Southern New Hampshire University

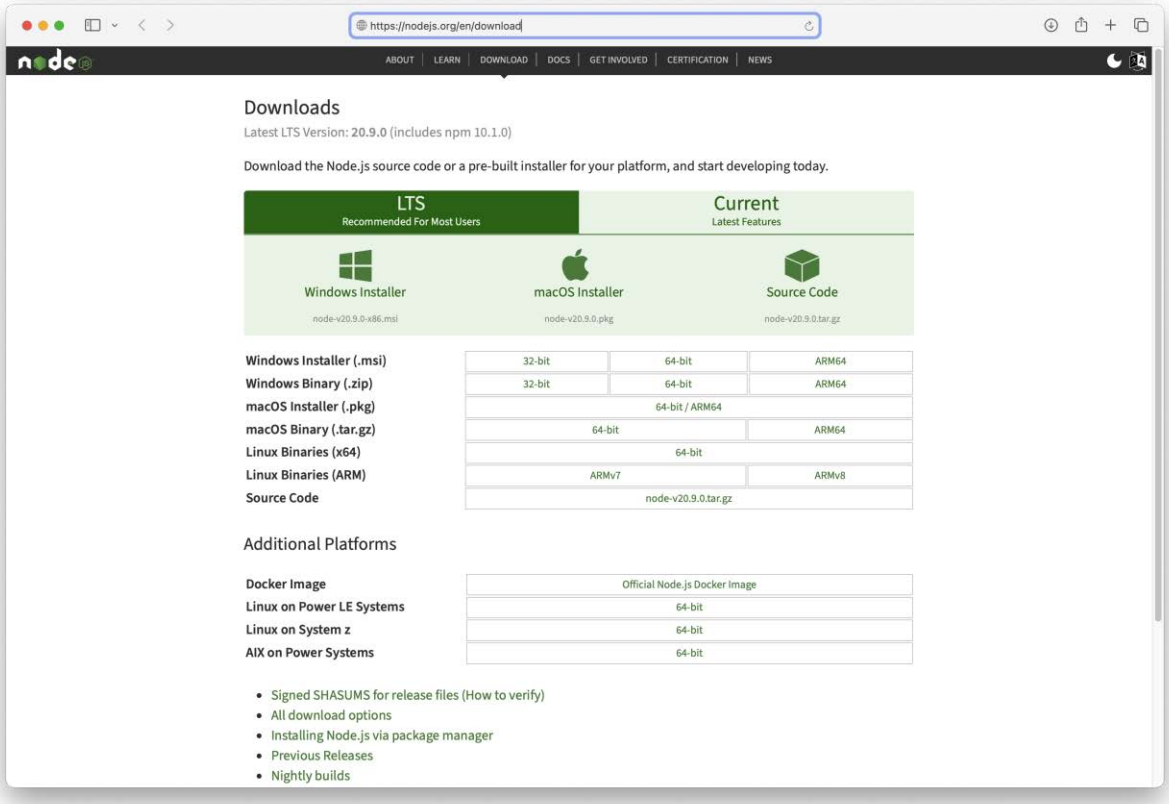
CS 465 – Full Stack Development
Full Stack Guide – Complete

Table of Contents

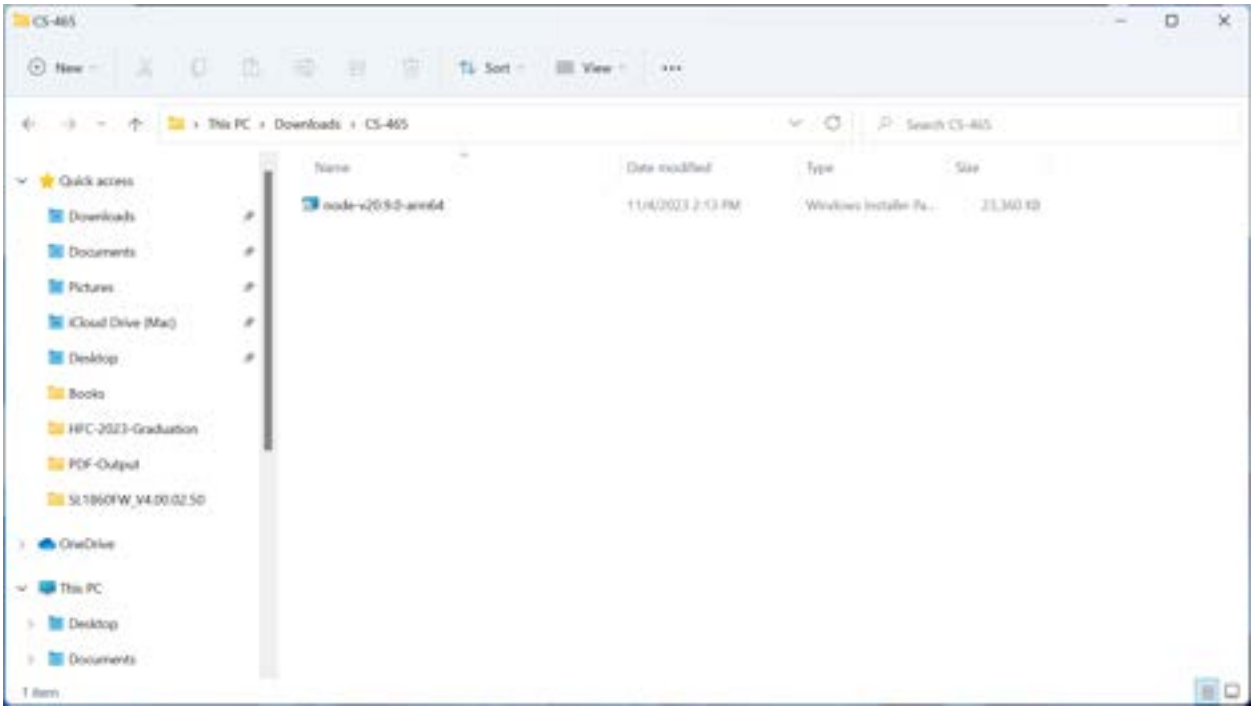
Module 1: Setting Up Your Environment

Installing Node.js and Node Package Manager (NPM)

The first package that we will install for this class is Node.js. This package can be obtained from: https://nodejs.org/en/download and is available for multiple operating systems and platforms. This guide will focus on deploying to a Windows 11 platform, but you can accomplish the same tasks with Windows 10, MacOS, or Linux.

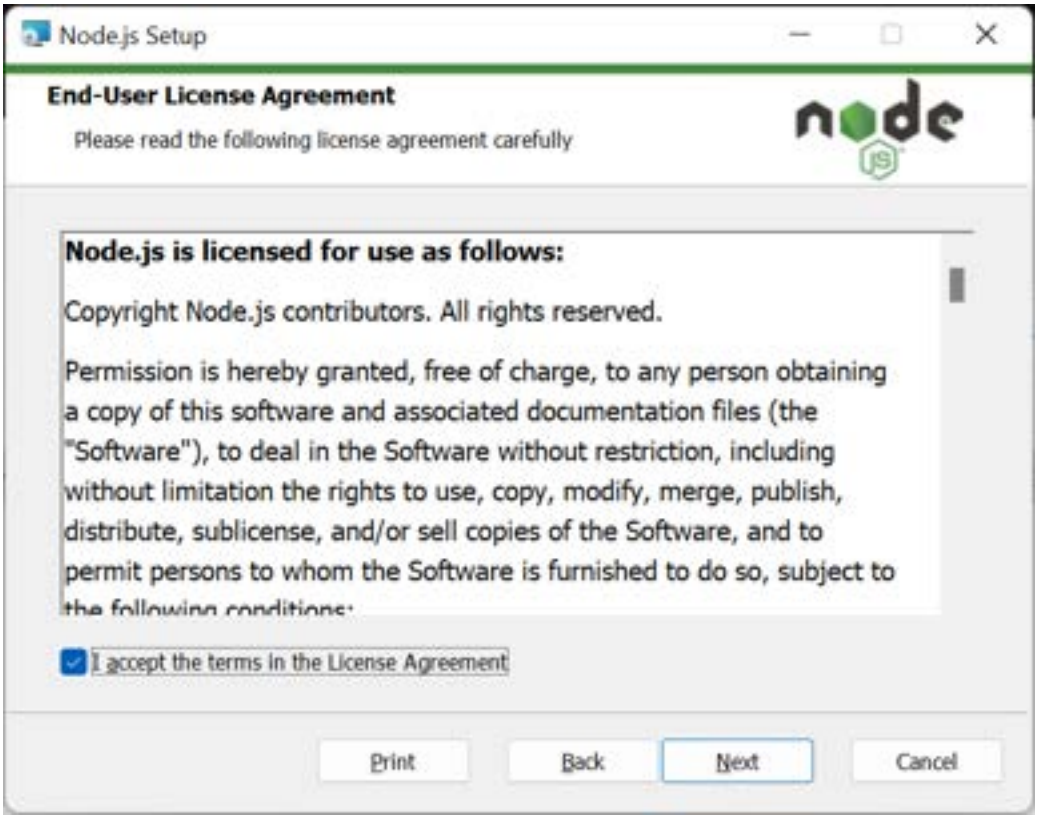


When you reach the download page, you are going to want to select the LTS (Long Term Support) version of Node.js for the operating system you are using. This will download the installer for your operating system to your default download directory.

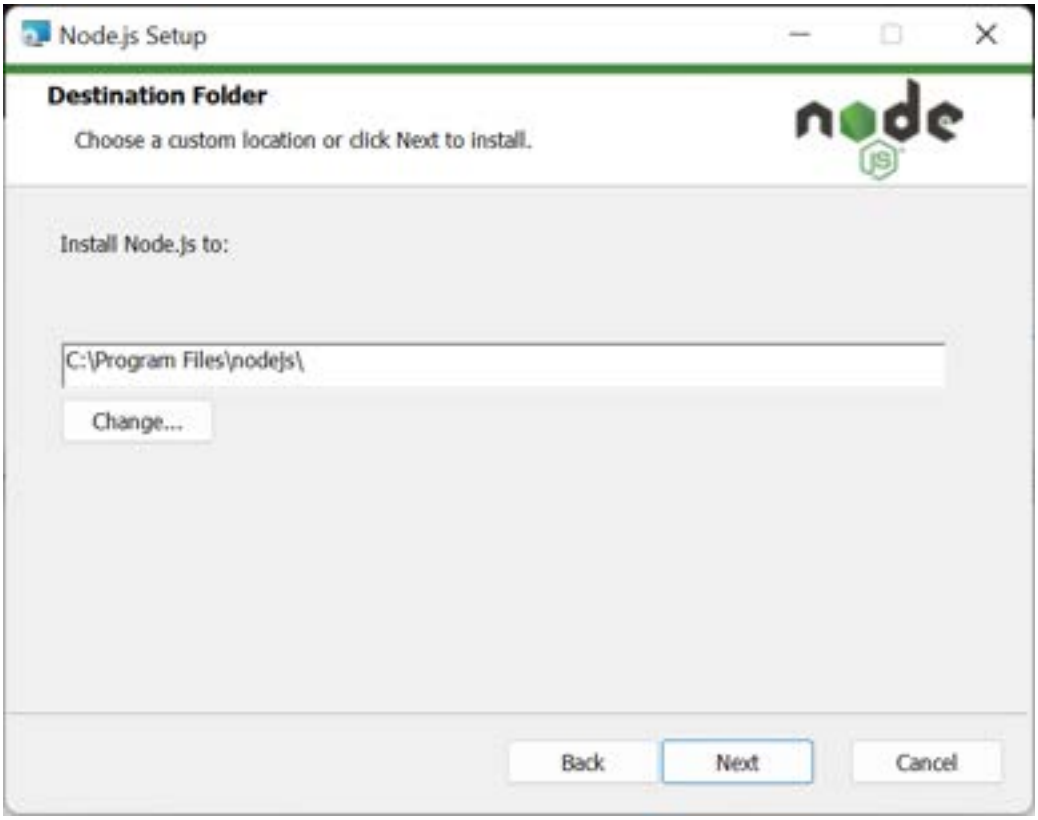


In this case, you will double-click on the installer to start the installation process.

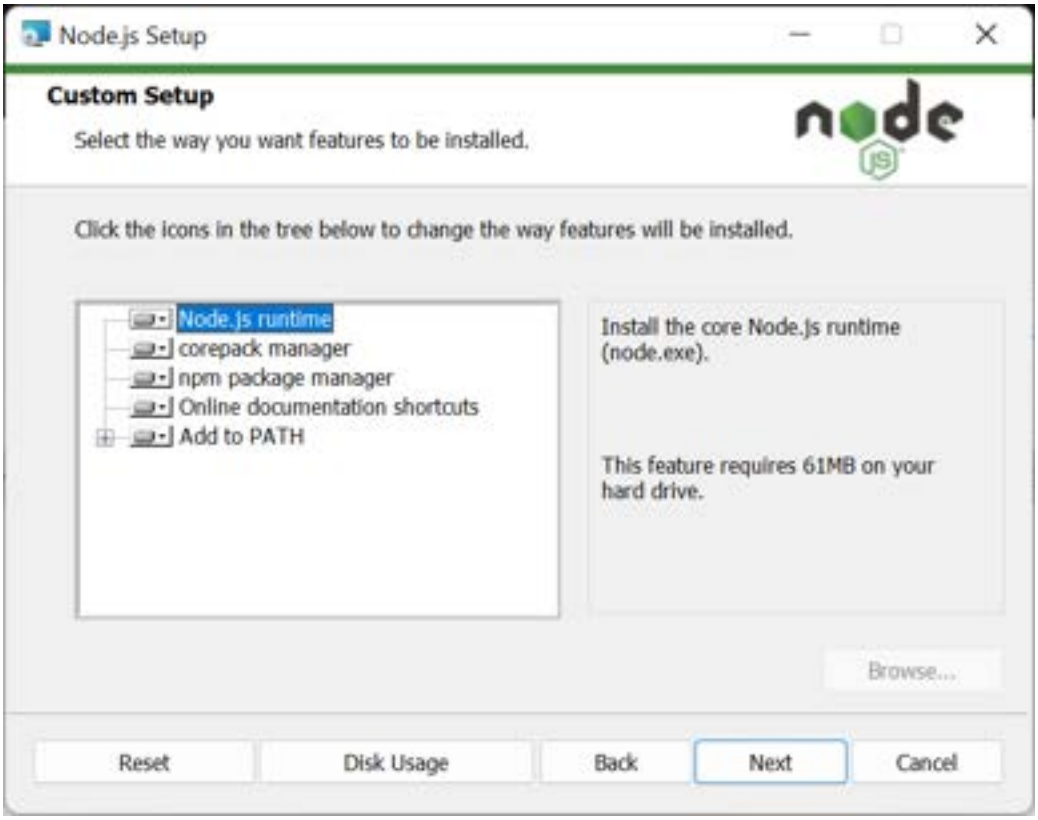
Selecting the Next button will bring up the End User License Agreement (EULA) that you must read and acknowledge acceptance of to continue the software install. Once you have read the EULA and checked the box agreeing to its terms, press Next.



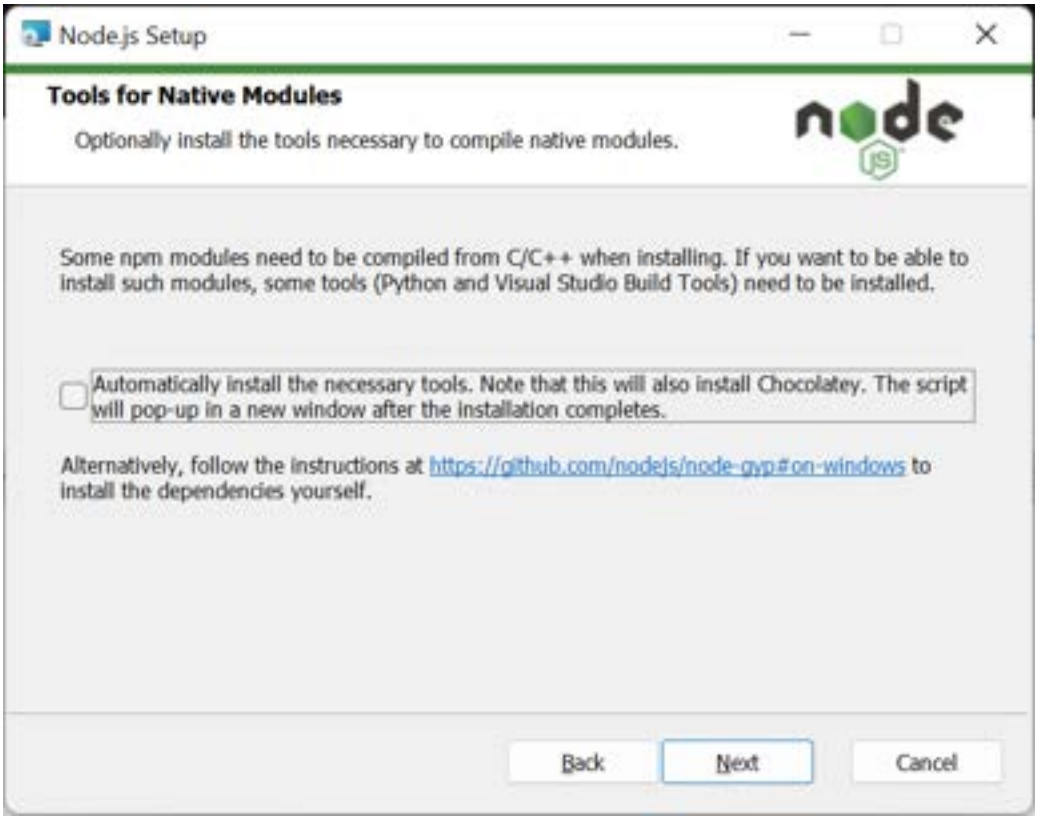
The next step requires you to choose where you are going to install the Node.js software. You can install this in a folder of your choice however, this guide is based on the default location of the software on a Windows computing platform (C:\Program Files\nodejs\). Select Next when you have made your selection.



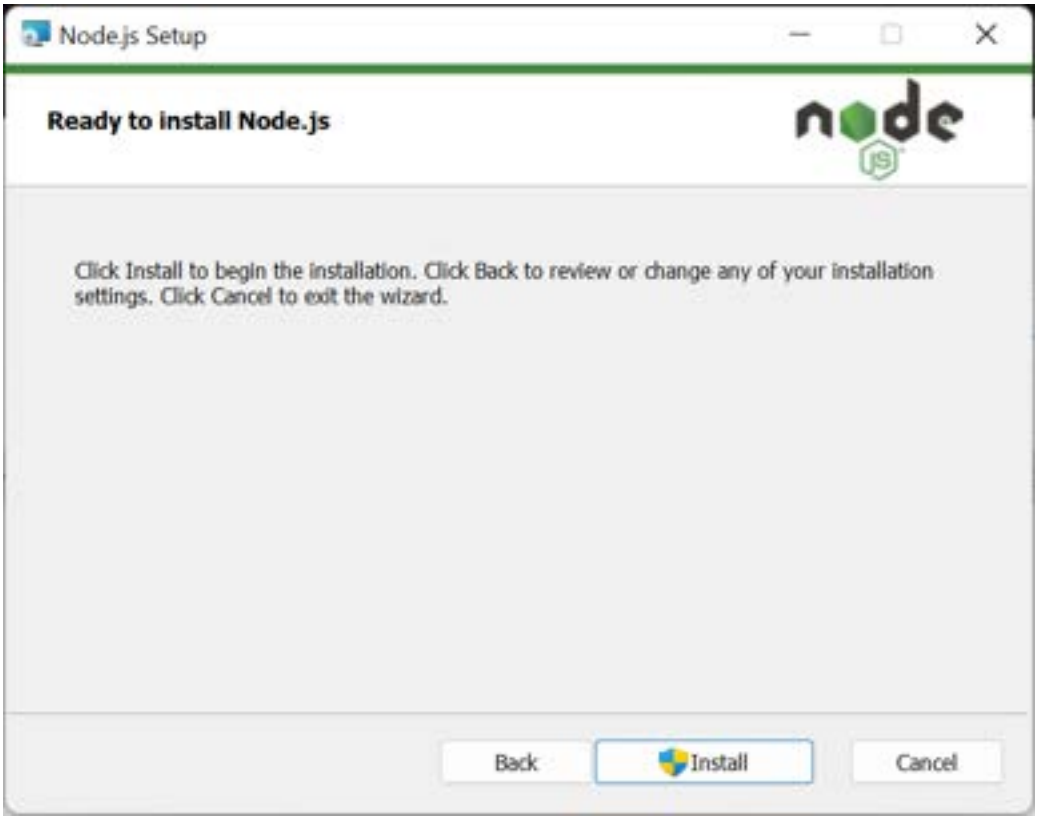
The next screen provides the opportunity to customize the installation. For the purposes of this class, we will be installing the entire package. This does not require any modification of options. Select Next to continue.



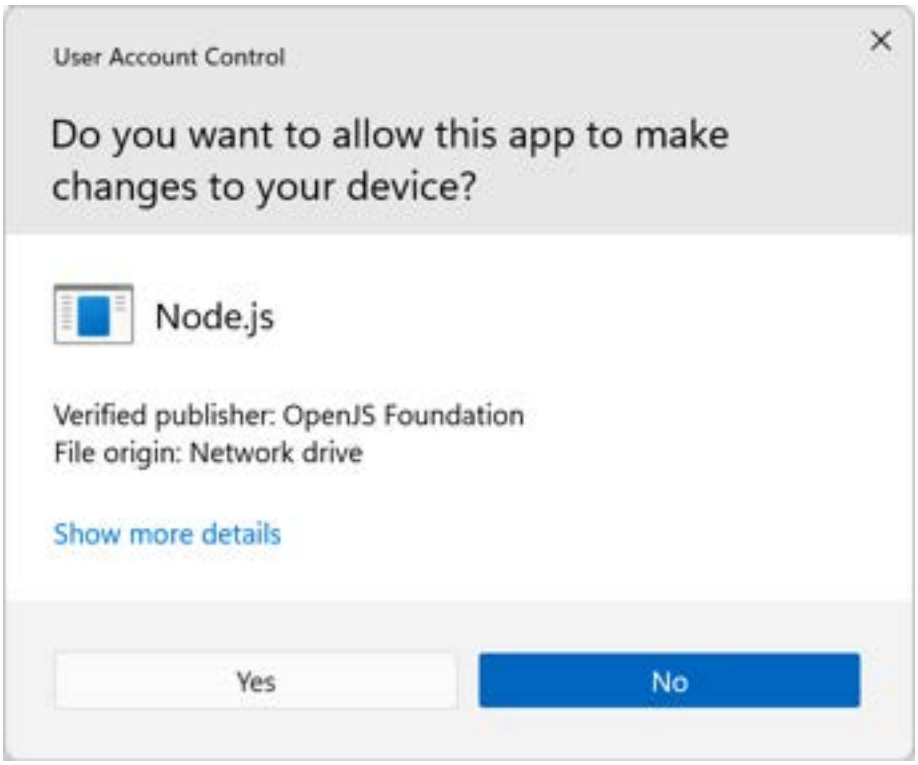
The next screen provides the opportunity to install tools for working with Native Modules. We will not be doing that in this course, and you should ***not*** select the checkbox that would install the extra material. Select Next to continue.



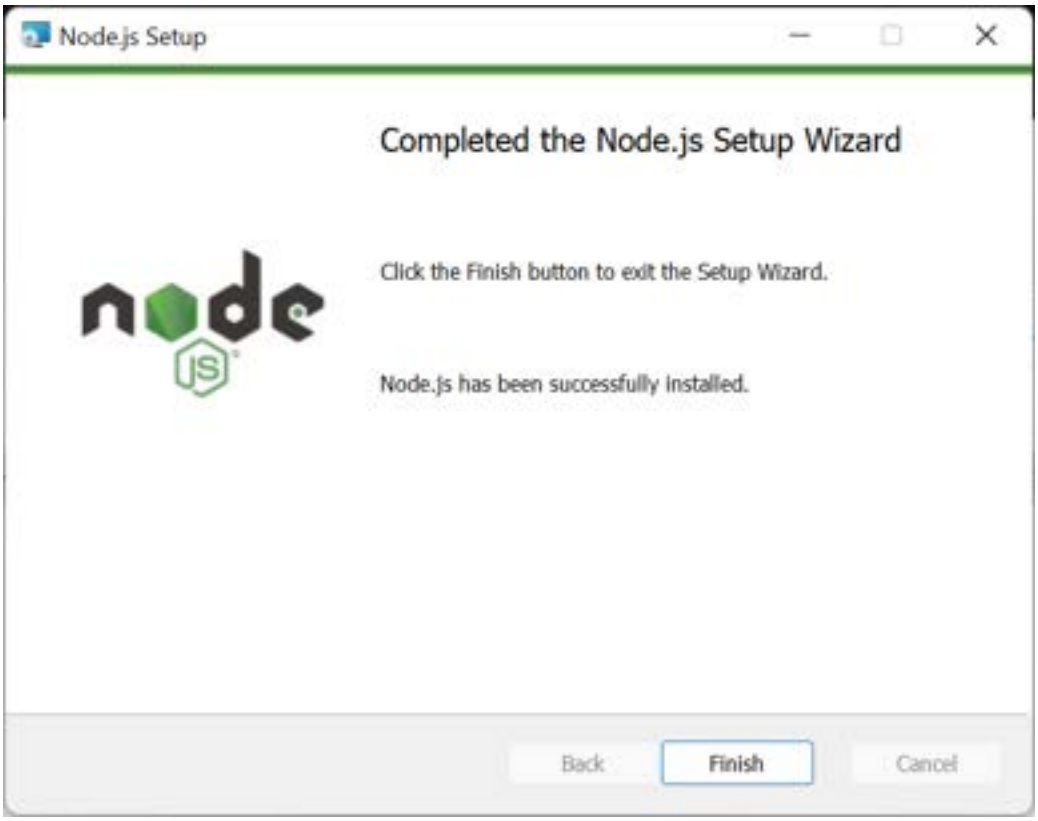
This is the end of the setup for Node.js, select the Install button to begin the installation.



You will have to acknowledge that the installation app wants to make changes to your device. Select Yes to continue.



When the software has successfully installed you will see the Completion window. Select Finish to close the installation process.

