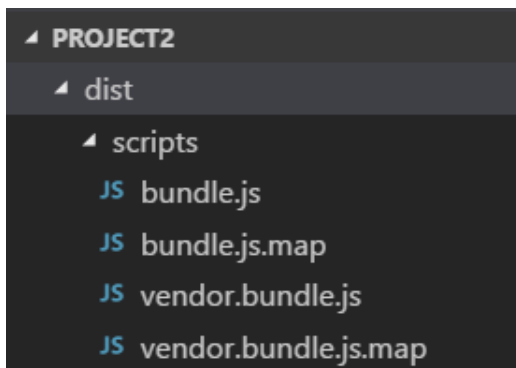
> These js.map files are used by JavaScript debuggers such as the browser-based debugger in Chrome to map JavaScript code back to the TypeScript source files which were used to generate them.

GSA2016: The Great SharePoint Adventure for Developers
Module 13 Lab: Developing with Node.JS, NPM and Visual Studio Code

Live Lab Version: Nov 6, 2017

12. Install the webpack plugin named **UglifyjsWebpackPlugin** to add JavaScript minification support into the build process.

    a) Type and execute the following **npm** command to install the **uglifyjs-webpack-plugin** package.

```
npm install uglifyjs-webpack-plugin --save-dev
```

    b) Return to the code editor window for **webpack.config.js**.

    c) Under the variables **path** and **CommonsChunkPlugin**, add a new variable named **UglifyJSPlugin** using the following code.

```
const path = require('path');
const CommonsChunkPlugin = require('webpack/lib/optimize/CommonsChunkPlugin');
const UglifyJSPlugin = require('uglifyjs-webpack-plugin');
```

    d) Move down inside **webpack.config.js** to the **plugins** section.

    e) Add a new instance of **UglifyJSPlugin** just beneath the plugin that's already there named **CommonsChunkPlugin**.

```
new UglifyJSPlugin()
```

    f) When you are done, the **plugins** section should match the following screenshot.

    g) Save your changes to **webpack.config.js**.

13. Rebuild the project to see the effects of minification provided by **UglifyJSPlugin**.

    a) Return to the console of the Integrated Terminal and execute the following **npm** command.

```
npm run build
```

    b) As the **build** command executes, you should see output in the console displaying the details of the webpack build process.

> Note that the two emitted JavaScript files are smaller than they were in the previous build. You should also take note that the JavaScript file named **vendor.bundle.js** no longer is displayed in yellow with the **[big]** flag.

    c) Open **bundle.js** and **vendor.app.js** in the **dist/scripts** folder to see what the minified JavaScript code looks like.

    d) Once you have seen the minified code, close **bundle.js** and **vendor.bundle.js**.

> At this point of this exercise, you have completed your work configuring the webpack build process to compile your TypeScript code into minified JavaScript code for distribution. Your next step is to update the index.html file with script links to the two generated JavaScript files named **bundles.js** and **vendor.bundle.js**.

14. Update the HTML file named **index.html** with script links for **bundles.js** and **vendor.bundle.js**.

    a) Open the **index.html** file in the **src** folder in an editor window.

    b) Move to the bottom of index.html and locate the comment which reads *add links to webpack bundles*.

    ```
    <!-- add links to webpack bundles -->

    </body>
    </html>
    ```

    c) Add two new script links for **bundles.js** and **vendor.bundle.js**.

    ```
    <!-- add links to webpack bundles -->
    <script src="scripts/vendor.bundle.js"></script>
    <script src="scripts/bundle.js"></script>
    ```

    d) Save ad close **index.html**.

Your final step in building out the **dist** folder will be to configure the build process to copy the three other files in the **src** folder named **index.html**, **app.css** and **AppIcon.png**. You will accomplish this by installing and using the plugin named **copy-webpack-plugin**.

15. Install and configure the **copy-webpack-plugin** plugin to bundle the remaining project files into the **dist** folder.

    a) Return to the console of the Integrated Terminal and execute the following **npm** command.

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

    b) Return to the code editor window for **webpack.config.js**.

    c) Add a new variable named **CopyWebpackPlugin** using the following code.