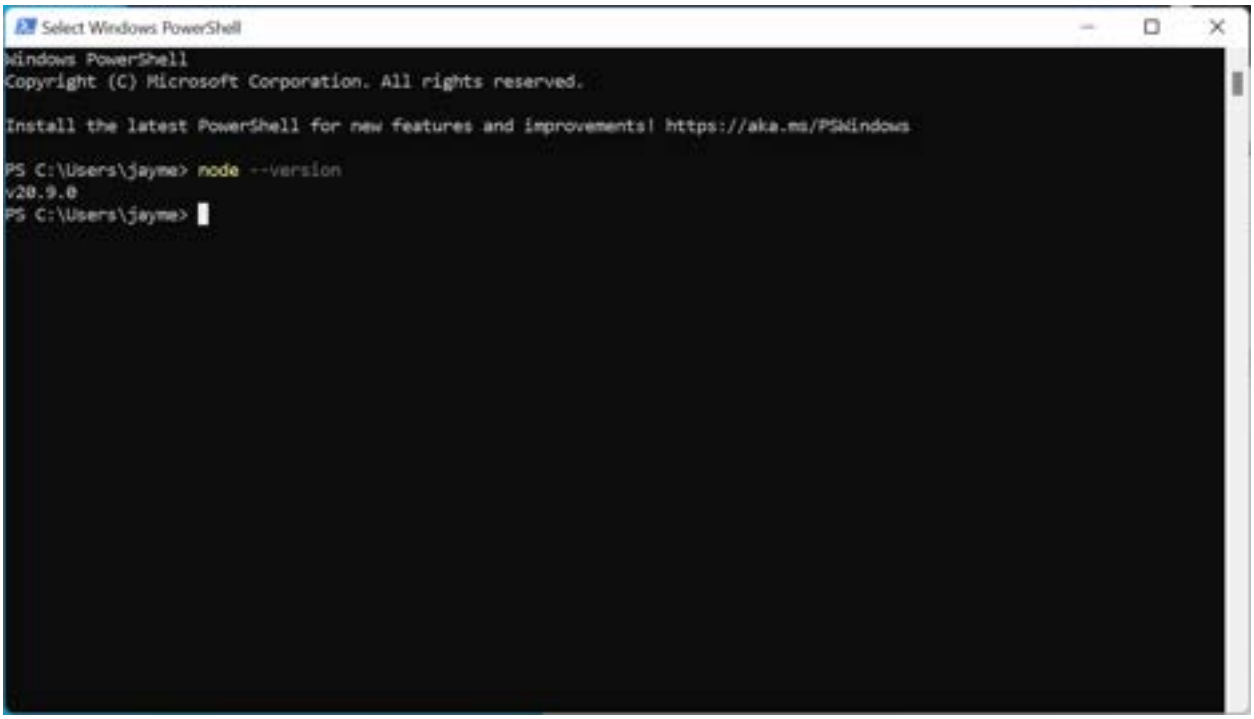


Now that the installation is complete, you are going to want to test and make sure that everything works. To do this, you will want to open a PowerShell window (you can do this by typing PowerShell in the search bar and hitting return).

In your PowerShell window, you will want to execute the following command:

```
node --version
```

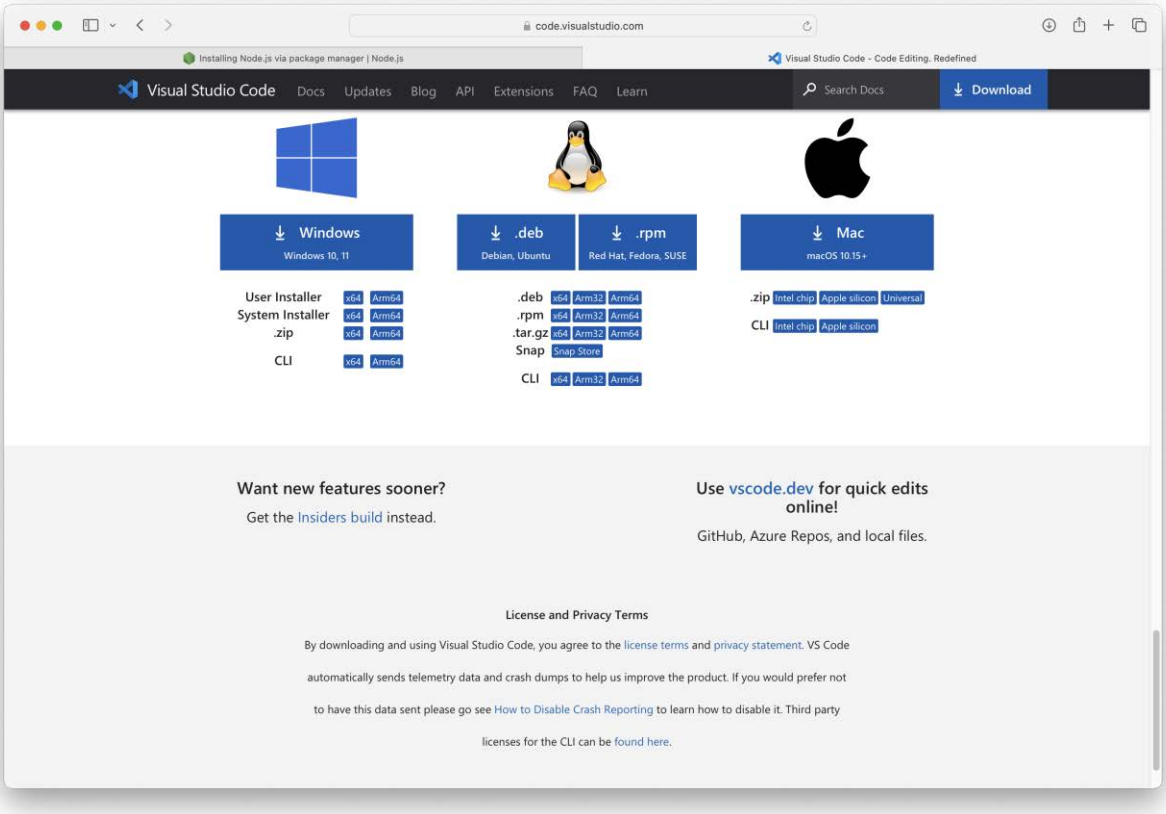
This should match the version of Node.js that you downloaded.



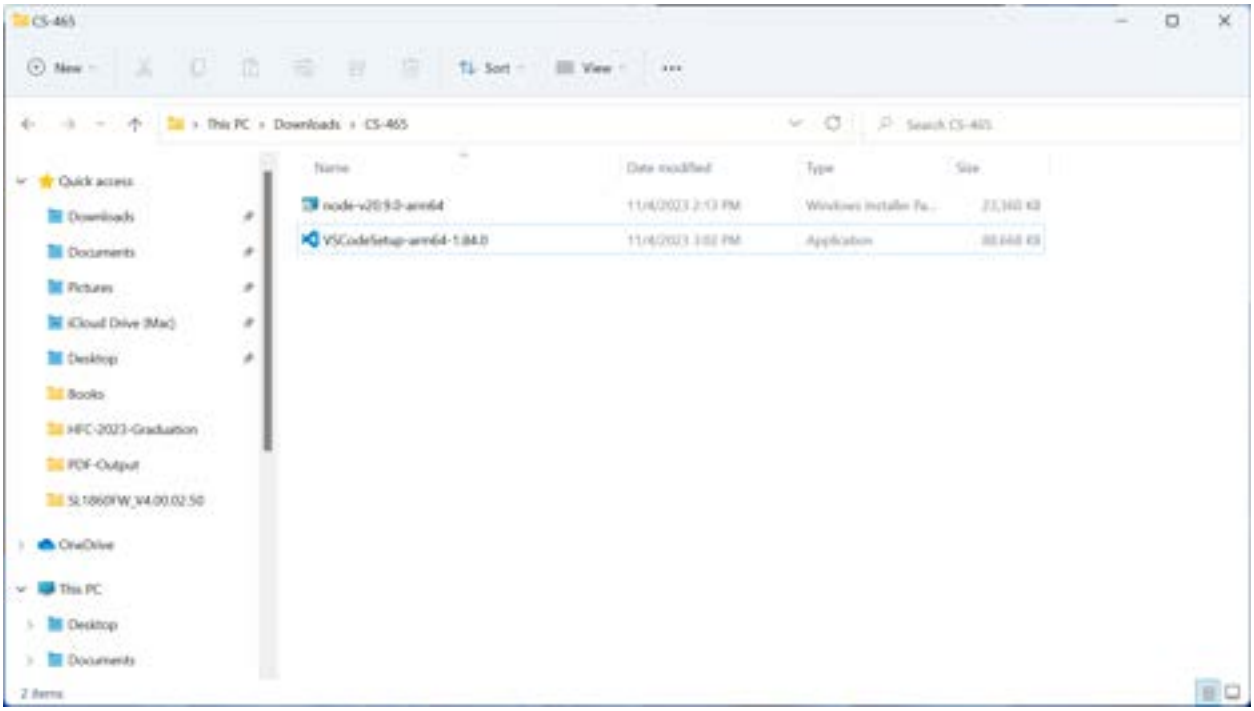
Your installation of Node.js has been successfully completed and is ready for use in this course.

Installing Visual Studio Code (VS Code)

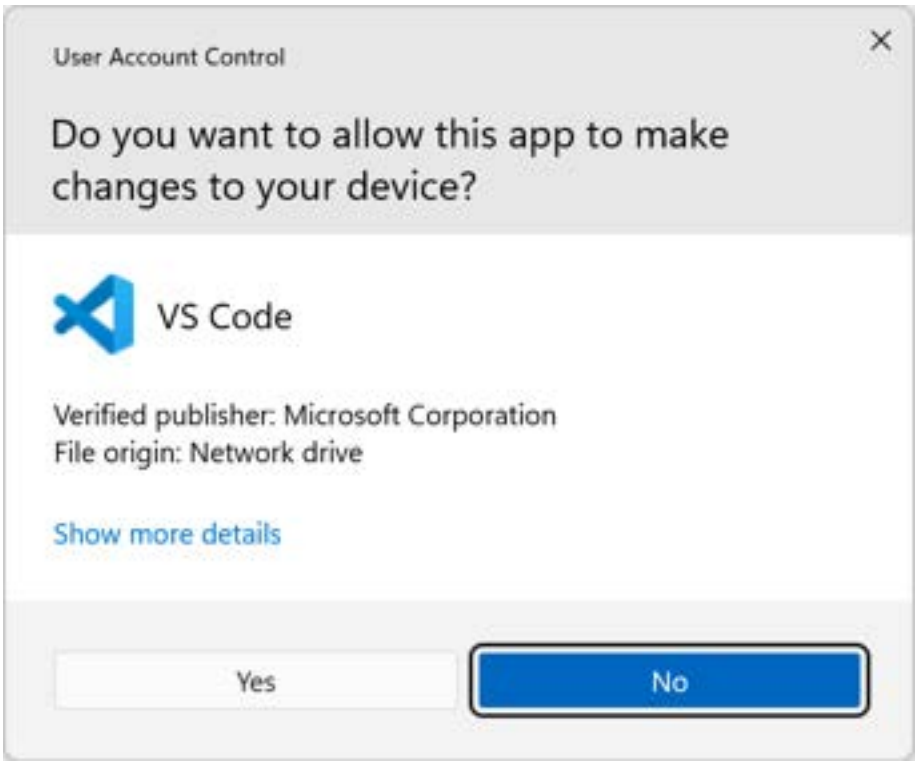
The next step in preparing the environment is the installation of Visual Studio Code (VS Code), an Integrated Development Environment. VS Code will allow you to manipulate files and file structures as well as providing a good platform for integrating with Git for version control and supplying a good editor for developing and maintaining source code. You will be able to download the necessary package here: https://code.visualstudio.com/#alt-downloads.



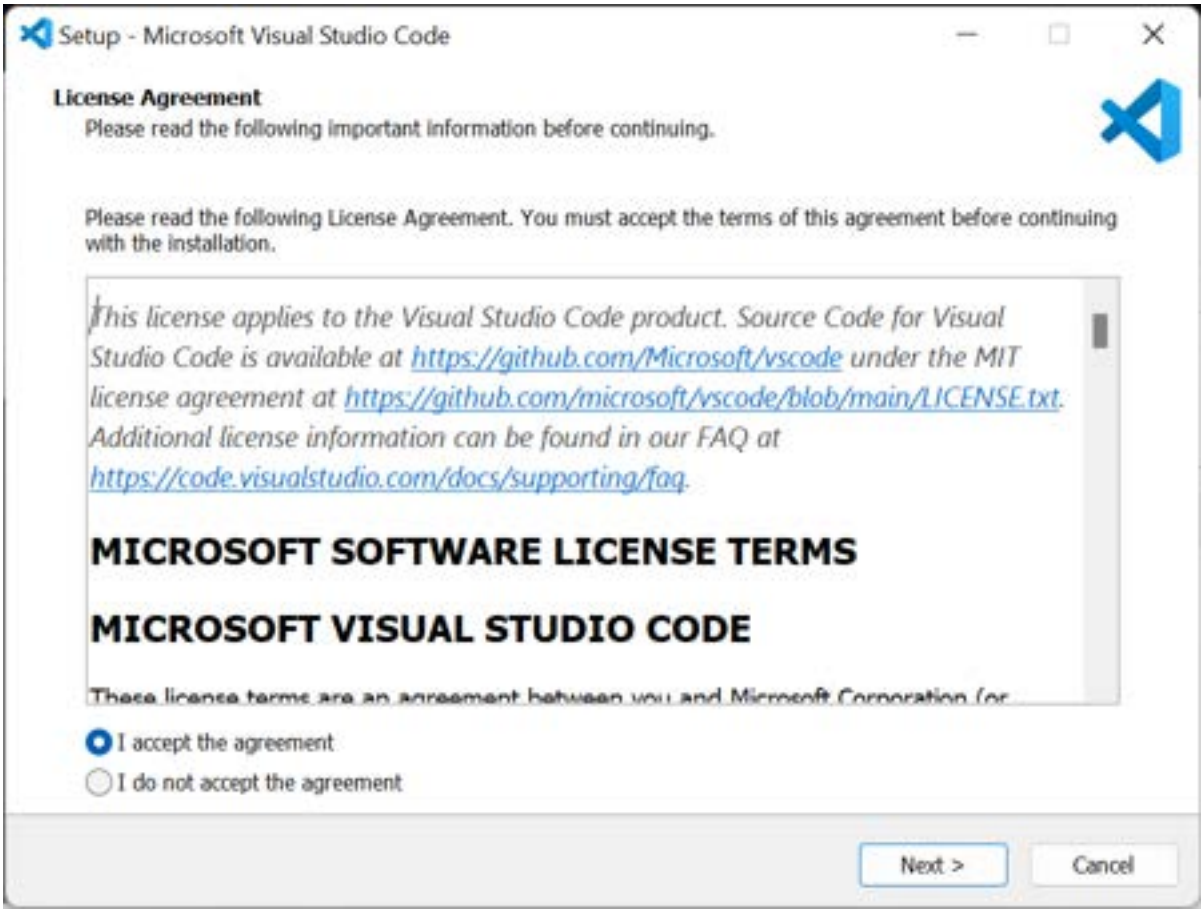
VS Code is available for many platforms and provides the same functionality across all instances. This guide will show the utilization of VS Code on a Windows 11 platform. Select the appropriate installer for your operating system and it should download to your browser's default download directory.



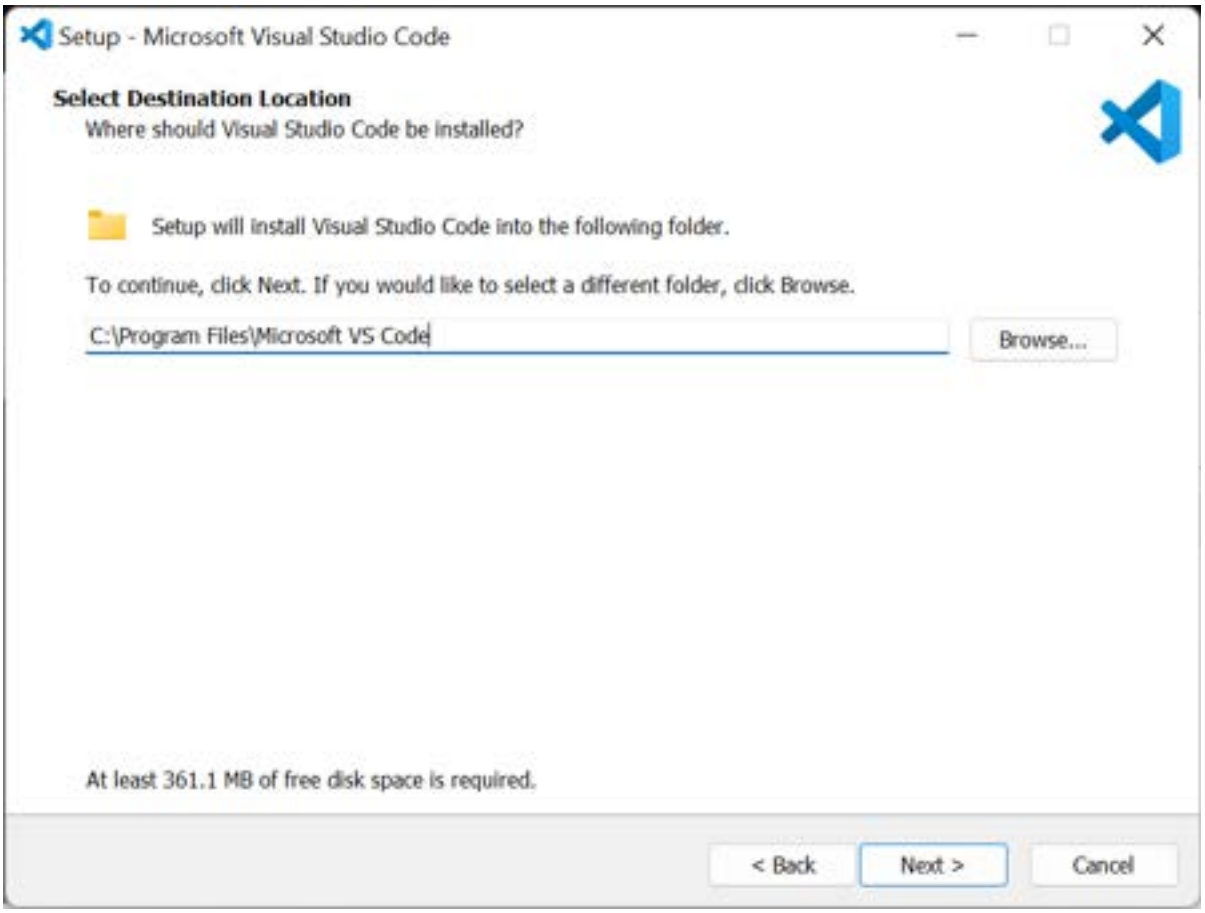
Double-clicking on the install file will open the installer to begin the installation process. Again, you will need to acknowledge that the installer needs to make changes to your environment. Select Yes to continue.



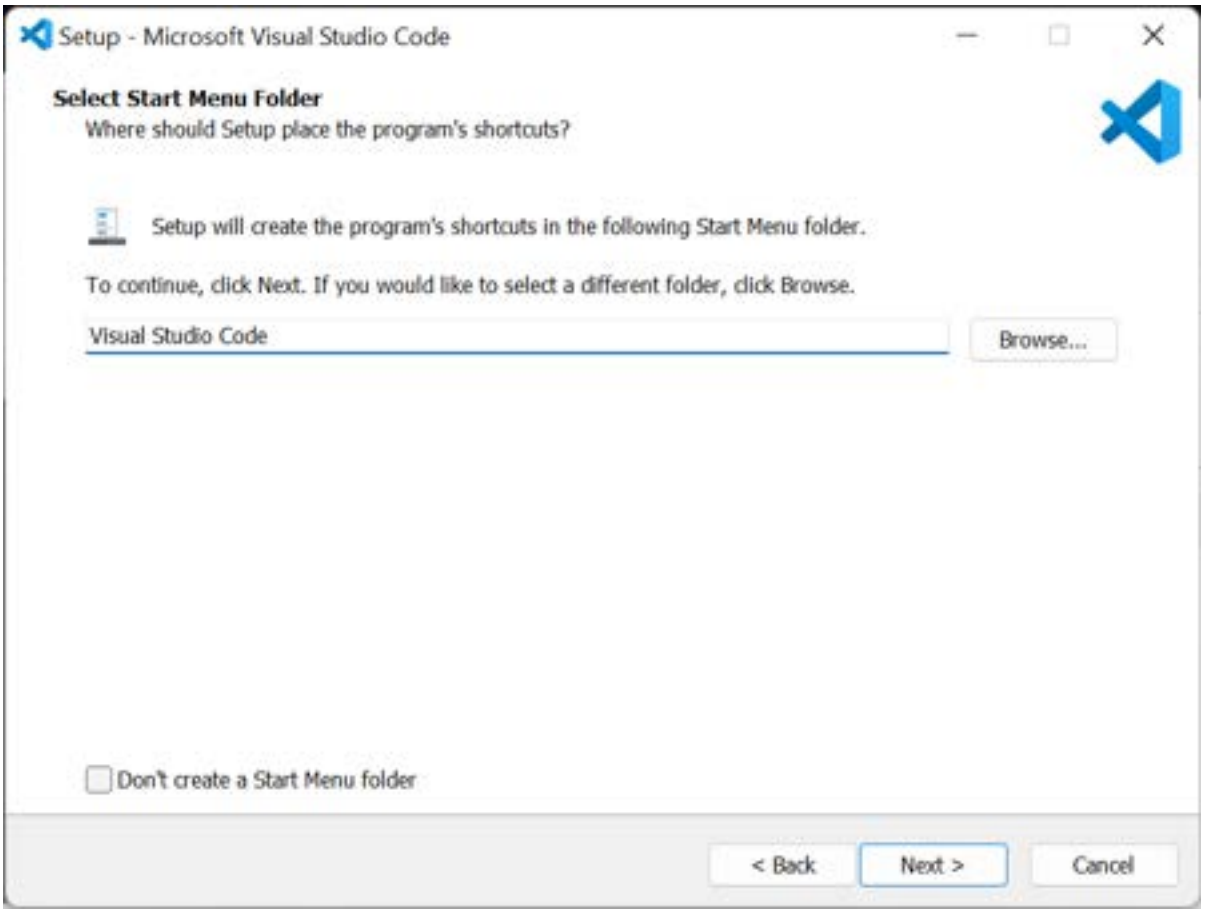
The next screen is the License Agreement for the software. Please read through the license agreement before continuing. When you have read and agreed to the EULA, select Next to continue.



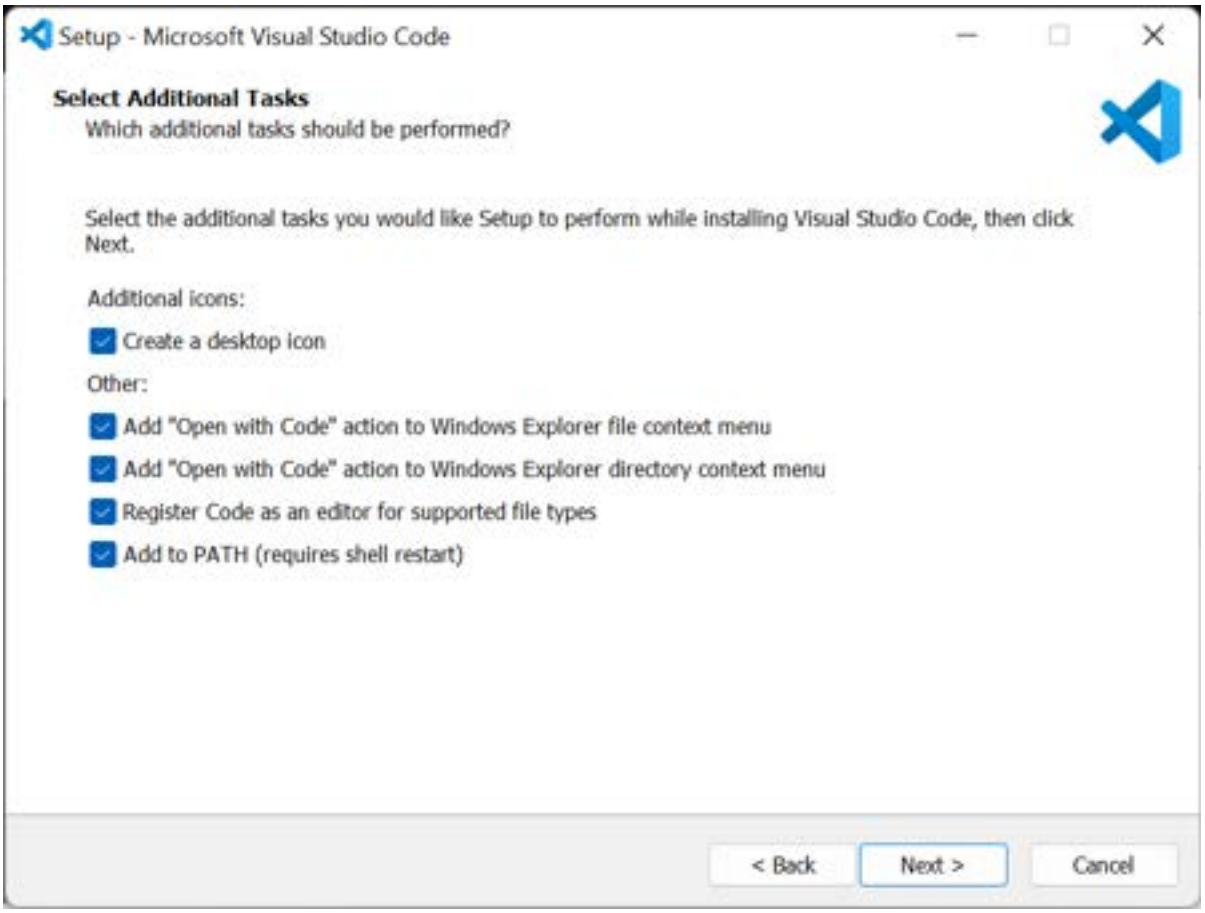
The next screen provides you with a choice for where you want to install the software. This guide is written with the software installed in its default location: C:\Program Files\Microsoft VS Code. Select Next to continue.



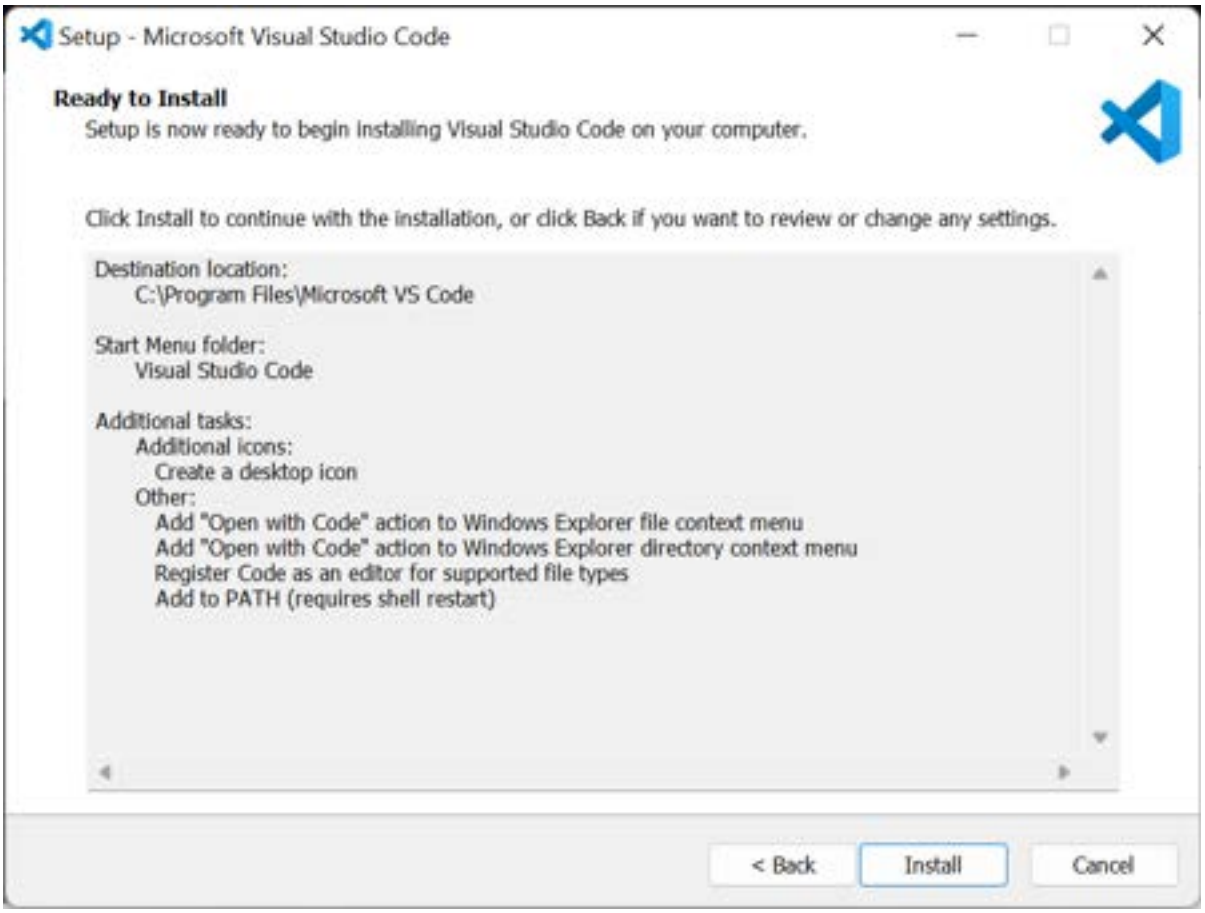
The next screen provides the option for you to name a folder that the installer will create in your Start Menu to store the VS Code short-cuts in. Select Next to continue.



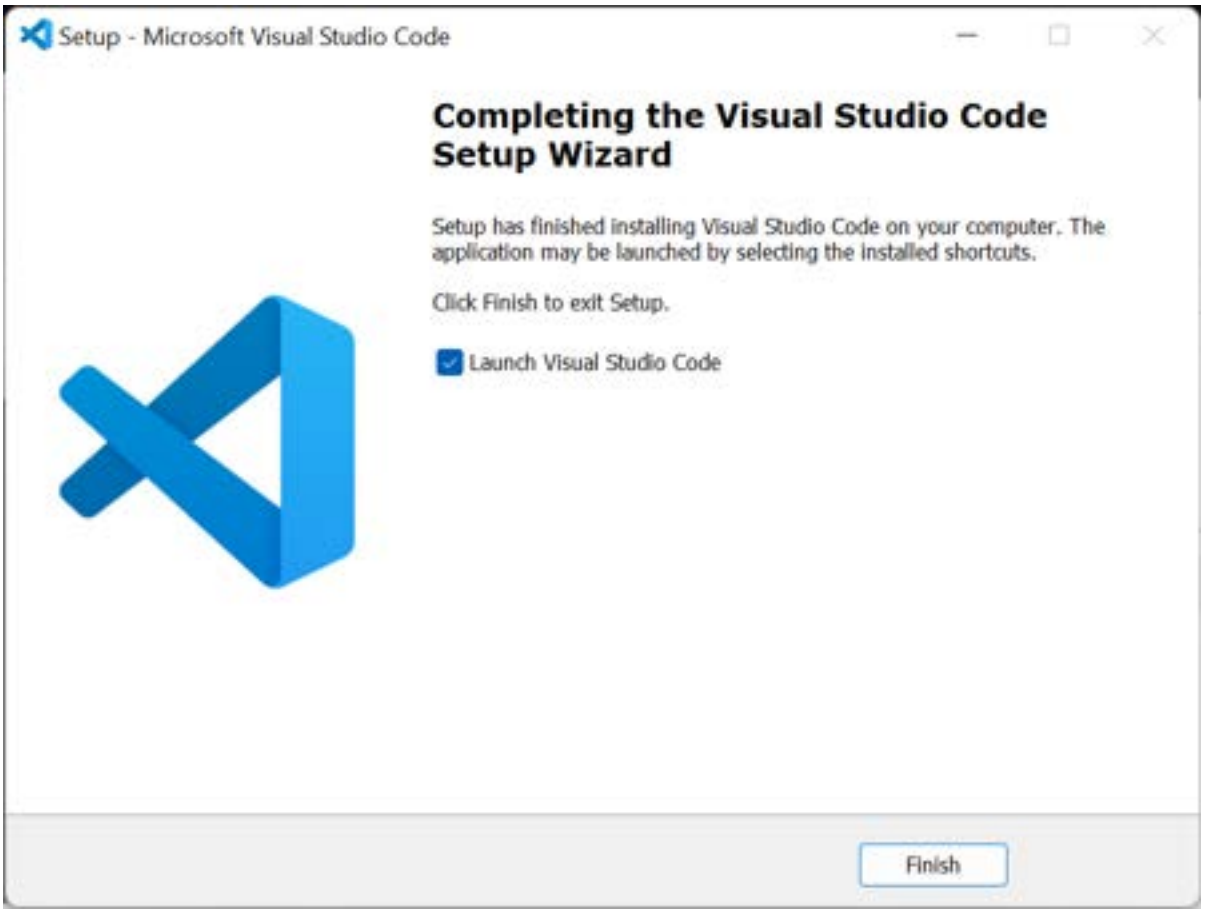
The next screen provides some installation options. It is convenient when using this editor to have all the features denoted by these options, so I have selected them all for this example. Select Next to continue.



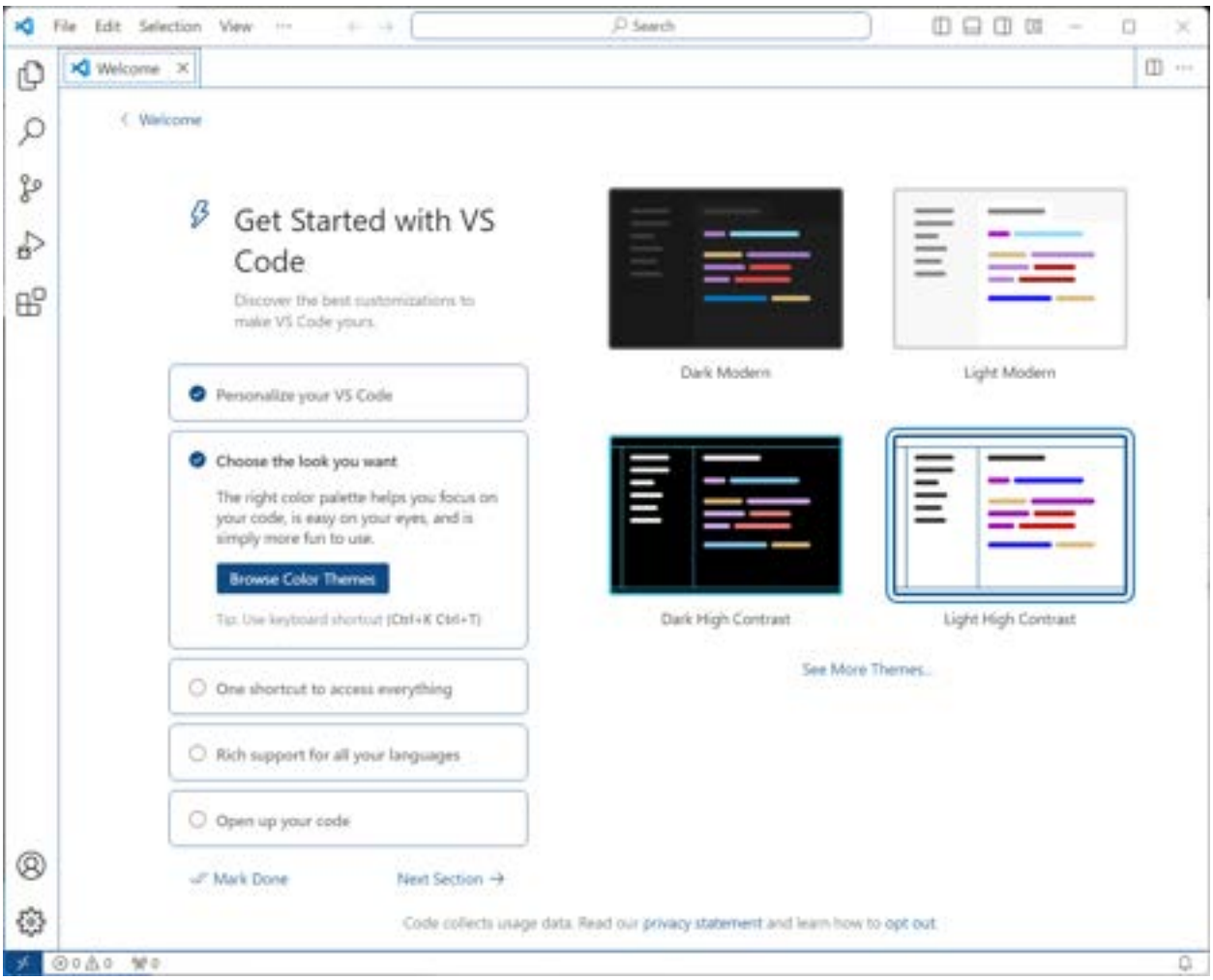
The next screen provides you an opportunity to review your selections. When you are happy with what is selected, press Install to continue (you can use the back button to return to a previous screen and make changes if necessary).



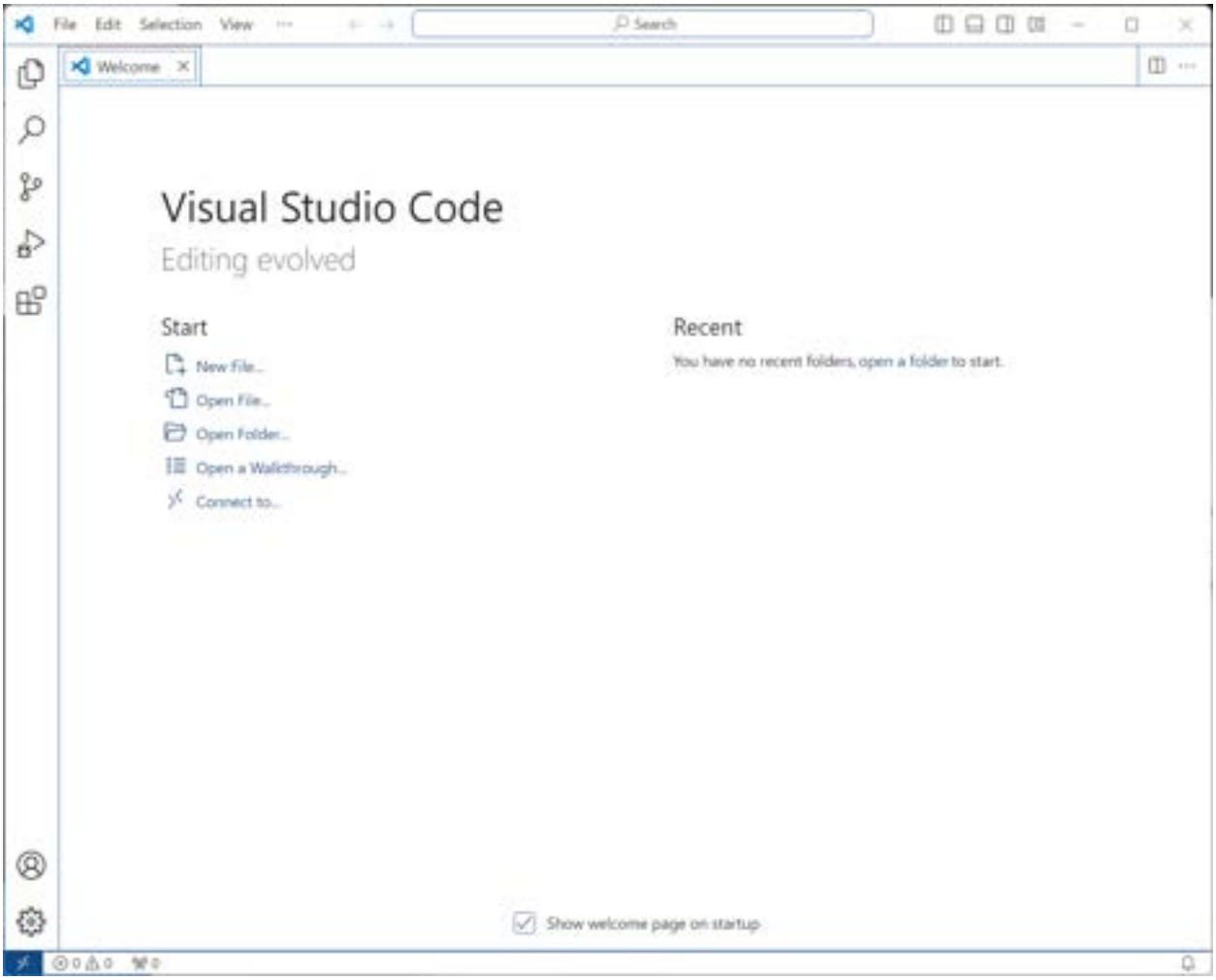
The next screen asks that you launch VS Code on the completion of the installation. Select finish to continue.



VS Code Opens up in dark mode by default. Click on the block ‘Choose the look you want’ to expand it into a panel where you can select your theme. I select ‘Light High Contrast’ because it helps me to more easily identify key topics in the editor – select whichever theme you like best.



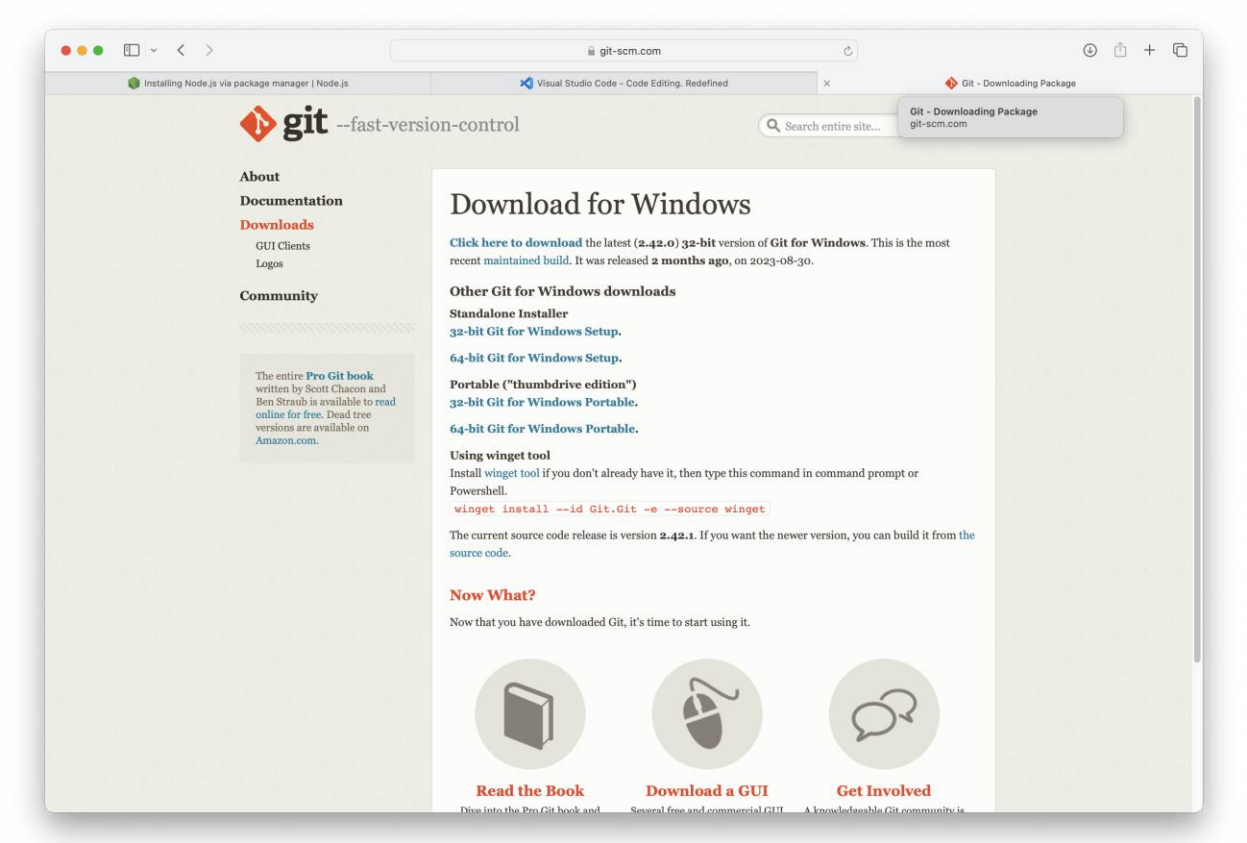
At this point I select the ‘Mark Done’ choice beneath the menu.



You can choose whether you would like to see the welcome page on startup by selecting or de-selecting the checkbox at the bottom of the page. You can now close or minimize VS Code as your initial install is now complete.

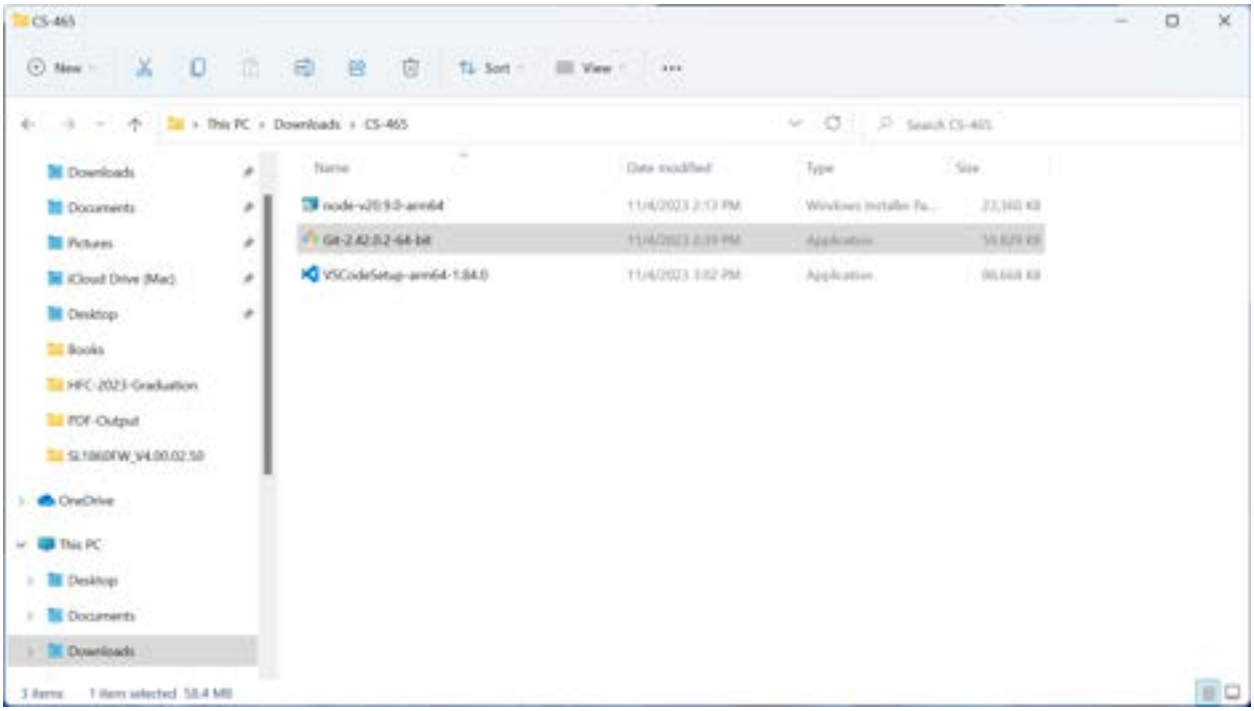
Installing the Git Distributed Version Control System

The next step in preparing your environment is the installation of a Git client for your operating system. The software will be available here: https://git-scm.com/downloads.

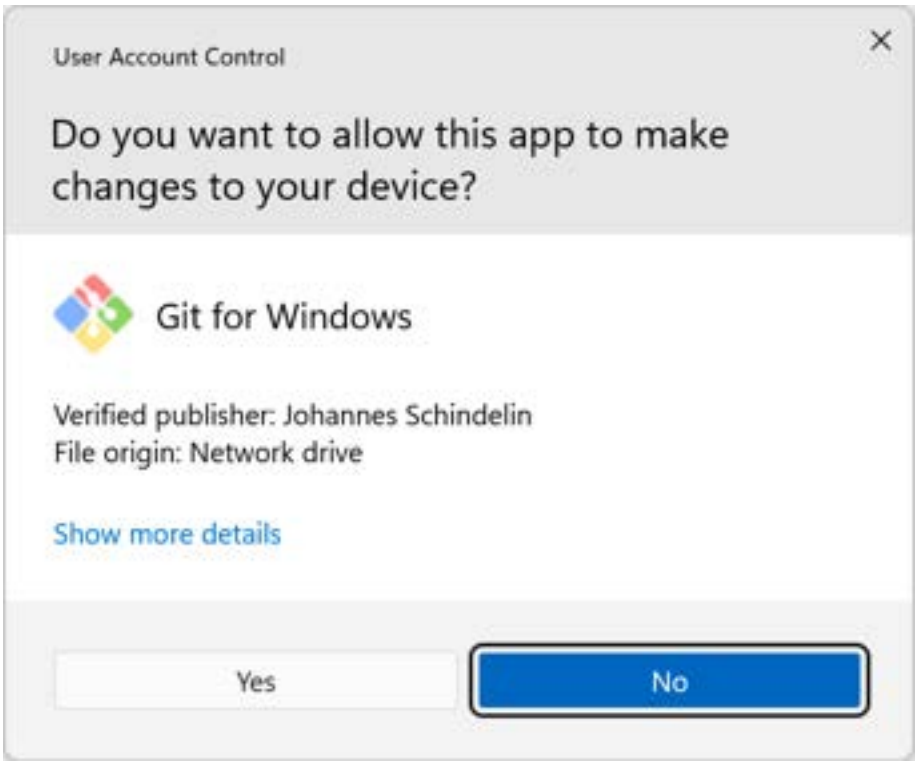


You need to select the package that is appropriate for your operating environment. This guide is focused on a Windows 11 platform, but you can install git on Linux and MacOS as well (it is generally pre-installed on MacOS when you have installed the development tools).

For this document, I selected the 64-bit Git for Windows setup, and it downloaded to my default downloads folder.

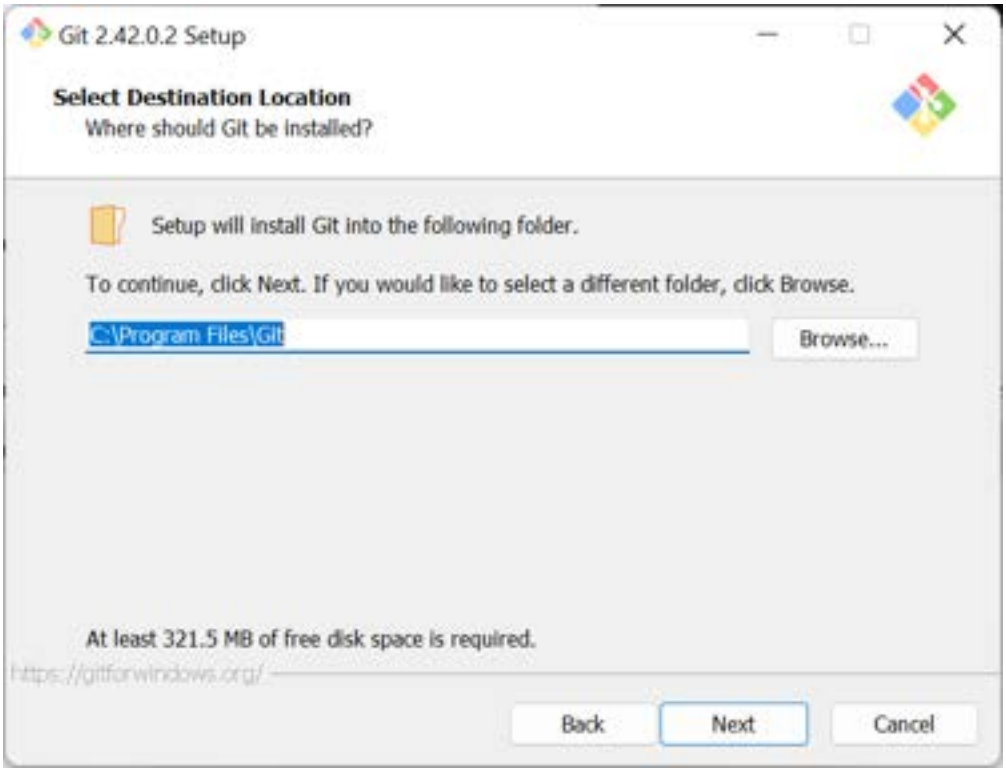


You can double-click on the Git executable file to begin the installation process. You will again have to acknowledge that the app can make changes to your environment. Select Yes to continue.

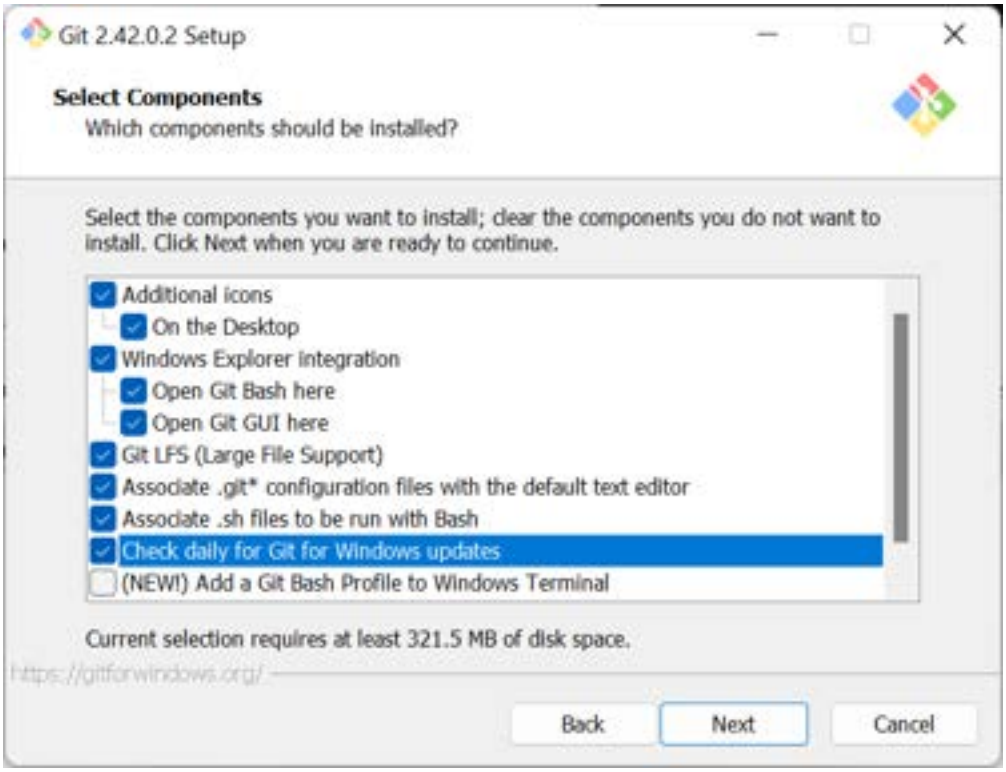


The next window you see is for the acknowledgement of the software license. The Git software is made available under the GNU General Public License. Please read and acknowledge the license by selecting the Next button.