    ```
    const path = require('path');
    const CommonsChunkPlugin = require('webpack/lib/optimize/CommonsChunkPlugin');
    const UglifyJSPlugin = require('uglifyjs-webpack-plugin');
    const CopyWebpackPlugin = require('copy-webpack-plugin');
    ```

    d) Move down to the **plugins** section and add a new instance of **CopyWebpackPlugin** under the two other plugins.

    ```
    new CopyWebpackPlugin([
      { from: './src/index.html', to: 'index.html' },
      { from: './src/css/app.css', to: 'css/app.css' },
      { from: './src/css/img/AppIcon.png', to: 'css/img/AppIcon.png' }
    ])
    ```

    e) At this point, your **plugins** section should match the following screenshot.

    ```
    plugins: [
      new CommonsChunkPlugin({
        name: 'vendor',
        filename: './scripts/vendor.bundle.js'
      }),
      new UglifyJSPlugin(),
      new CopyWebpackPlugin([
        { from: './src/index.html', to: 'index.html' },
        { from: './src/css/app.css', to: 'css/app.css' },
        { from: './src/css/img/AppIcon.png', to: 'css/img/AppIcon.png' }
      ])
    ],
    ```

    f) Save your changes to **webpack.config.js**.

16. Rebuild the project to see the effects of minification provided by **UglifyJSPlugin**.
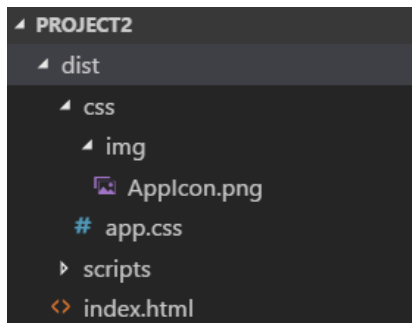
    a) Return to the console of the Integrated Terminal and execute the following **npm** command.

    ```
    npm run build
    ```

    b) As the **build** command executes, you should see output in the console indicating that the build process is now integrating **index.html**, **app.css** and **AppIcon.png**.

    ```
                        Asset       Size  Chunks             Chunk Names
            ./scripts/bundle.js    2.39 kB       0  [emitted]  app
     ./scripts/vendor.bundle.js    88.8 kB       1  [emitted]  vendor
                     index.html  787 bytes          [emitted]
                    css/app.css    1.66 kB          [emitted]
            css/img/AppIcon.png  703 bytes          [emitted]
    ```

    c) Look inside the **dist** folder and verify you can see the files **index.html**, **app.css** and **AppIcon.png**.

    ```
    ◢ PROJECT2
      ◢ dist
        ◢ css
          ◢ img
            🖼 AppIcon.png
          # app.css
        ▸ scripts
        <> index.html
    ```

> At his 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.

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

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

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

    b) Return to the editor window for **webpack.config.js**.

    c) At the bottom of **webpack.config.js** under the **devtool** section, add a new **devServer** section using the following code.

    ```
    devServer: {
      contentBase: 'dist' ,
      port: 1234
    }
    ```

    d) When you are done, the bottom of the **webpack.config.js** file should match the following code listing.

    ```
      devtool: 'source-map',
      devServer: {
        contentBase: 'dist' ,
        port: 1234
      }
    };
    ```

    e) Save your changes to **webpack.config.js**.

18. 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**.

19. 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 start in the browser and should run as expected.
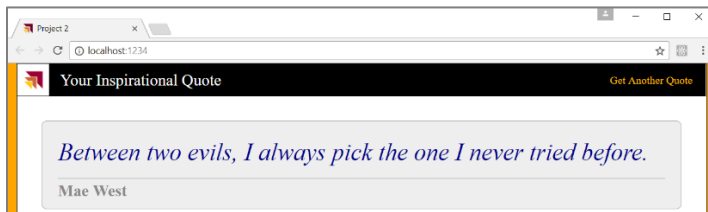
    d) While the application is running, try to update source files inside the **src** folder such as **index.html** or **app.css**. When you save your changes, the webpack file watch support should automatically rebuild the project and refresh the browser to show your changes.

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**.