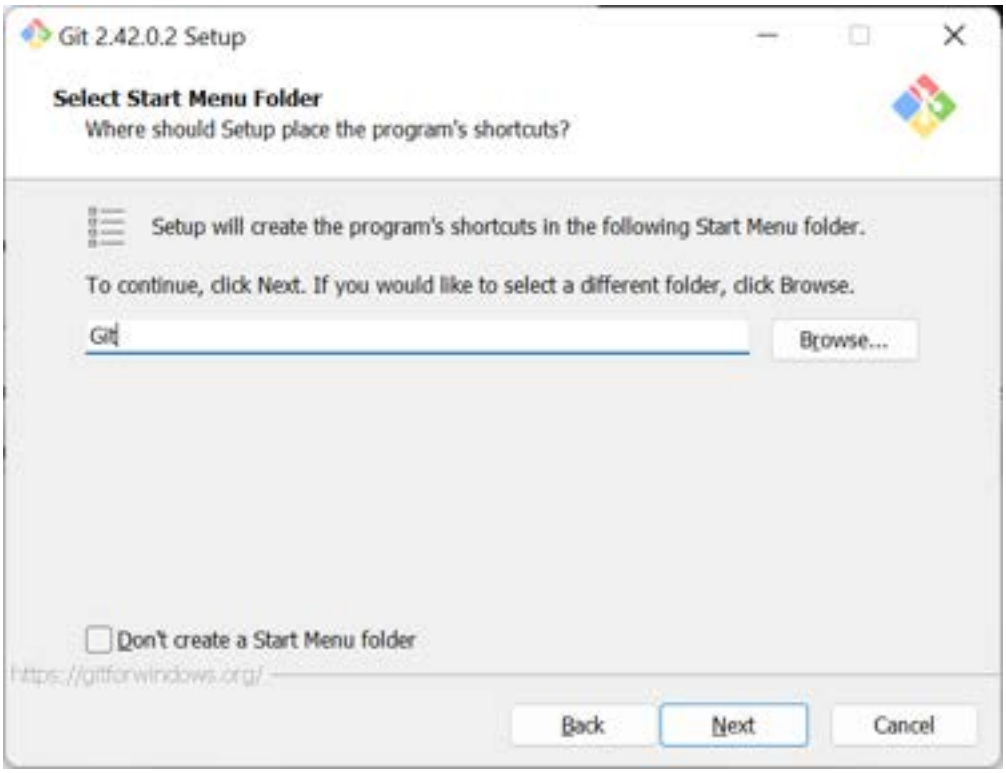
Like the Node.js install, you can choose where this package is installed. For this example, we are using the default installation directory: C:\Program Files\Git. Select Next to continue.



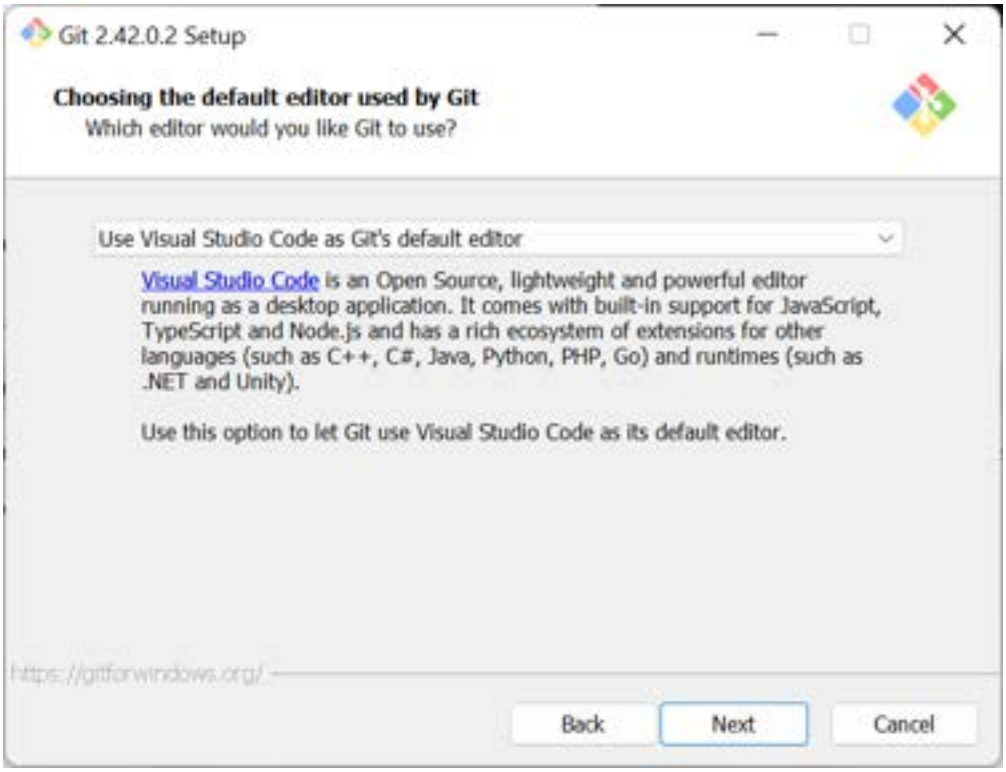
The next screen is where the installation is customized. It is safe to choose the defaults here, but you might also want to add an Icon to your desktop and configure a daily check for updates (reflected in the next image). When you have made your selections, select Next to continue.



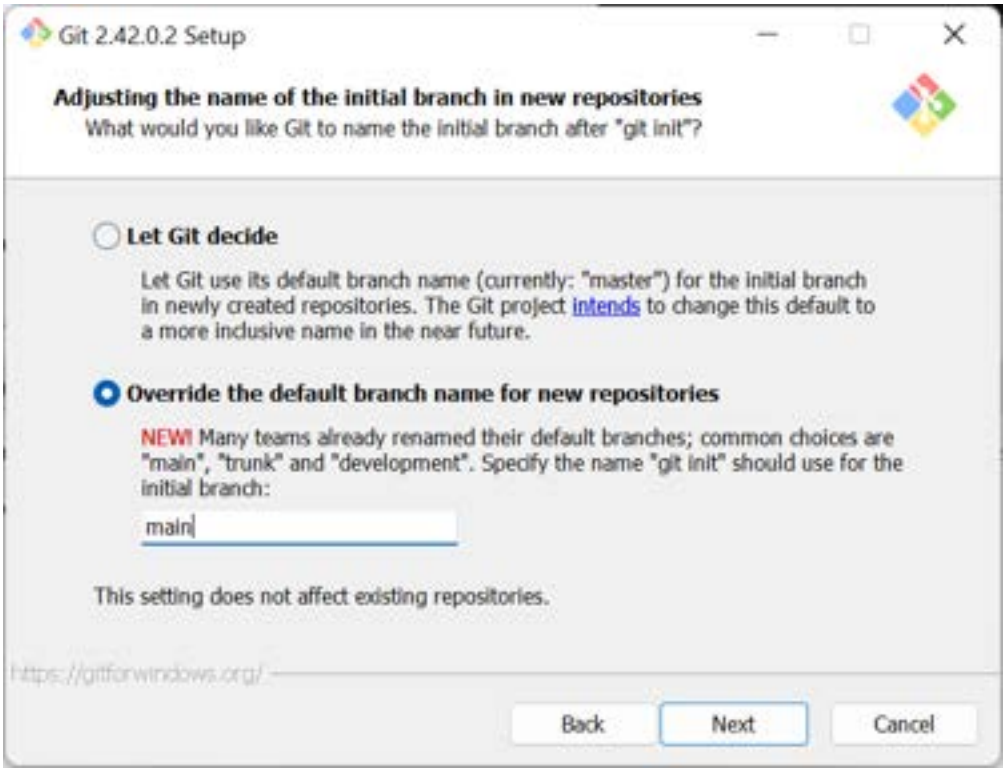
The next screen allows you to select the folder for your Start Menu where shortcuts for Git will be located. I have selected the default here. Select Next to continue.



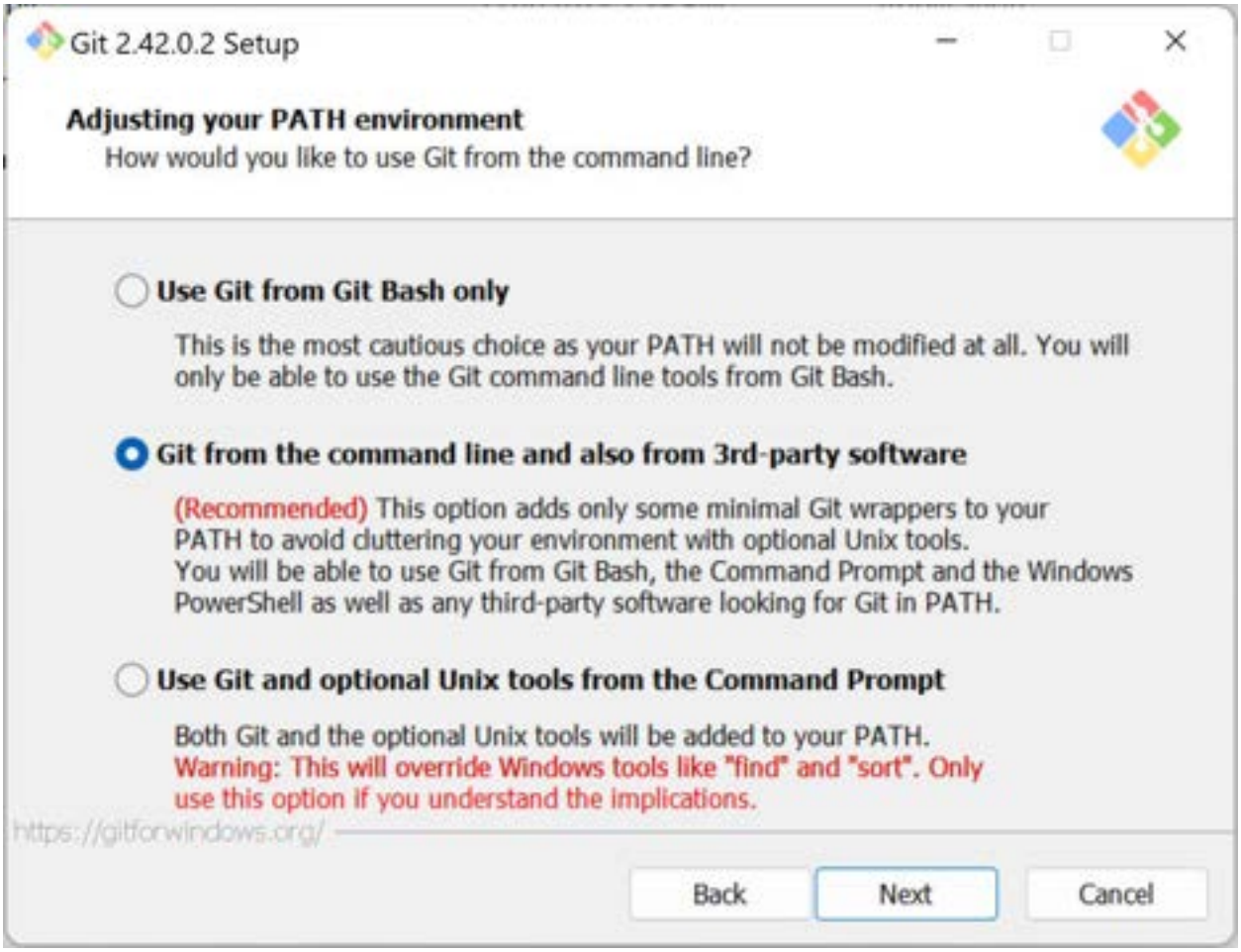
The next screen allows you to select the default editor to use with Git. The default for this selection is vim (VI Improved) which makes a lot of sense for people with a Unix or Linux background. However, due to the course focusing on utilizing Visual Studio Code (VS Code) as an Integrated Development Environment, it may make sense to select VS Code instead. Make your selection and choose Next to continue.



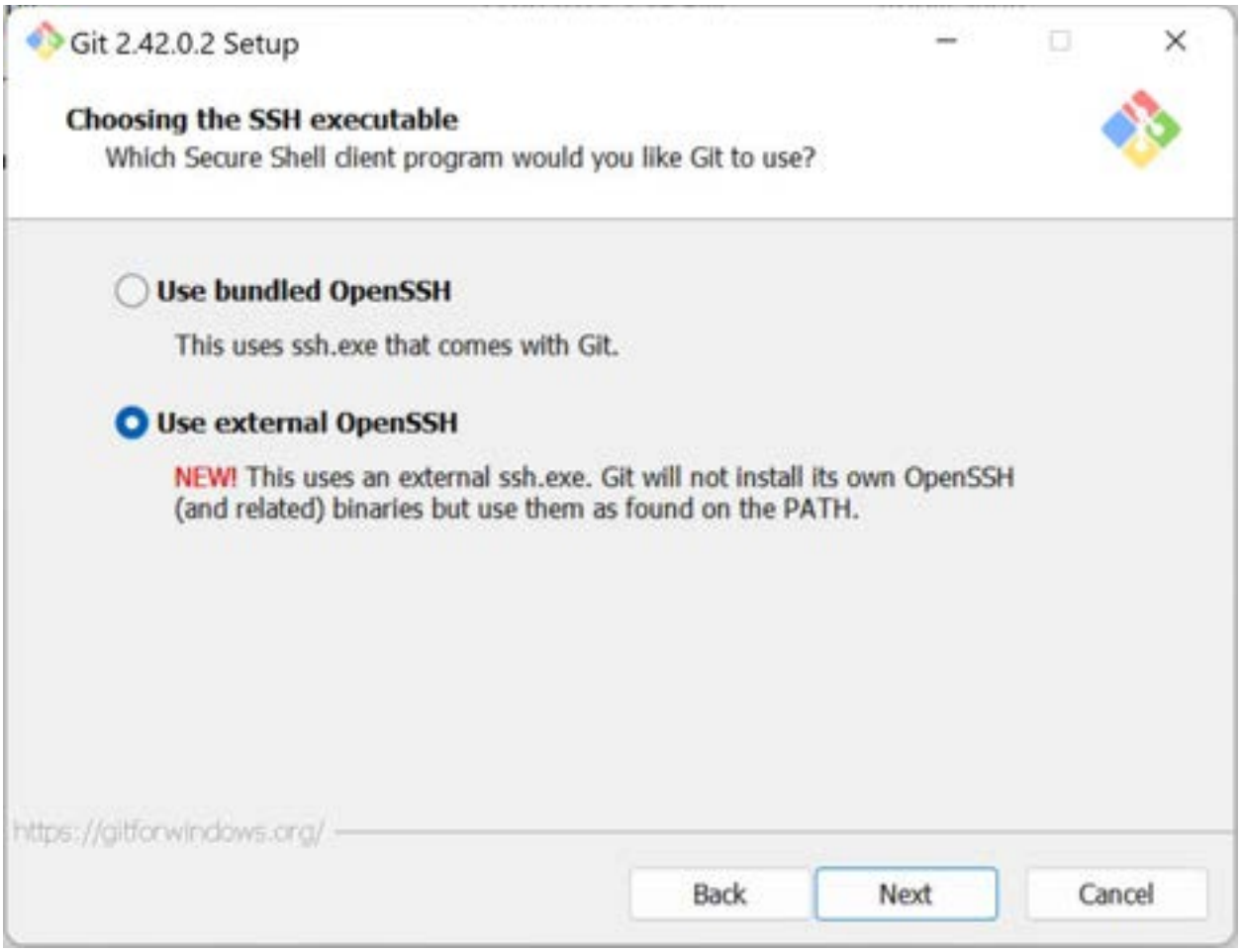
The next screen allows you to select the default branch name for a new git repository. The industry standard for this has most recently changed from master to main which is how we will configure it here. Select Next to continue.



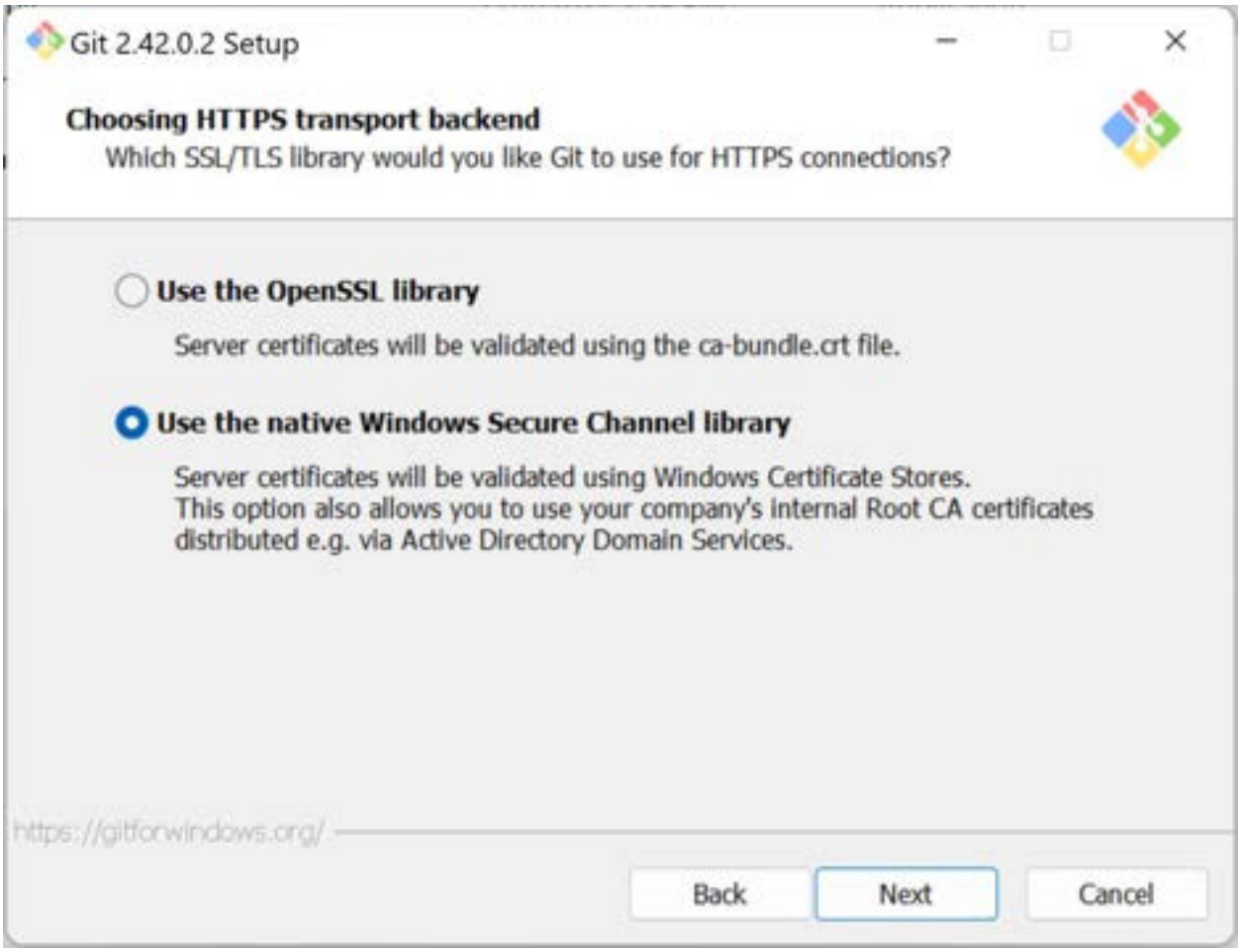
The next screen allows you to determine how Git will integrate with the rest of your environment. While there are three choices you need to select either the recommended solution to make git available to 3rd Party software, or the selection that will install additional Unix tools for the command prompt (this can be very useful if you have a Linux or Unix background and are working in a Windows environment). This guide is based on taking the recommended choice here. Select Next to continue.



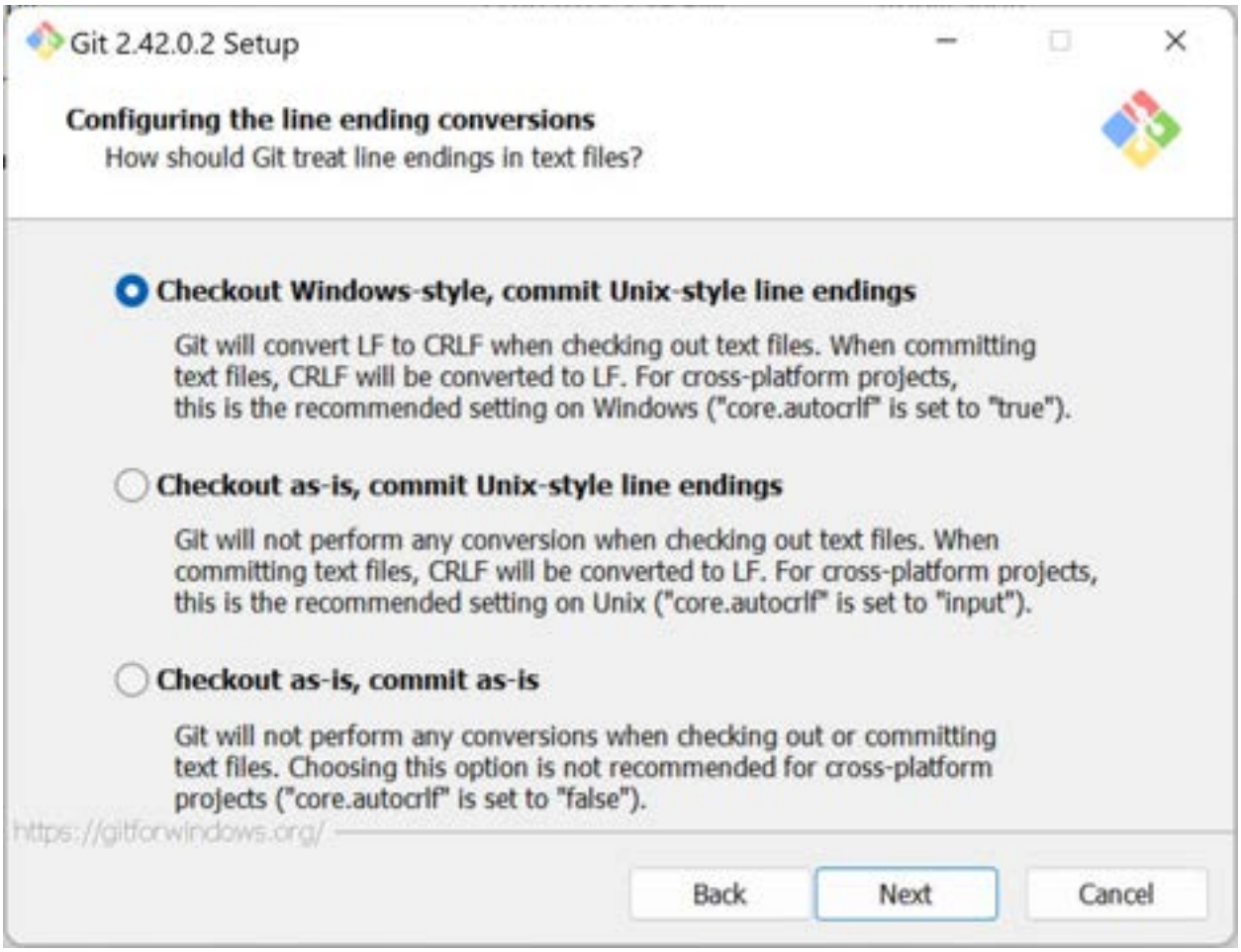
The next screen asks whether to use the bundled OpenSSH client, or to use and external OpenSSH application. If you are running a current build of Windows 10 or Windows 11, the operating system provides an ssh.exe that should be used here (the OS will be responsible for keeping it patched and up-to-date). If you type ssh at a command prompt and the command isn't found, select 'Use Bundled OpenSSH'. Select Next to continue.



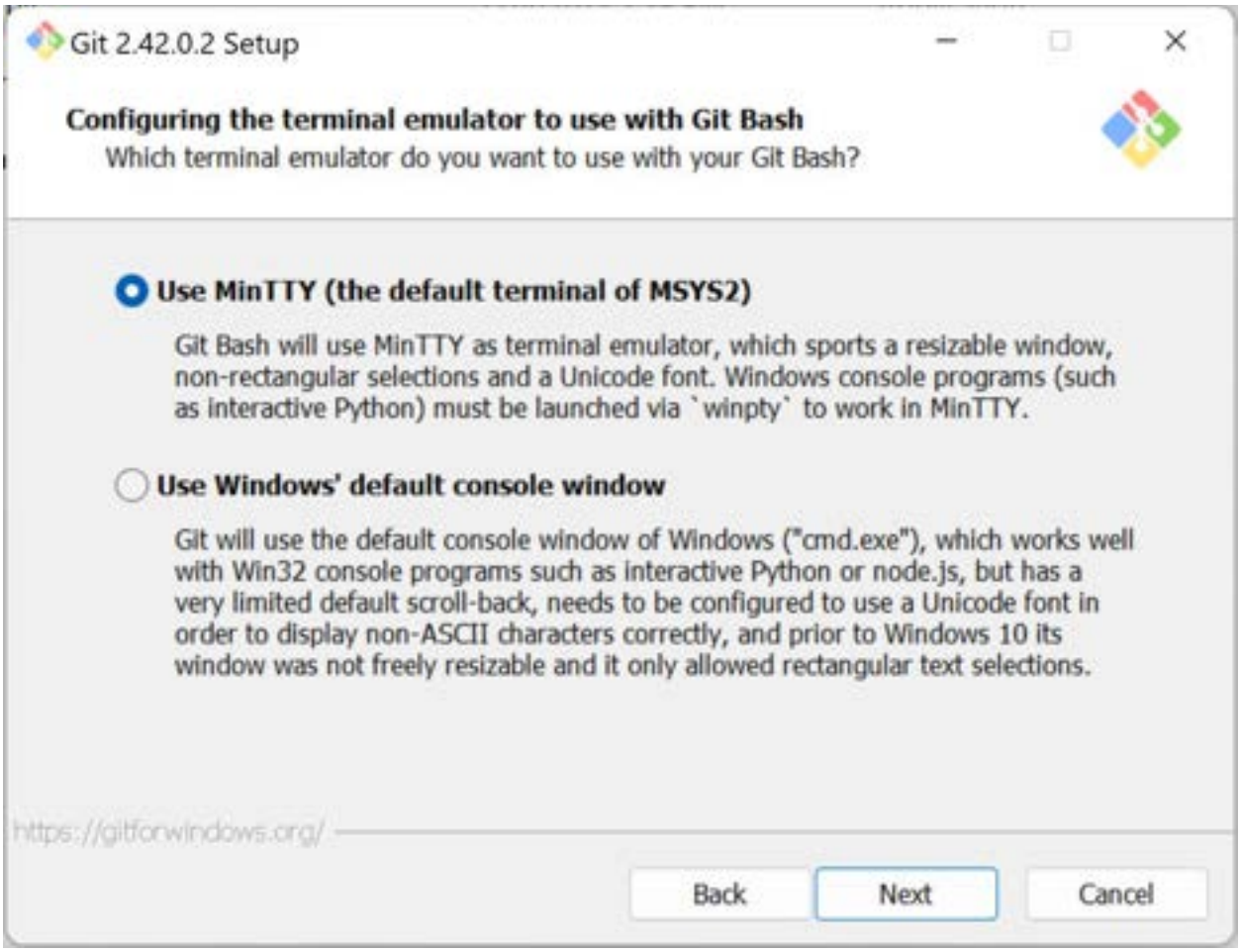
The next screen requires you to make a choice as to how Git will make secure connections. The two options are using the OpenSSL library or using the native Windows methods. Either choice is reasonable (especially for this class). If you are going to use the software in a corporate setting, choose the Windows Secure Channel Library because it will be updated as necessary via Windows update. Select Next to continue.



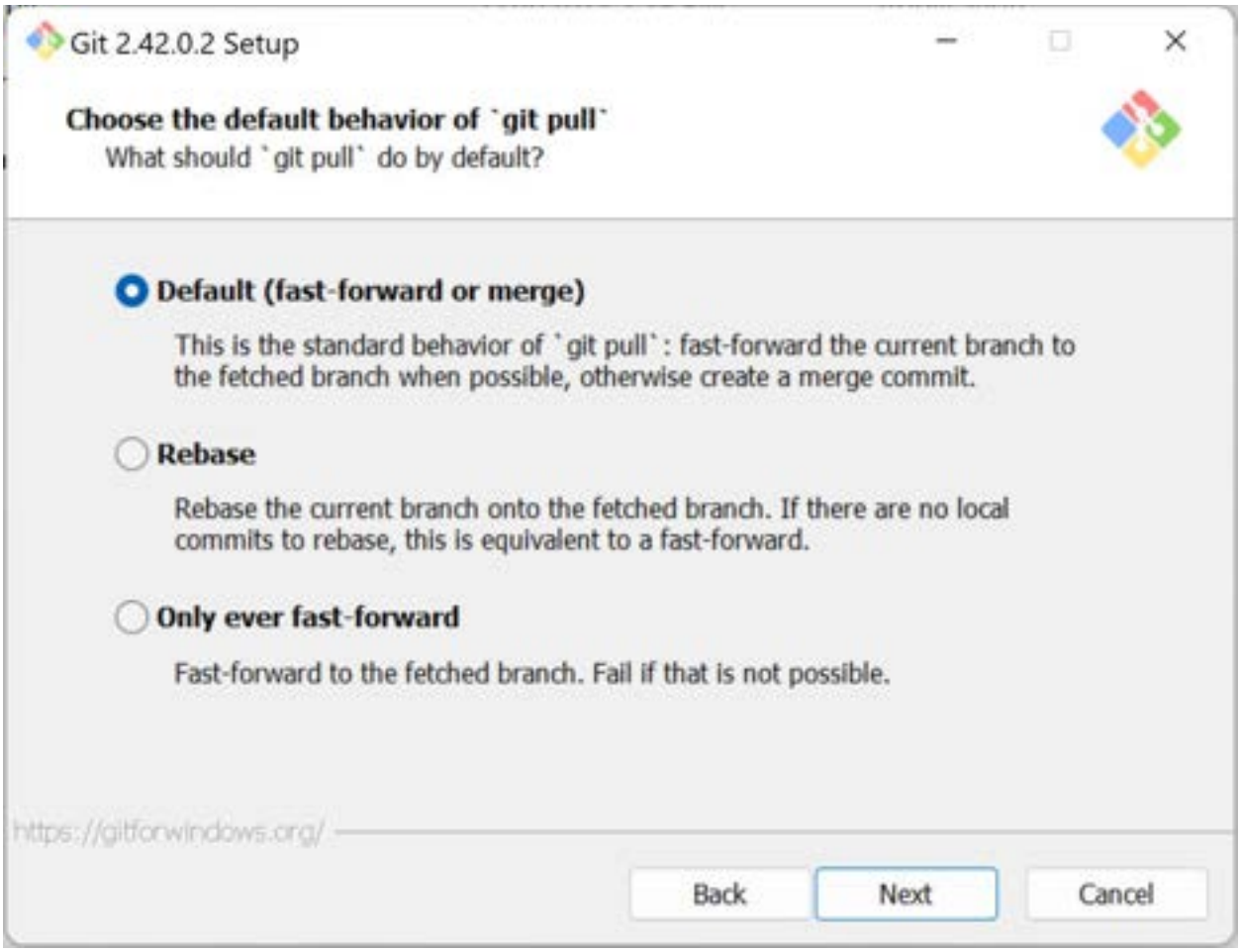
The next screen discusses how to treat the end of line characters in your text files. This is important as Windows and Unix/Linux text files differ. Windows uses a Carriage Return and a Line Feed (CRLF), whereas Unix/Linux just uses a Line Feed (LF). The default is safe to accept here. Select Next to continue.



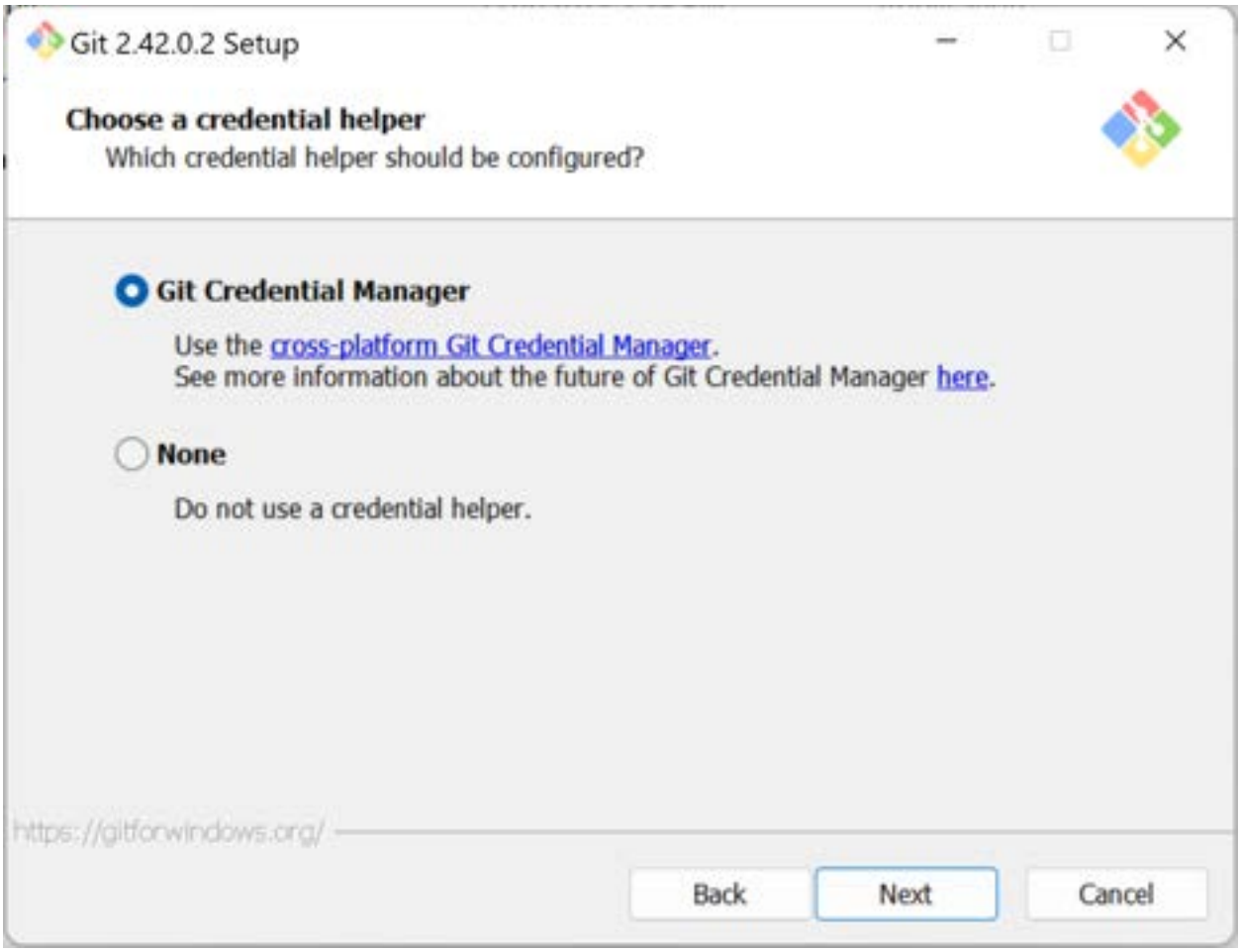
The next screen allows the configuration of the terminal that will be used with the Git shell. Git provides MinTTY or you could use the Windows default console window. Selecting the default here generally works well. Select Next to continue.



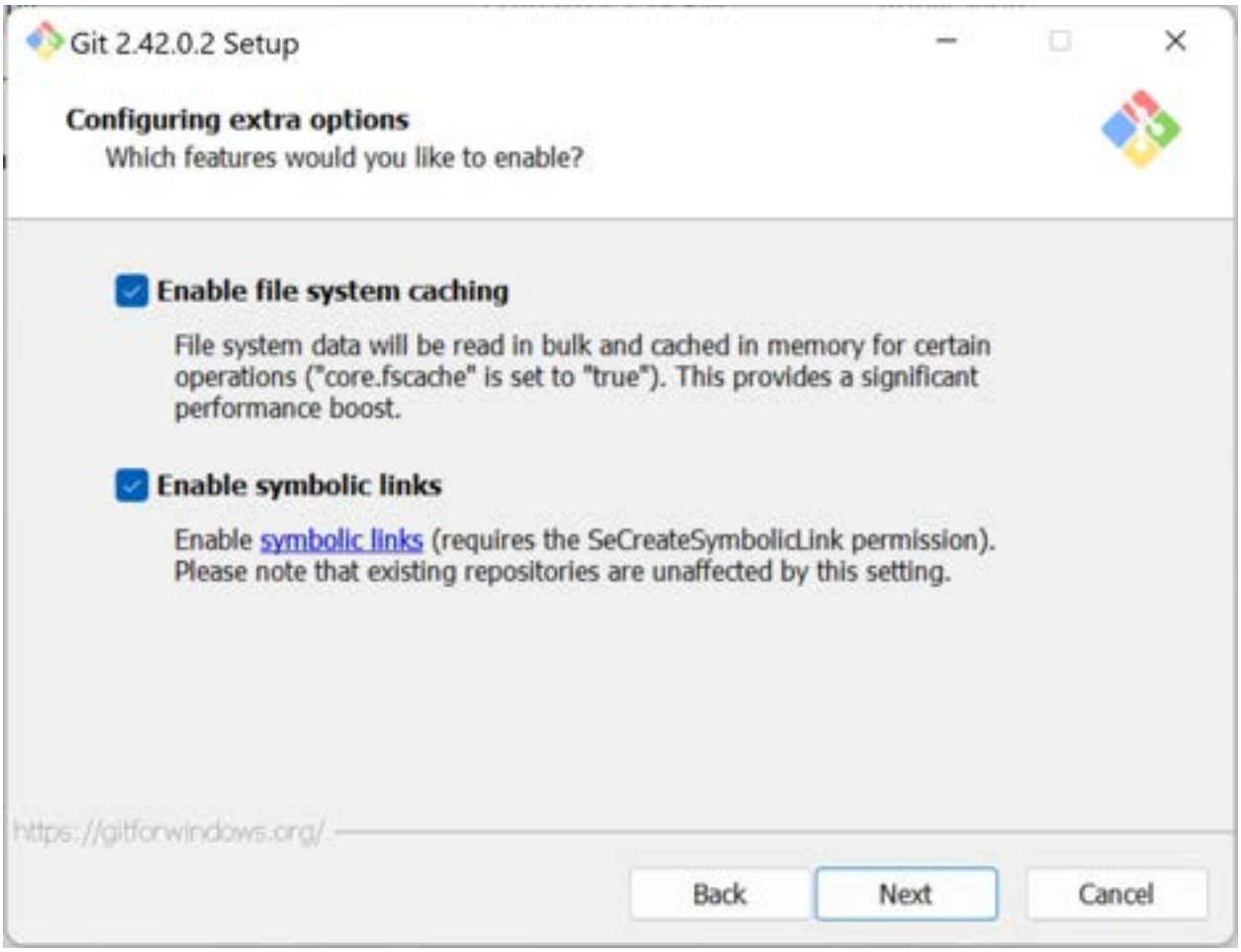
The next screen selects the default behavior for the ‘git pull’ command. The default is to perform a fast-forward (or merge). This is the best selection here. Select Next to continue.



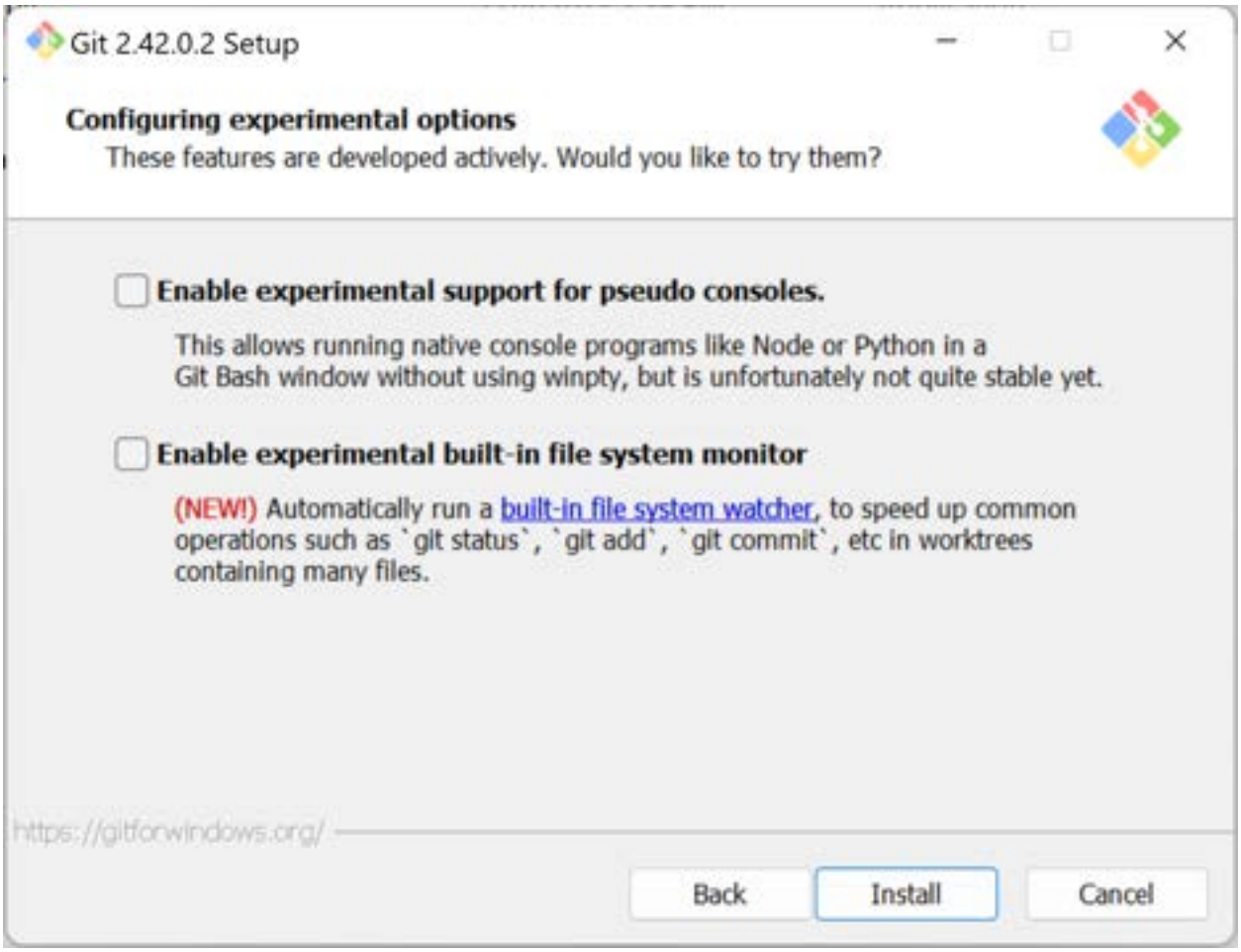
In the next screen you can select a credential helper. This can be important to manage access credentials and tokens. There is only one choice, select the default. Select Next to continue.



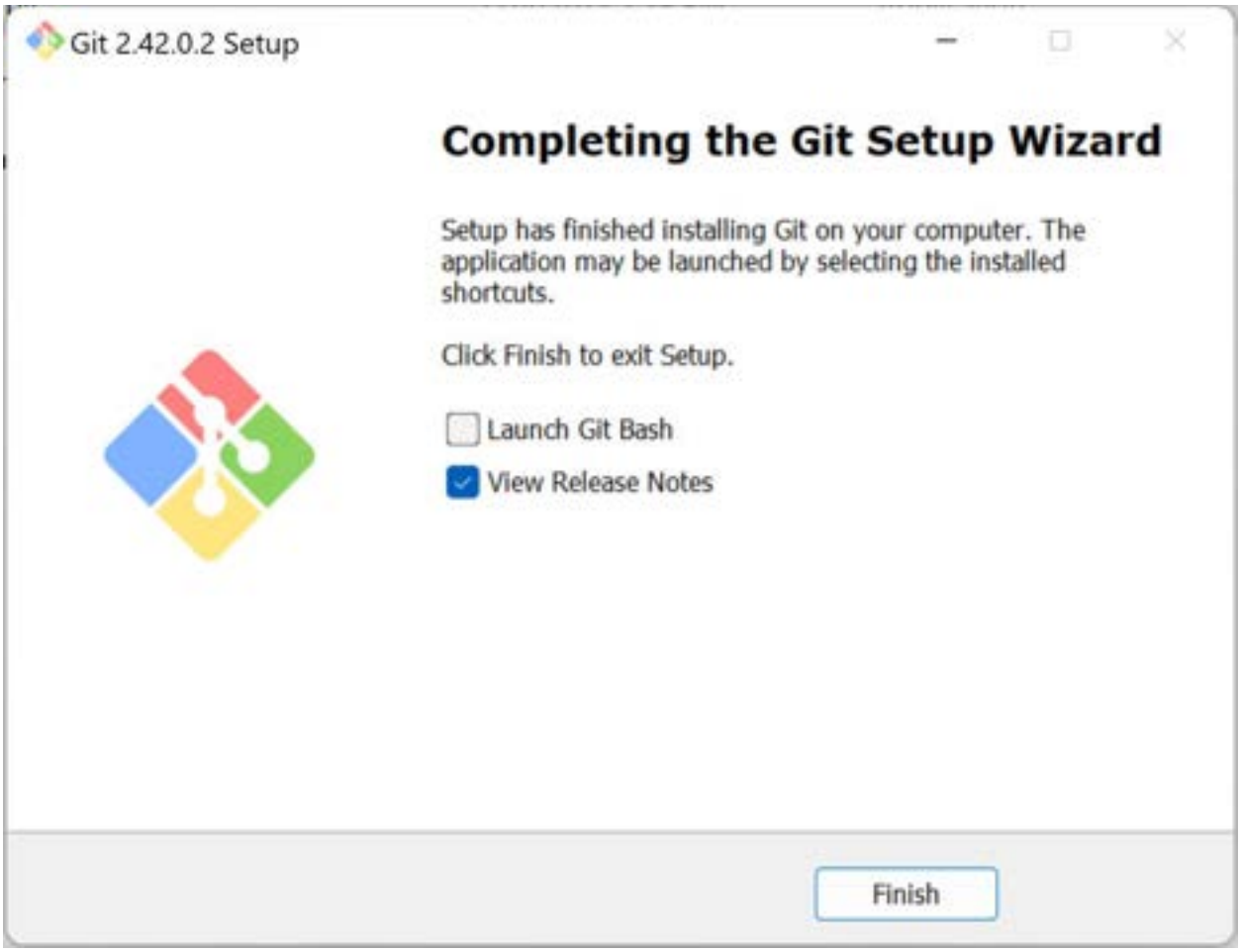
The next screen provides the option to install two additional capabilities that enhance the performance of Git. Enable both of these. Select Next to continue.



The next screen provides choices for experimental options. Do ***NOT*** select either of these options. Select Install to continue.

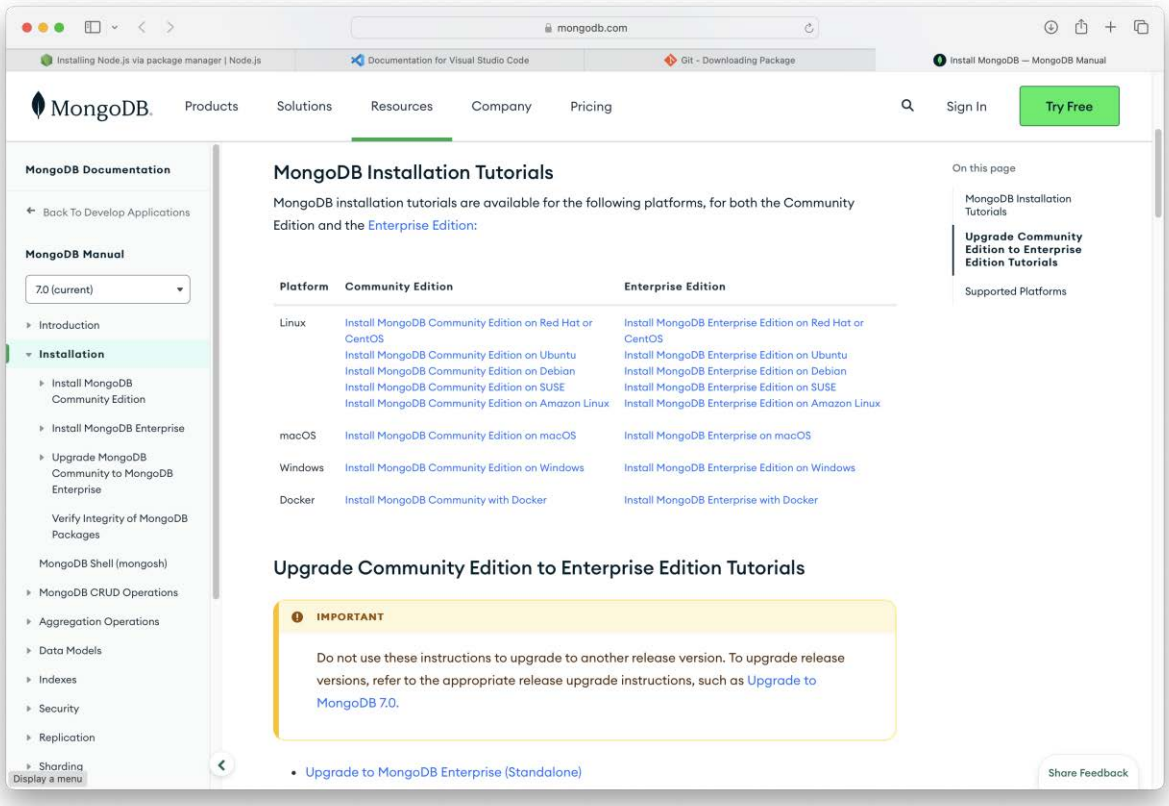


The final screen provides the opportunity to review the release notes for this version of the software. Select Finish to complete the installation process.

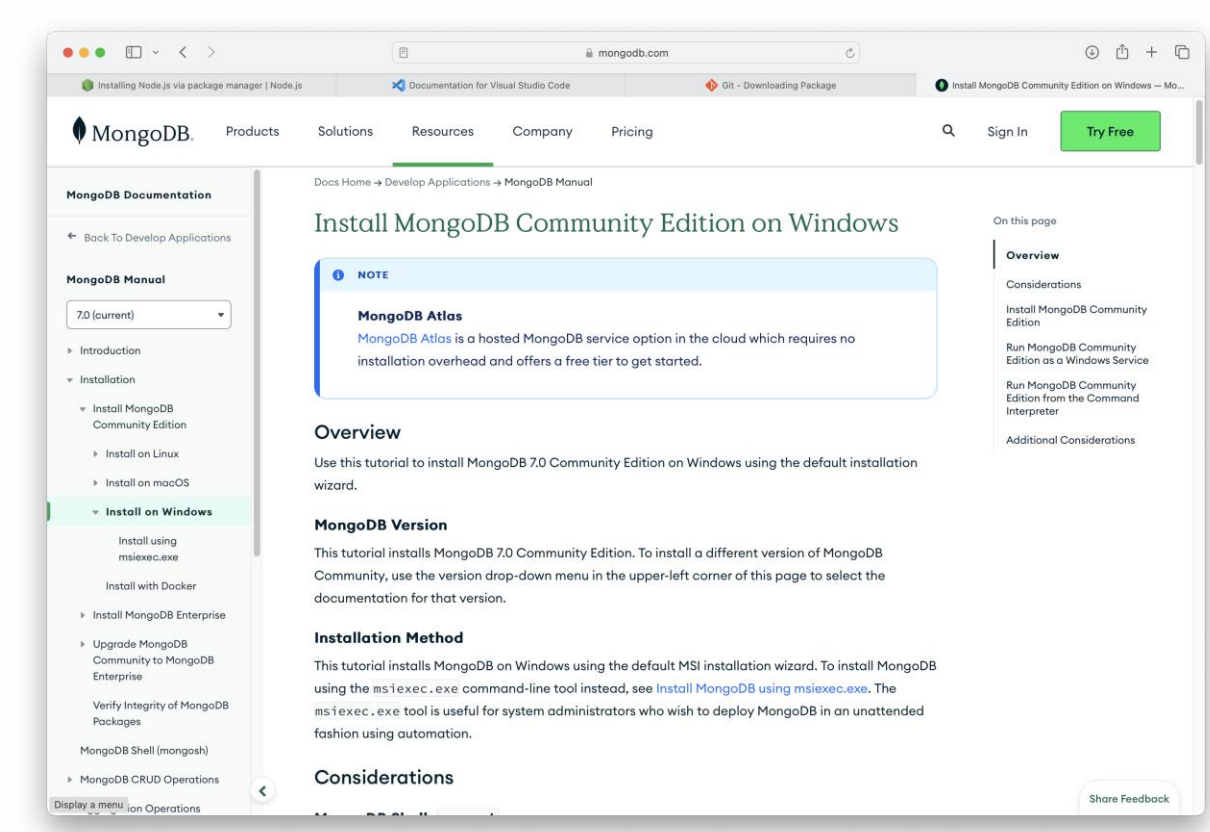

Git 2.42.0.2 Setup
Completing the Git Setup Wizard
Setup has finished installing Git on your computer. The application may be launched by selecting the installed shortcuts.
Click Finish to exit Setup.
Launch Git Bash
View Release Notes
Finish

Install MongoDB

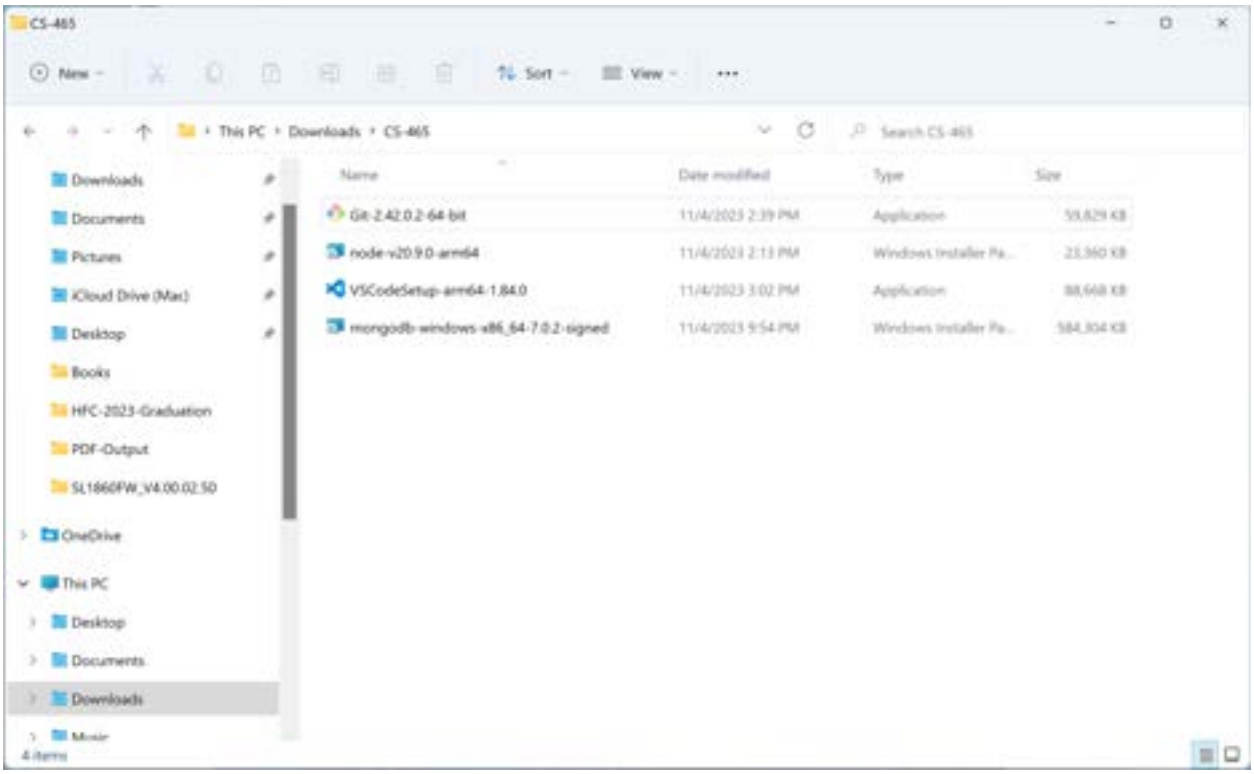
The next step in installing the environment is installing MongoDB. This is the database that we will be using for the project in this class. You can find MongoDB for a variety of different operating systems and architectures here: https://www.mongodb.com/docs/manual/installation/



We will be installing the Community-Edition on an instance of Windows 11, so we will download the Windows .msi package. You can find that directly here: https://www.mongodb.com/try/download/community

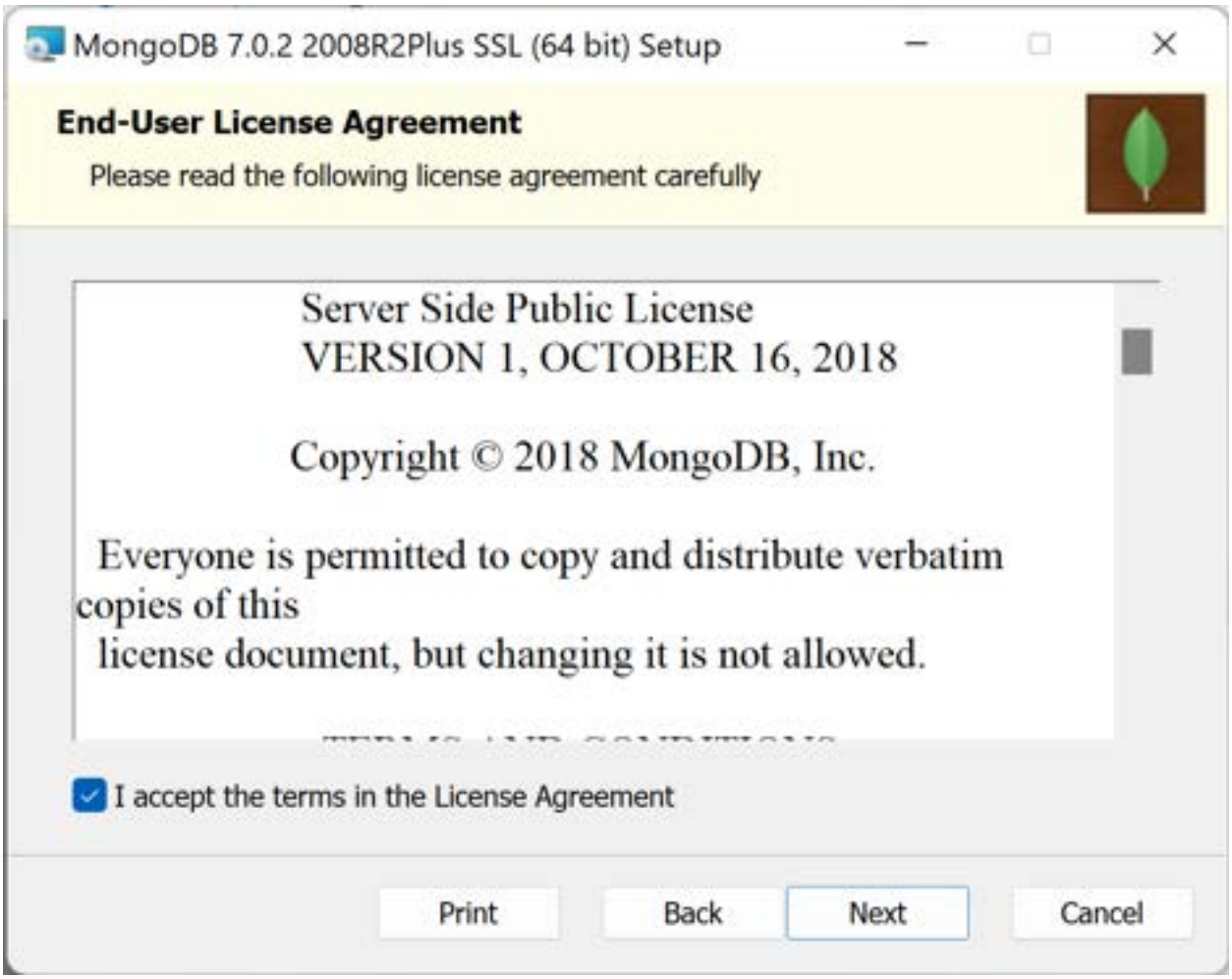


When you select the appropriate download it will be dropped into your default download directory.

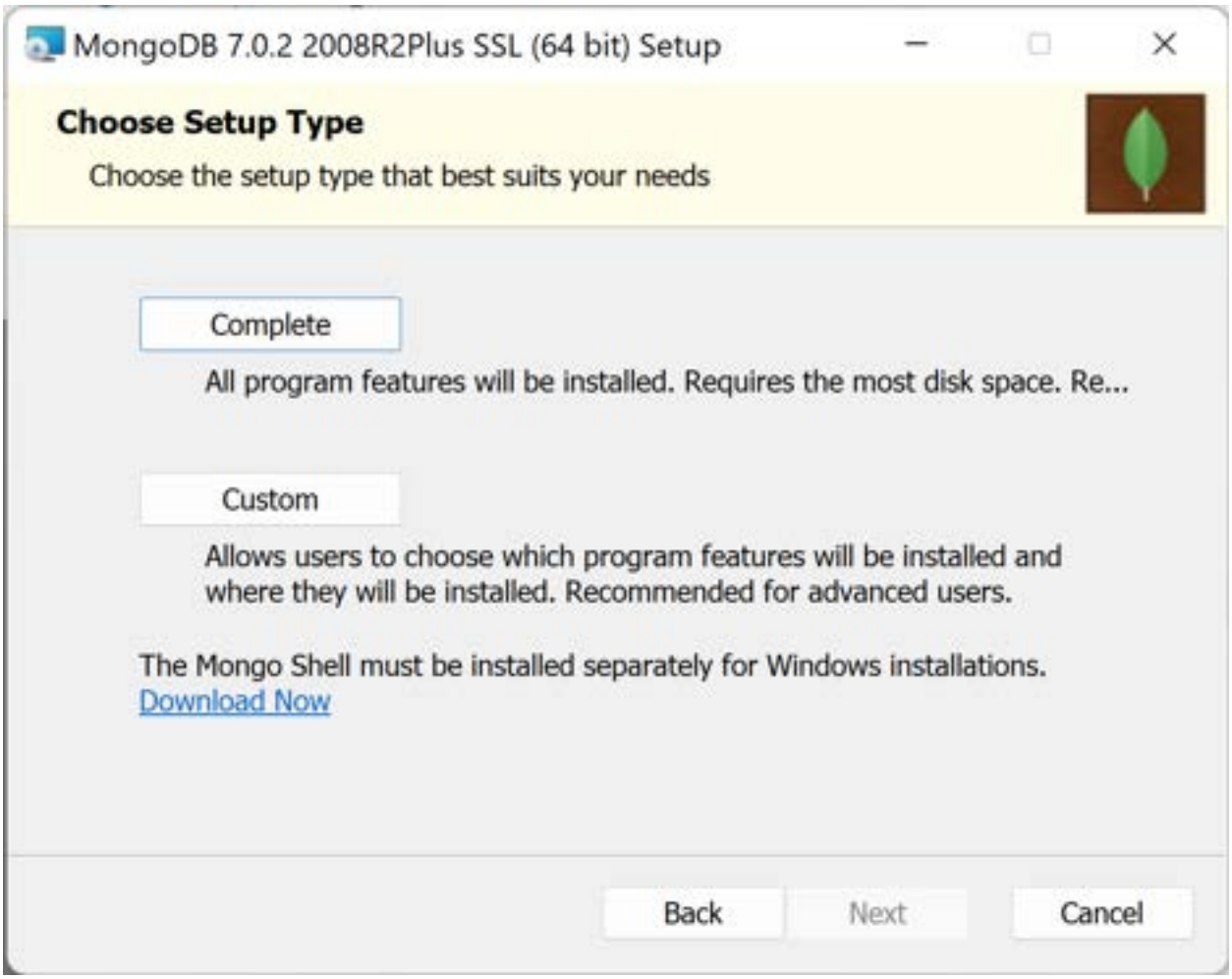


Double-click on the installer to begin the installation process. Select Next to continue.

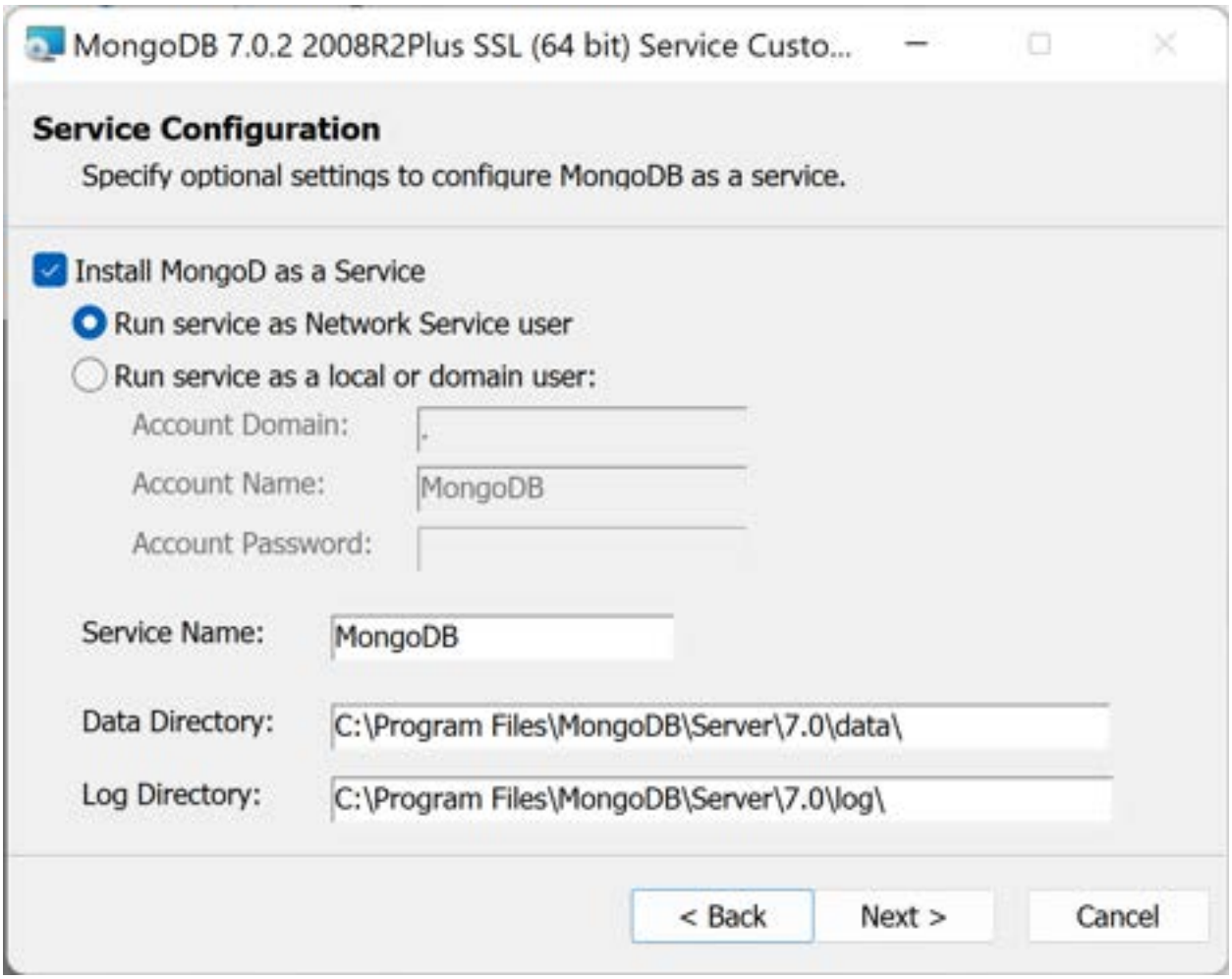
The next window presents the License Agreement for you to read and accept the EULA to continue the installation. Select Next to continue.



The next screen identifies the components of MongoDB that you will be installing. For this example, we will be selecting the complete installation.

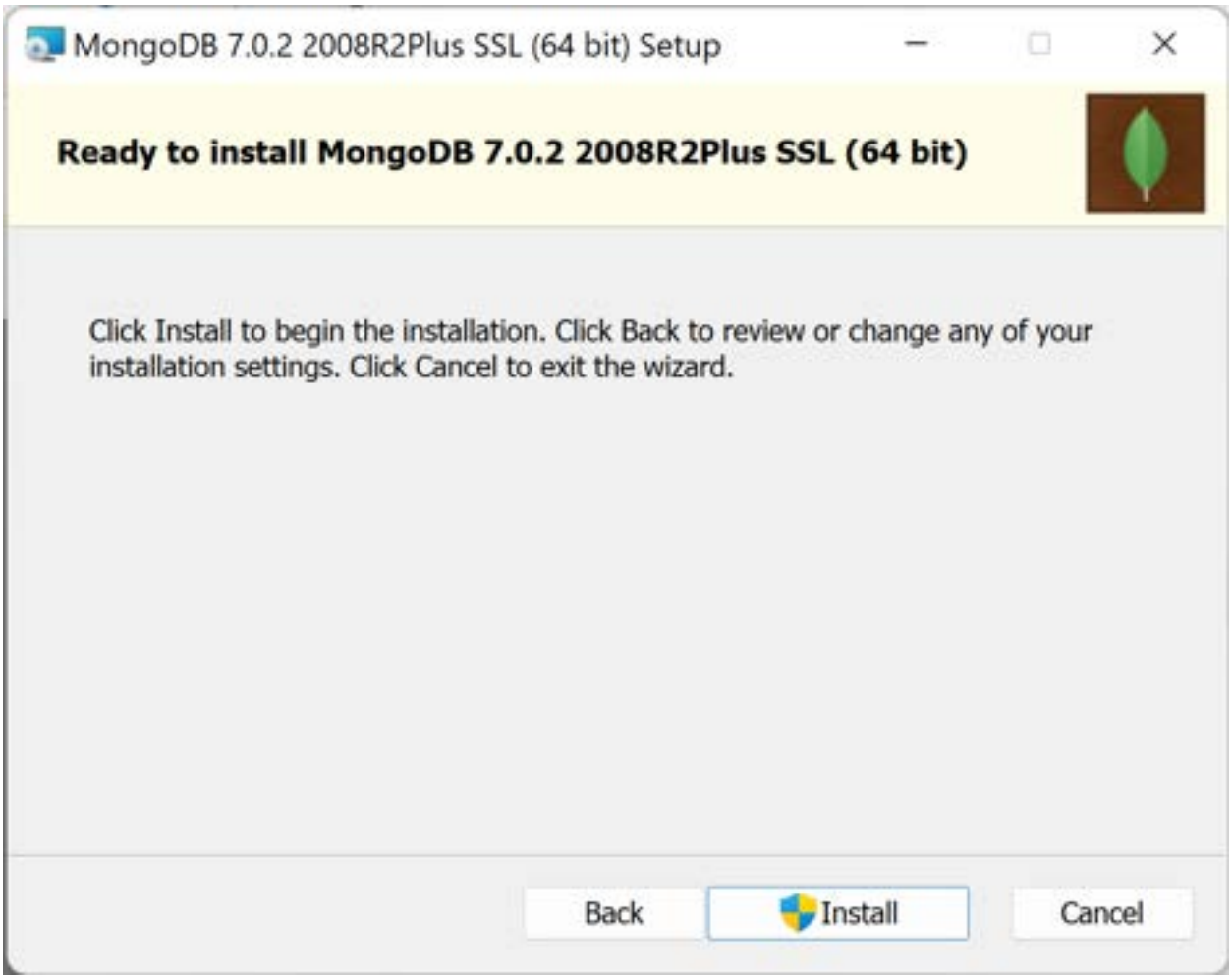


The next screen shows the configuration panel that will allow you to configure the service that will run MongoDB. For this installation, we will be taking the defaults and installing it as a Network Service. Select Next to continue.

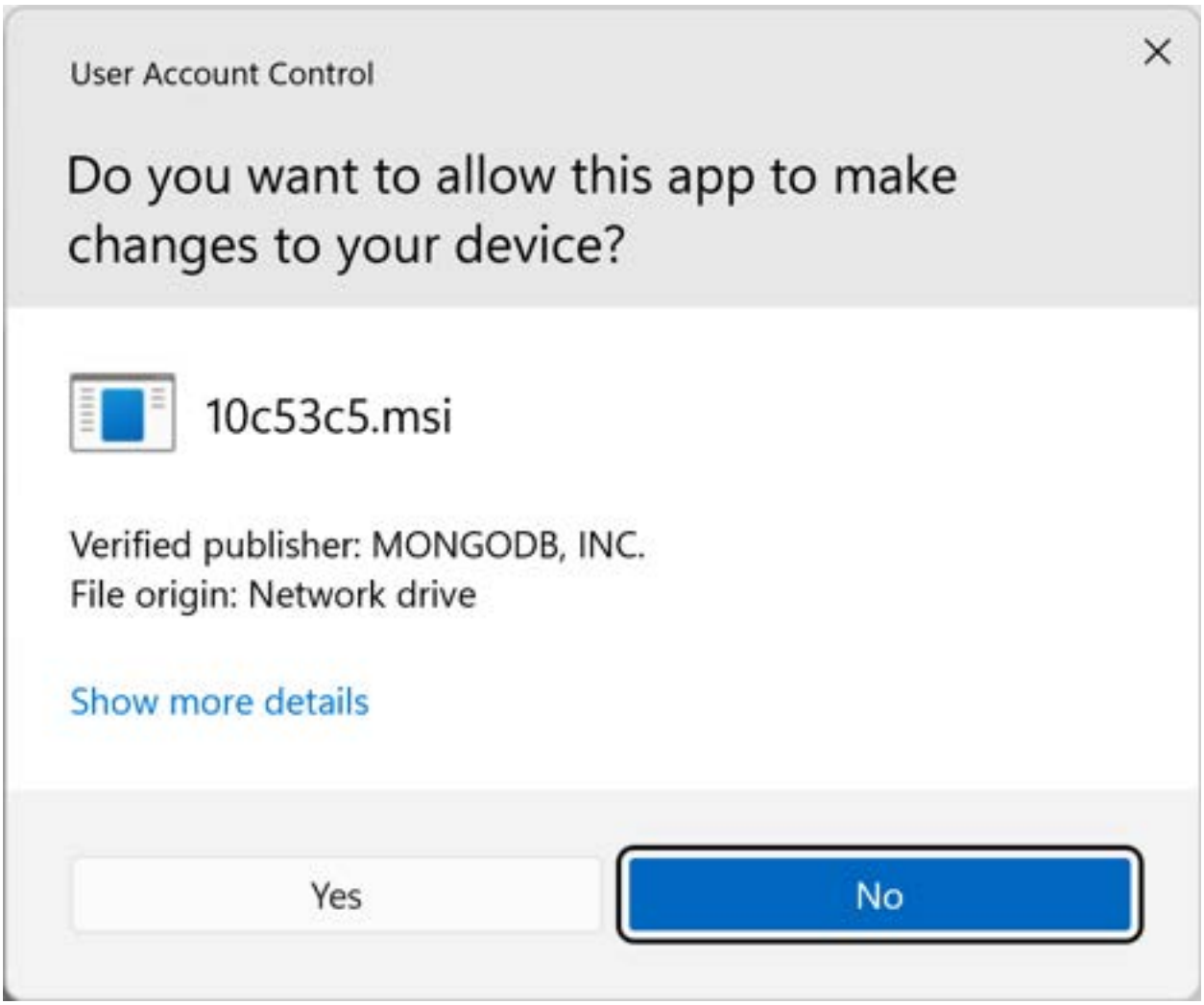


The next screen offers the opportunity to install MongoDB Compass – a graphical interface that will allow you to interact directly with your MongoDB installation. We will install this and use it to develop our application. Select Next to continue.

The next screen allows you to proceed with the installation. Select Install to continue.

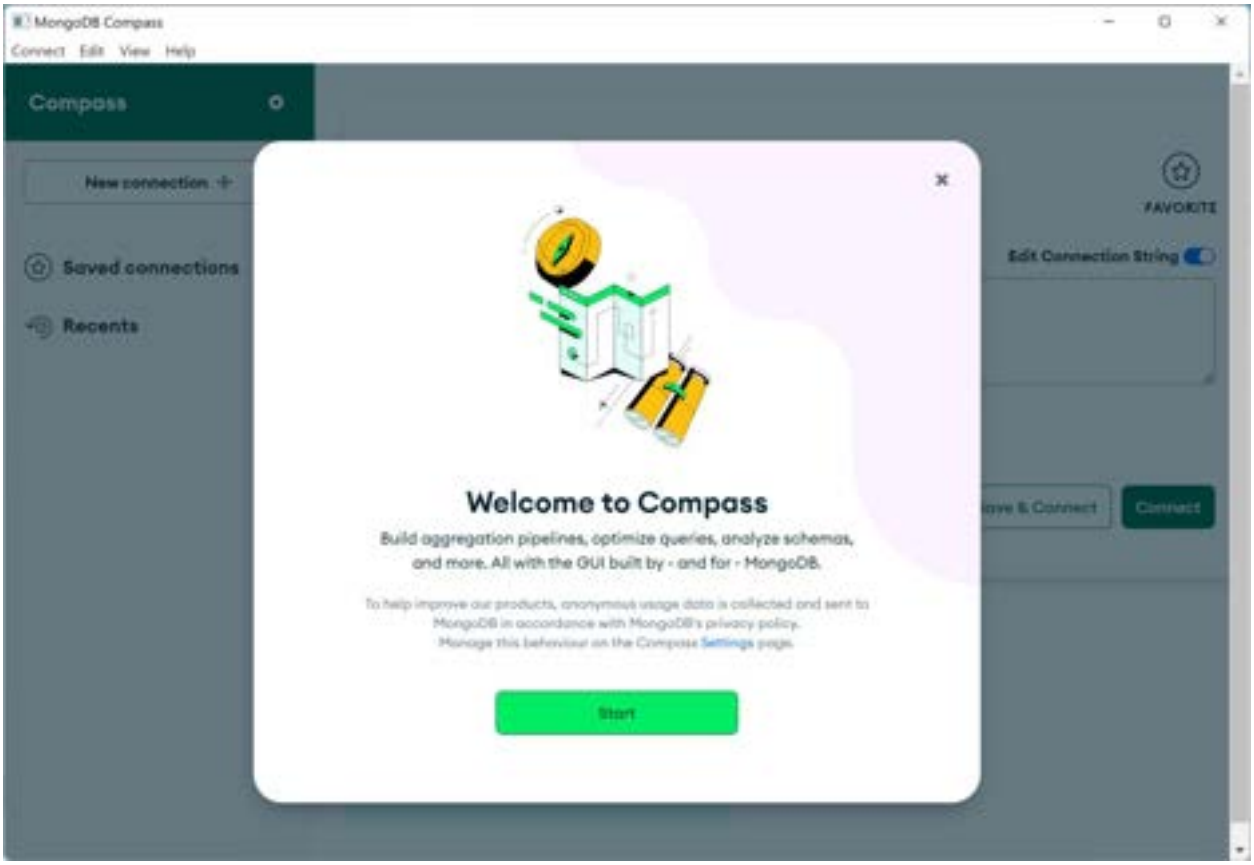


The system will prompt you to acknowledge that the installer needs to make changes to your environment. Select Yes to continue.

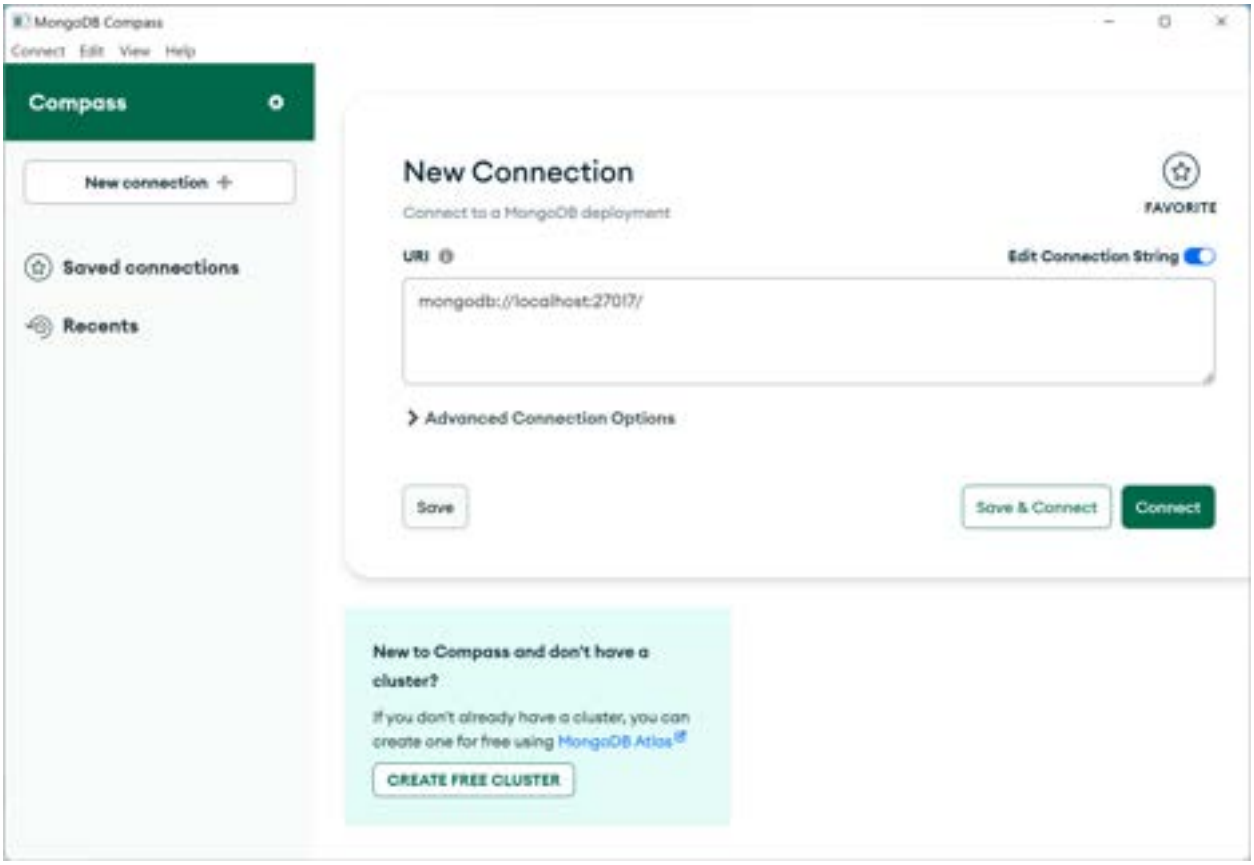


The final screen will show that the software has been successfully installed. Select Finish to continue.

The installer will automatically start MongoDB compass. Select Start to continue.



The next frame shows you the connection pane to connect to your new MongoDB database. You can select connect from here to connect to the MongoDB instance.



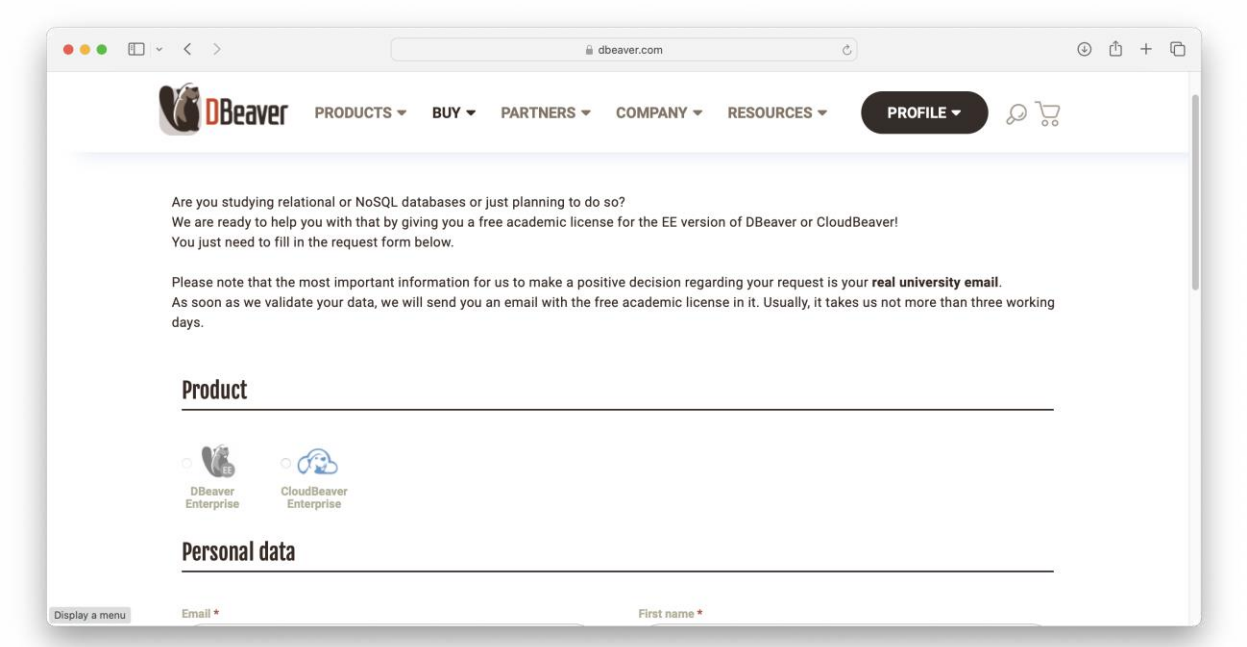
The following screen indicates a successful connection to your MongoDB database and we can use this later on to verify, validate, and manipulate the records in your Database.

Installing DBeaver

DBeaver is an optional tool for examining data in the Mongo Database that we will be using in this course and it is added here because Mongo Compass is a Windows only tool, whereas DBeaver is multi-platform. DBeaver is an application that allows you to interact with many different Databases to Extract, Transform, and Load (ETL) data in a database. This tool is available from DBeaver at https://dbeaver.com/academic-license/ . It is important that when you register for an account with the company you use your SNHU e-mail address. This is how they determine whether you will qualify for an Academic (free) license.

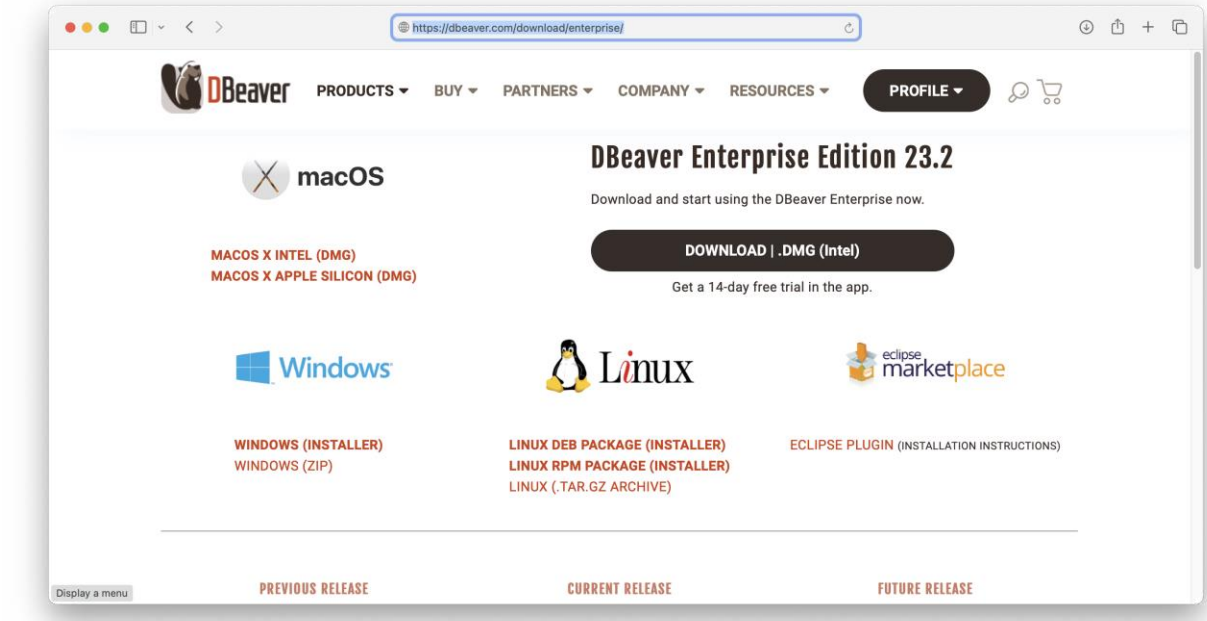
As this guide has already covered the installation of Mongo Compass on Windows, the following will walk through the installation of DBeaver on macOS as an option for those using a Mac for this course.

Once you have registered for an Account with DBeaver at the above link, you can request an Academic license. When you fill out the form, select DBeaver Enterprise, enter your Name, your SNHU e-mail address and indicate that you want to use the software for this course:



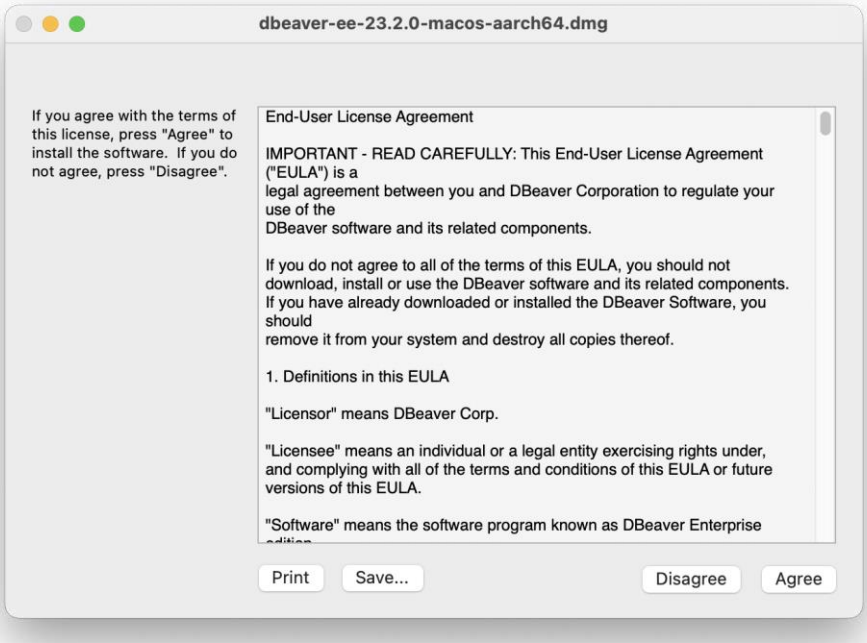
When your request has been successfully processed, you will receive an e-mail that indicates you have been granted a one year license for DBeaver Enterprise. You can access the license through the link provided in e-mail or through the Profile button on their website.

When you have been granted the license, you can download the software from the download page (Products -> Download -> DBeaver Enterprise) on their web page.

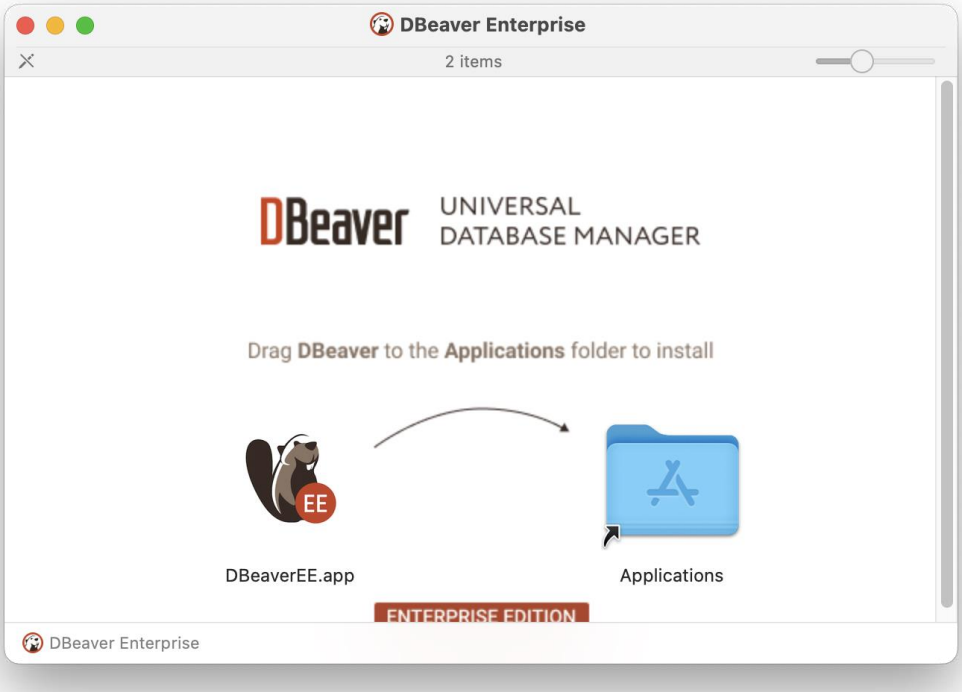


Select the appropriate download and continue. For the purposes of this guide, we will be working with an install on macOS X.

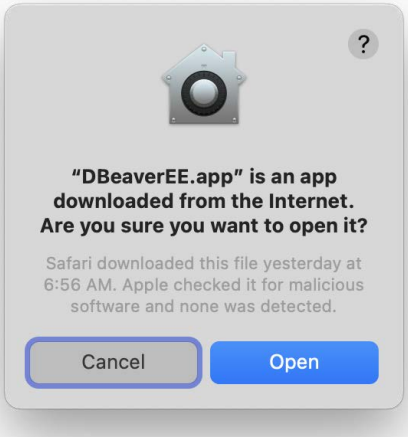
Double-click on the package to start the installation process. The first step will be to read and accept the End-User License Agreement (EULA). Click ‘Agree’ to continue.



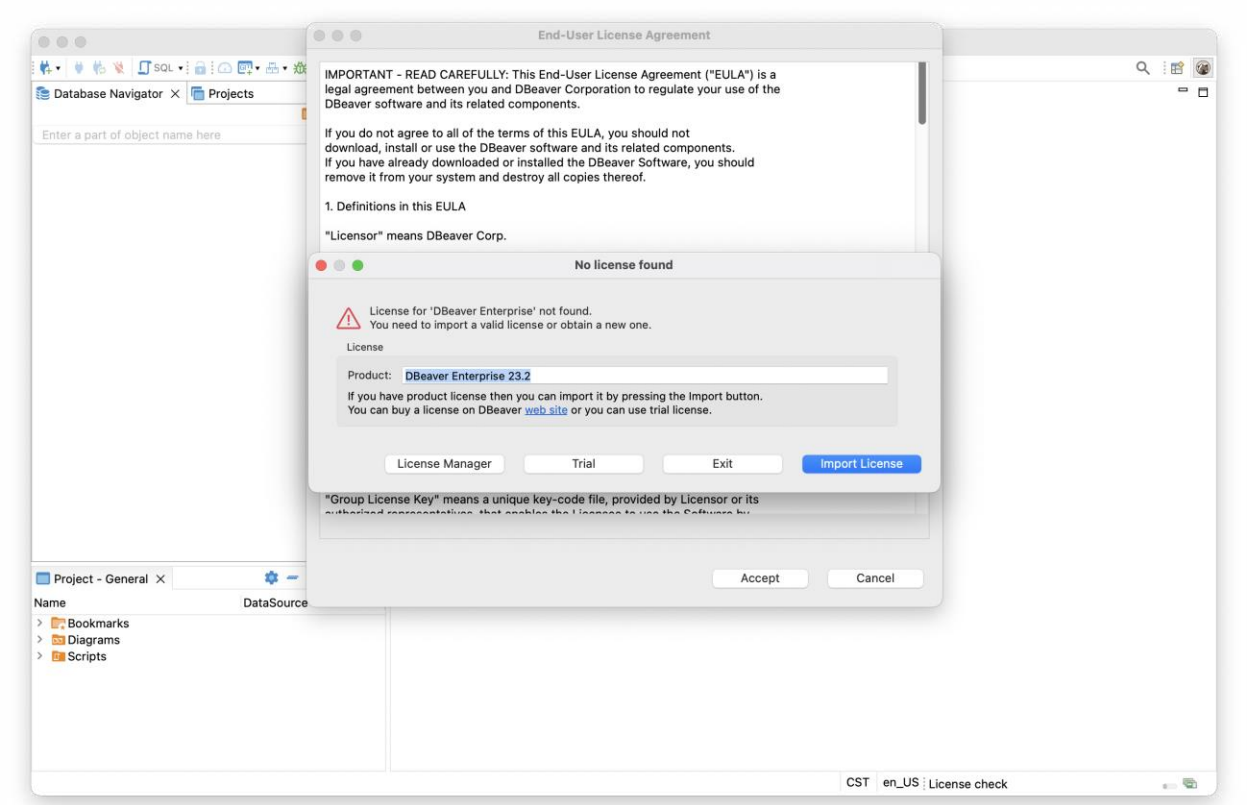
Like many macOS applications, the installation process is quite simple. Drag the App into the Applications Folder Link:



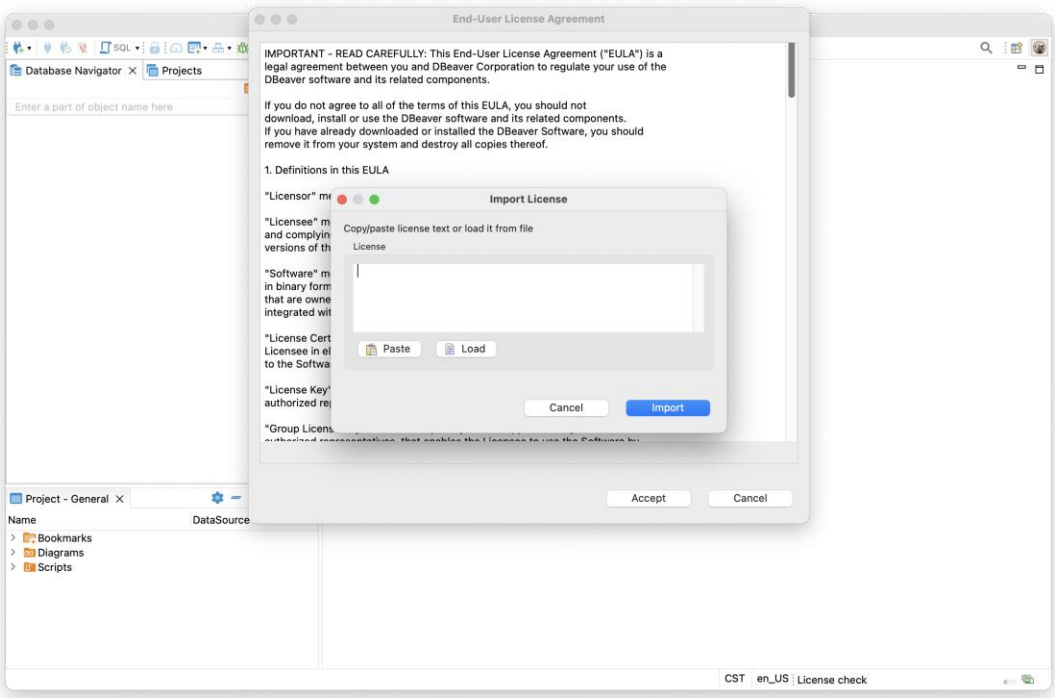
You can now access DBeaverEE in your applications folder. Double-Click to start the software. You will have to acknowledge that the application has been downloaded from the Internet. Select 'Open' to continue:



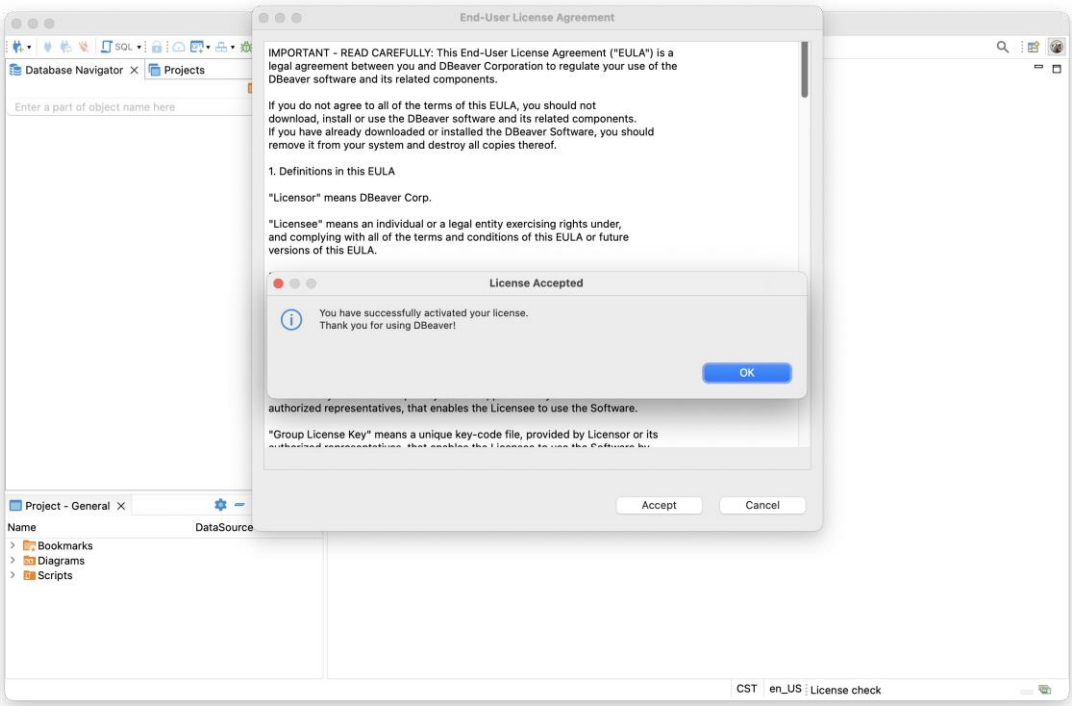
The first time you run the application, it will not detect the license. You will have to add the license at this point so that the application will be fully functional. Select ‘Import License’ to continue.



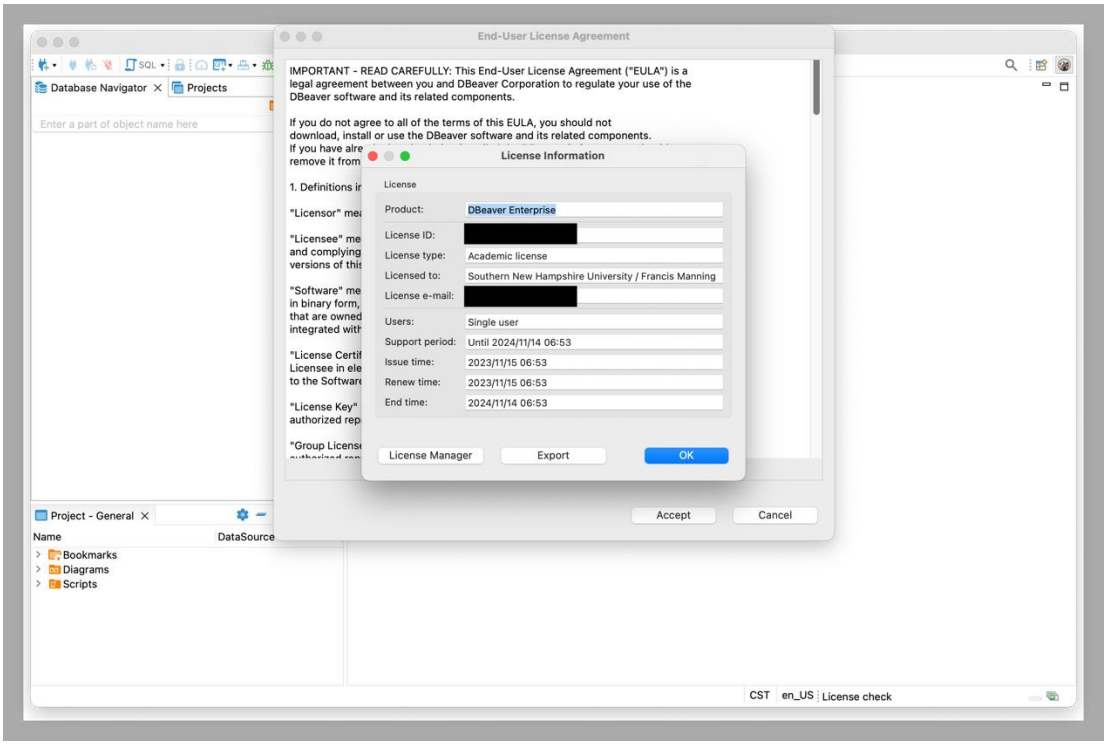
You can now copy/paste the license (you received instructions on how to retrieve the license in an e-mail from DBeaver) into the box for the license key. Select ‘Import’ to continue:



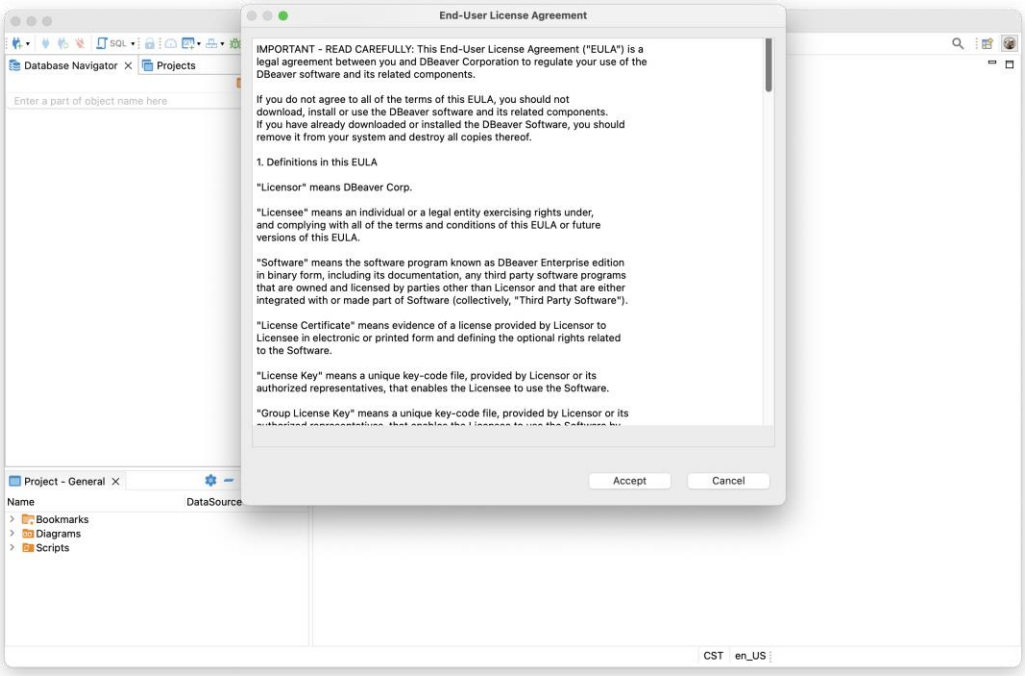
Once you have imported your license, you should receive a message that you are now running a licensed copy of the software. Press ‘Ok’ to continue.



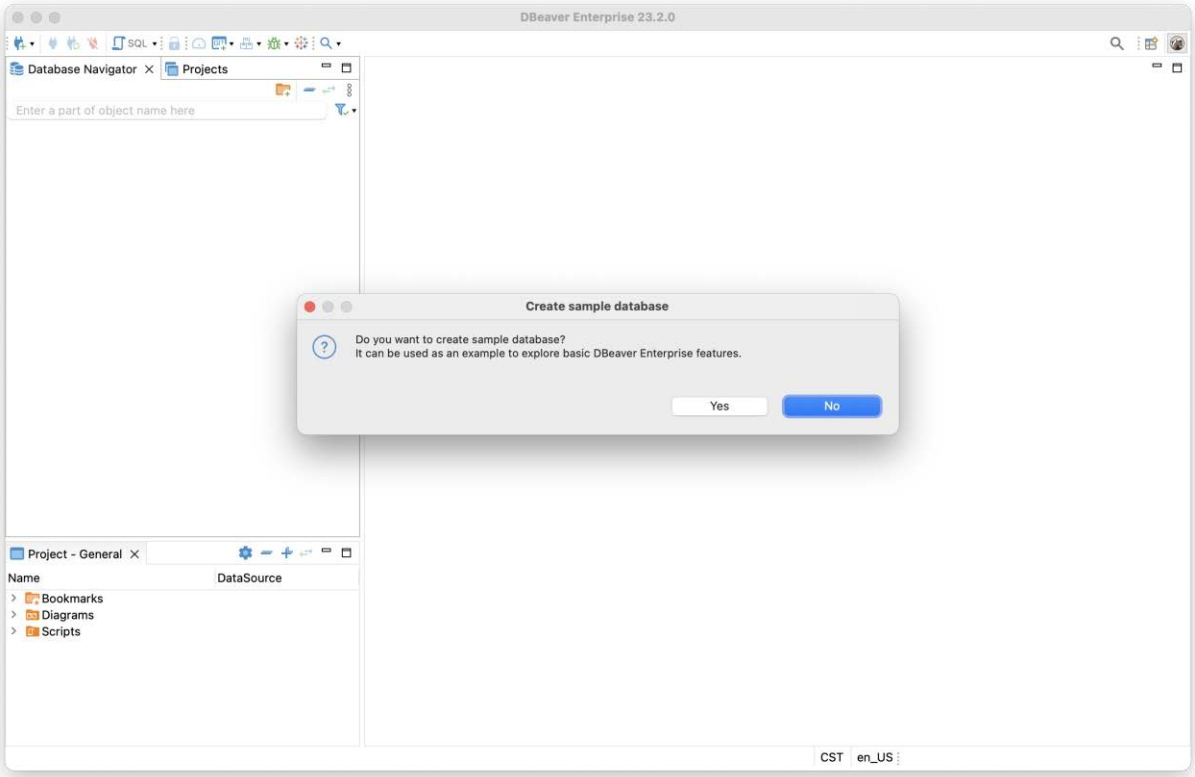
You will now be brought to a screen that summarizes your license. Select ‘Ok’ to continue.



You will have to read the EULA and accept it one more time to begin using the software. Select 'Accept' to continue:



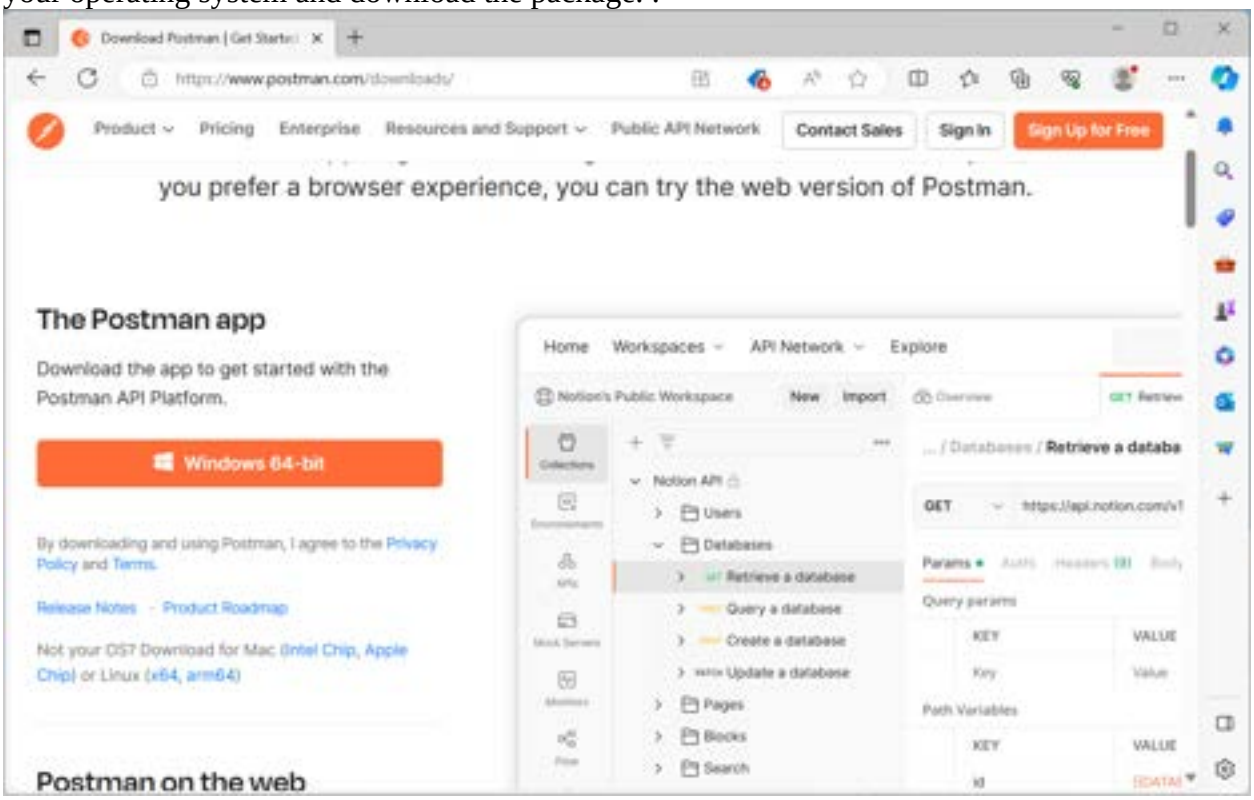
At this point you can choose to create a sample Database to explore some of the features of this application.



You are now ready to use this tool with your database!

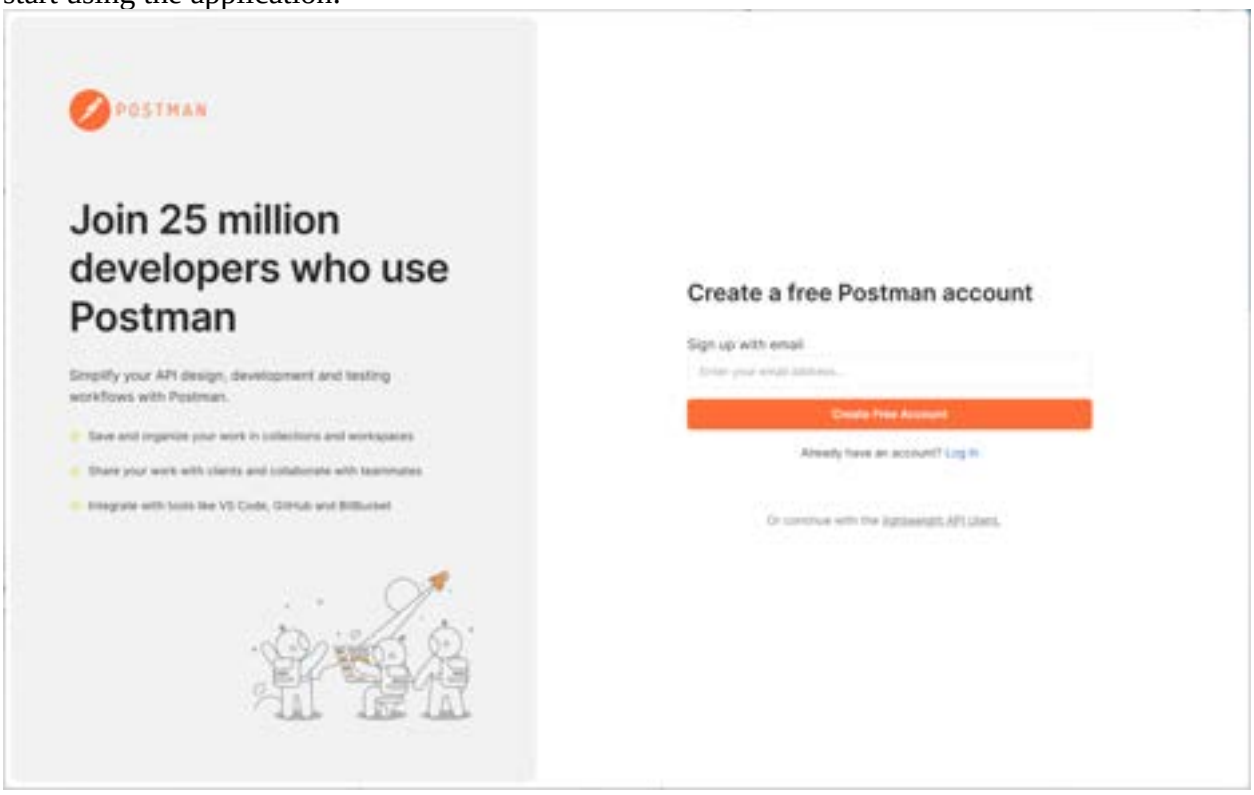
Installing Postman

Postman is an application that is commonly used to debug and test APIs. It allows you to have a convenient platform to manipulate API parameters as well as HTML header information which reduces the amount of time needed to debug API development. We will be downloading Postman from here https://www.postman.com/downloads/ . Select the version appropriate to your operating system and download the package. :

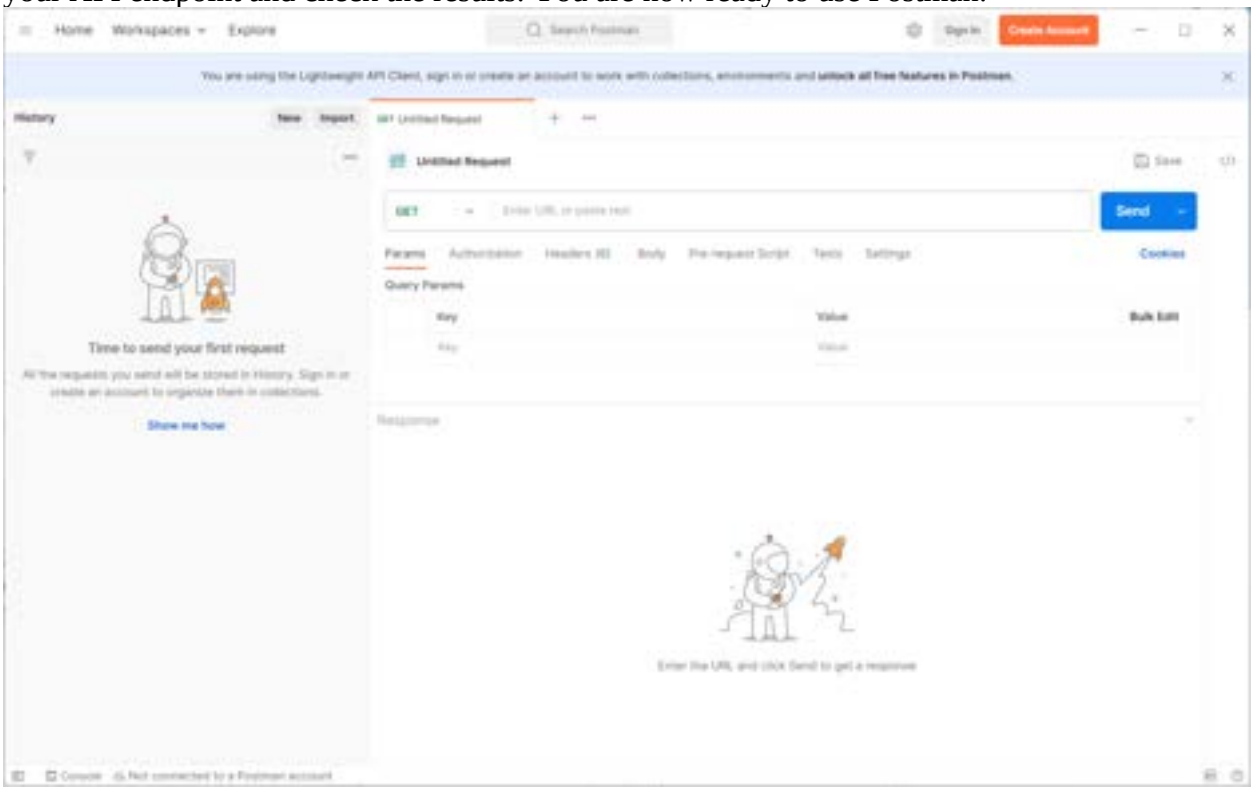


When the file has downloaded, double-click on the installer to start the installation process. There are no options to consider here, the installer will install the application into your environment and place a link on your desktop. The installer will open the application. You do not need to create a postman account, you can select ‘continue with the lightweight API client’ to

start using the application:



Selecting the lightweight API client will bring up the panel where you can select the URL for your API endpoint and check the results. You are now ready to use Postman:



Creating Your First Static HTML Website

These instructions assume that you have already created your cs465-fullstack repository in your GitHub account. If you have not already done so, please follow the GitHub Tutorial to get your initial (empty) repository setup. You need to complete the tutorial through page 6 before continuing.

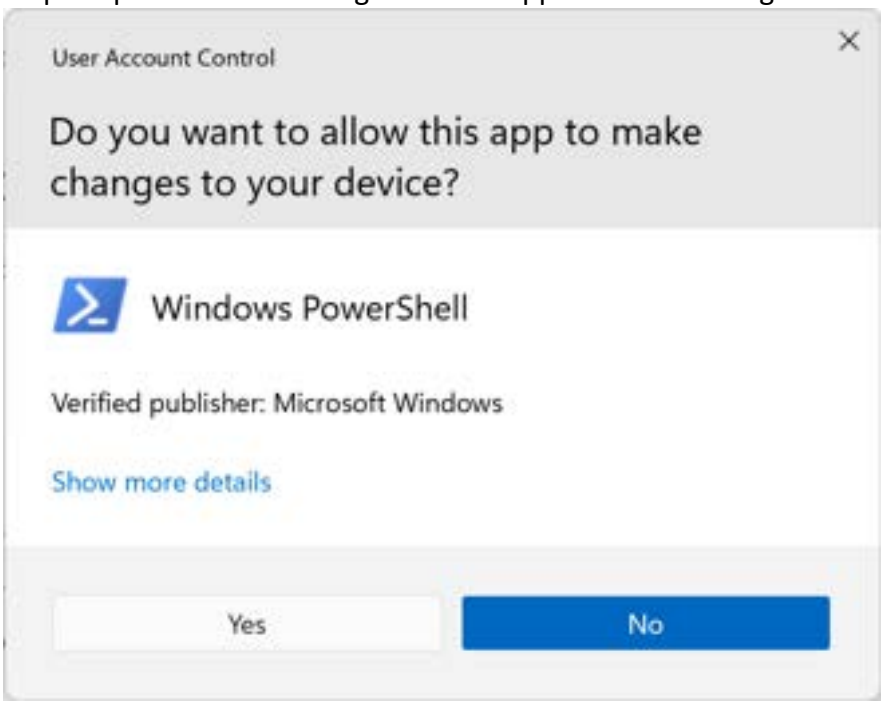
Please Note: Make sure that you have already downloaded the travlr.zip file (linked in the 1-6: Module One Assignment Guidelines and Rubric).

Configure PowerShell to Accept and Run Scripts

The next step will be to adjust the ability of PowerShell to run scripts on your system. Current versions of Windows 10 and Windows 11 prevent script execution in PowerShell by default, so we will need to adjust this setting.

Please Note: if you are using a Windows Instance that is a member of an Active Directory Domain you will either need administrative rights on your computer or a Domain Admin will need to make the change for you in a Group Policy Object. The following instructions assume that you have full rights and control on your own system.

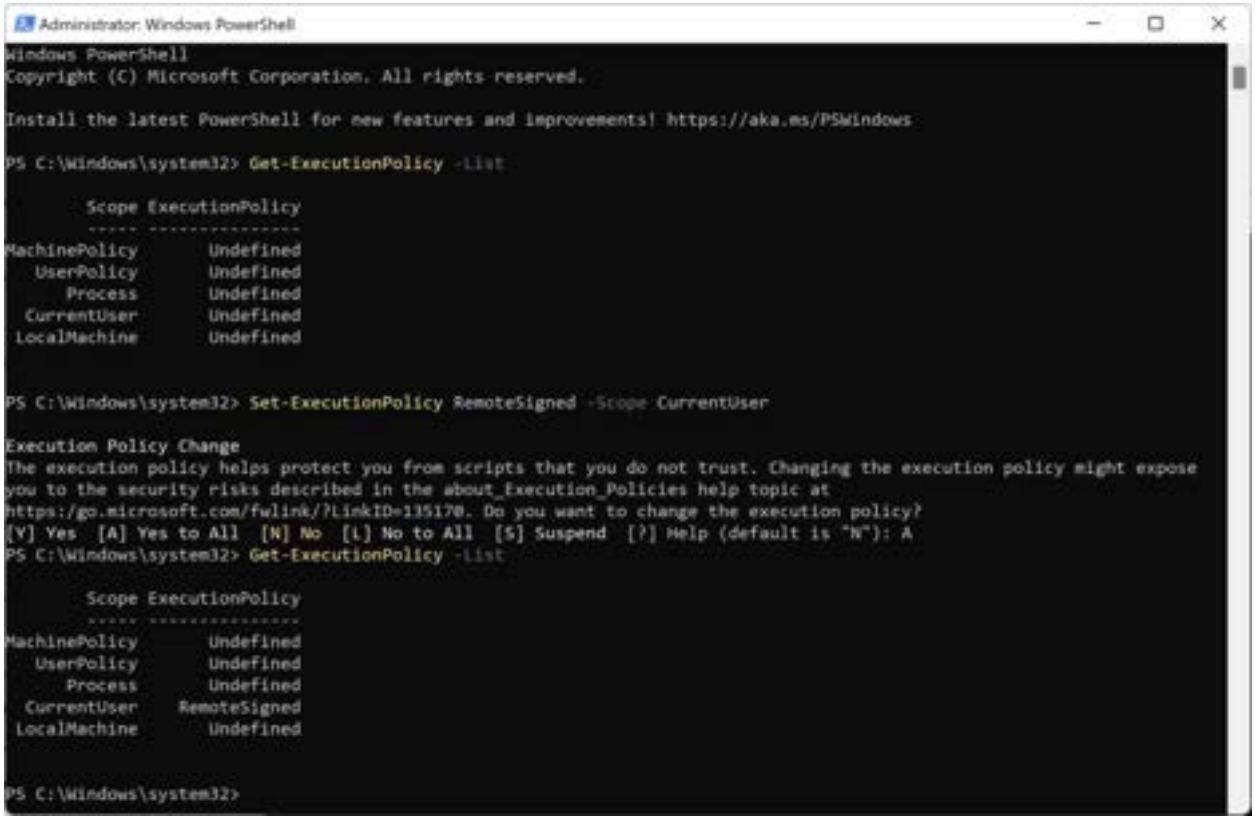
1. Open a Windows PowerShell Command Prompt as an Administrator. You can accomplish this by right-clicking on the power-shell icon and selecting Run-As Administrator. You will be prompted to acknowledge that the App Can make changes to the system. Select Yes:



2. Now that we are working in a PowerShell Window with administrative capabilities we are going to check the execution policy – by default set to Undefined (meaning no privileges), and then we are going to set the execution policy for the Current User to

'RemoteSigned'. This allows all locally developed scripts to run as well as scripts downloaded from the Internet that have been signed by a recognized authority.

```
Get-ExecutionPolicy -List
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser
Get-ExecutionPolicy -List
```



3. You can verify this in your non-administrative PowerShell window by running:

```
Get-ExecutionPolicy -List
```

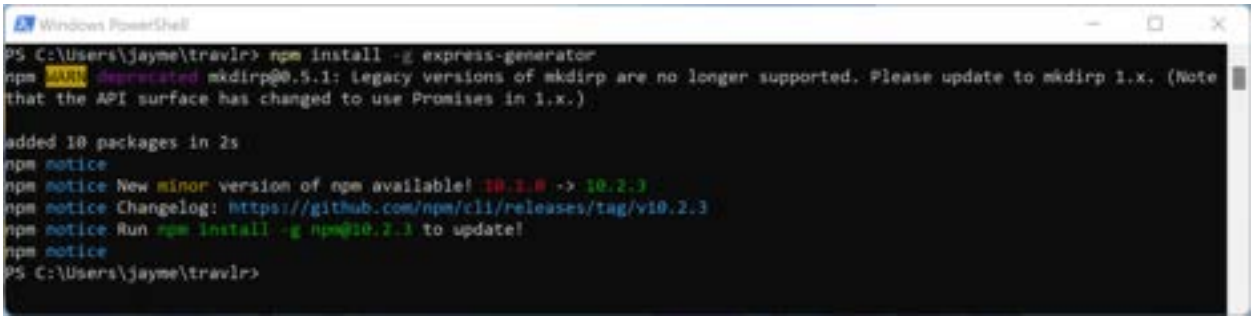
Create Your Initial Website

1. Open a Windows PowerShell Command Prompt and make certain that you are in the root directory of your travlr folder:

```
cd ~/travlr
```

2. In the next step, we are going to create and initialize a Node Express web application, configured with the Handlebars (HBS) view engine.
    a. First, install the Express application template generator using the `-g` switch for global installation, which will make the generator available for all projects.

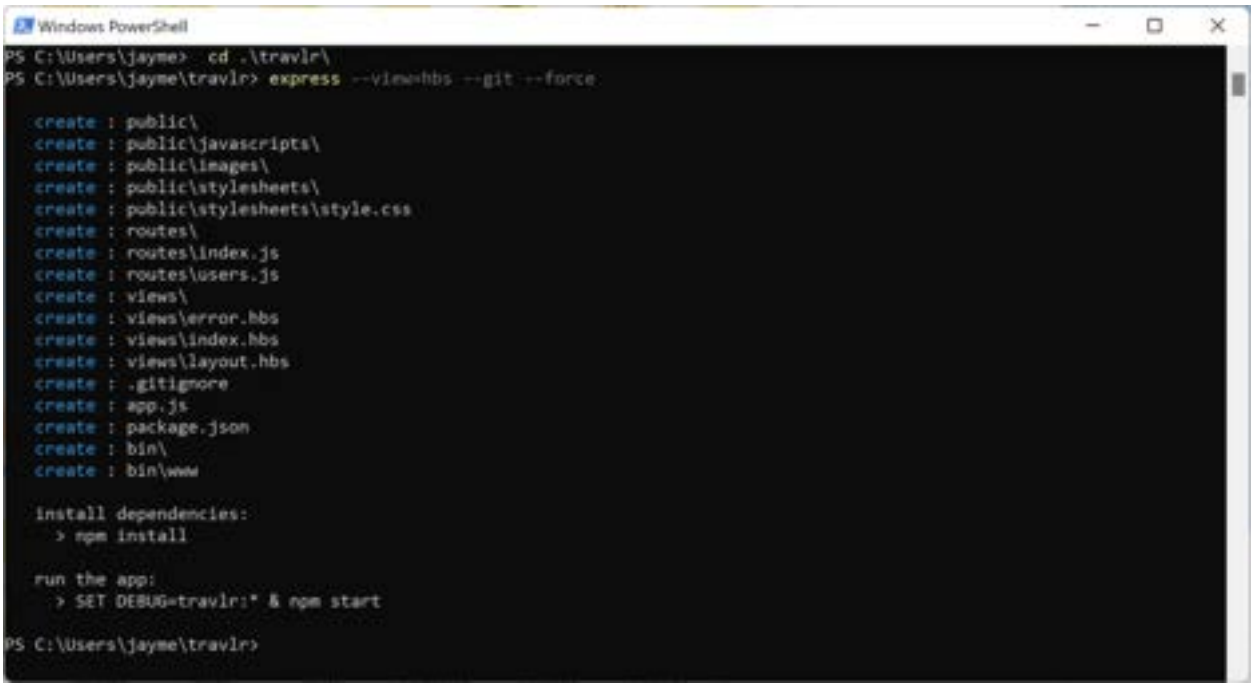
```
npm install -g express-generator
```



Note: if you receive a message that you can upgrade to a newer version of node, you can do that at this point but it should not be necessary.

b. Generate an (empty) Express web application using the **Handlebars view engine** and a default Git configuration, if one does not already exist.

```
express --view=hbs --git --force
```

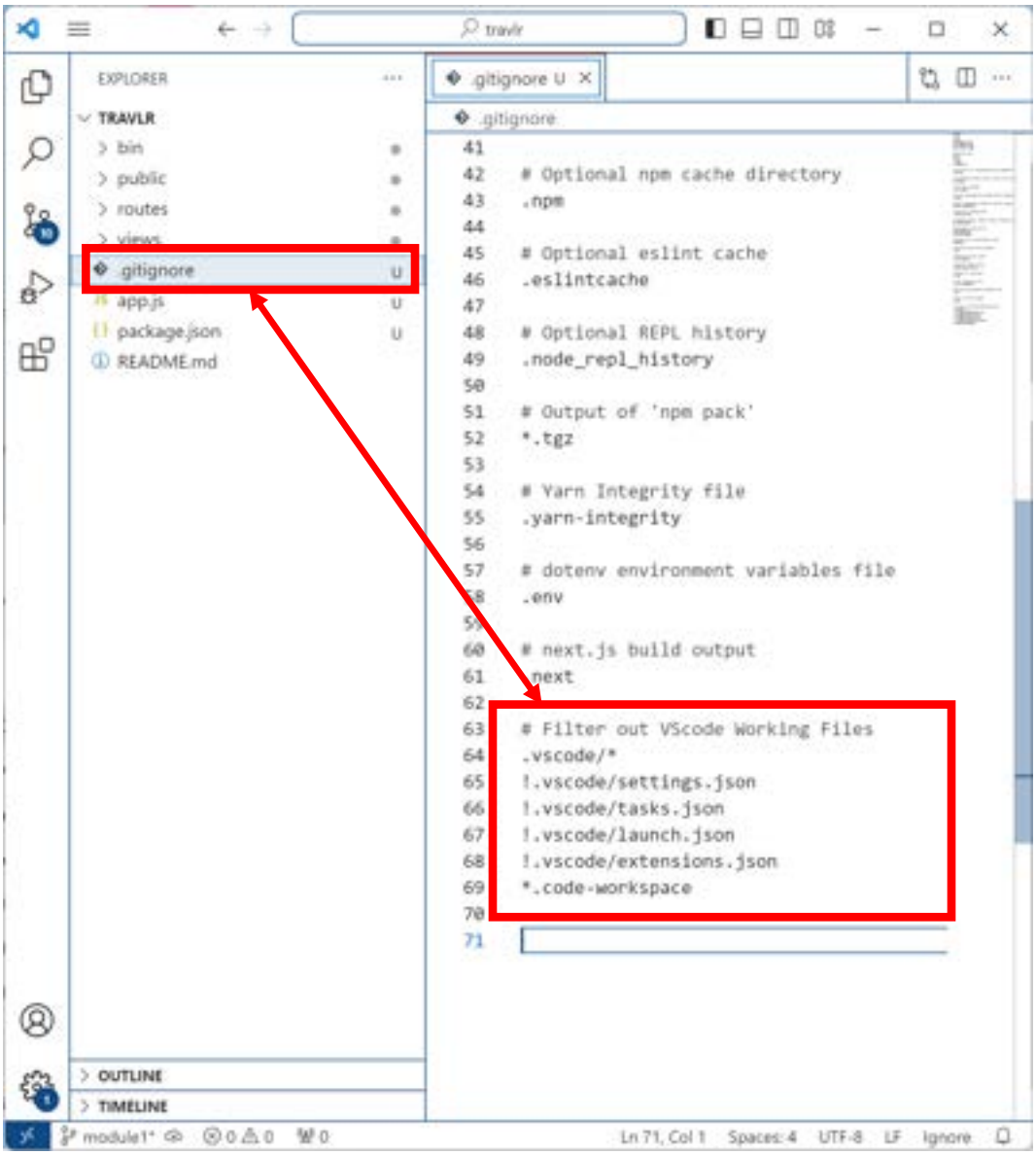


Please Note: the various sub-directories and website code files shown above that were created for you by the Express Generator tool.

3. Launch the Visual Studio (VS) Code editor and open the newly created Express website.
   a. Click the **Explorer tool** button (two document icons in the upper left).
   b. Click **Open Folder**, select the **travlr** folder you just created, and click **Open**.

4. Edit the ".gitignore" file and add instructions to ignore the VS Code working files when committing your source code to Git by copying the following:

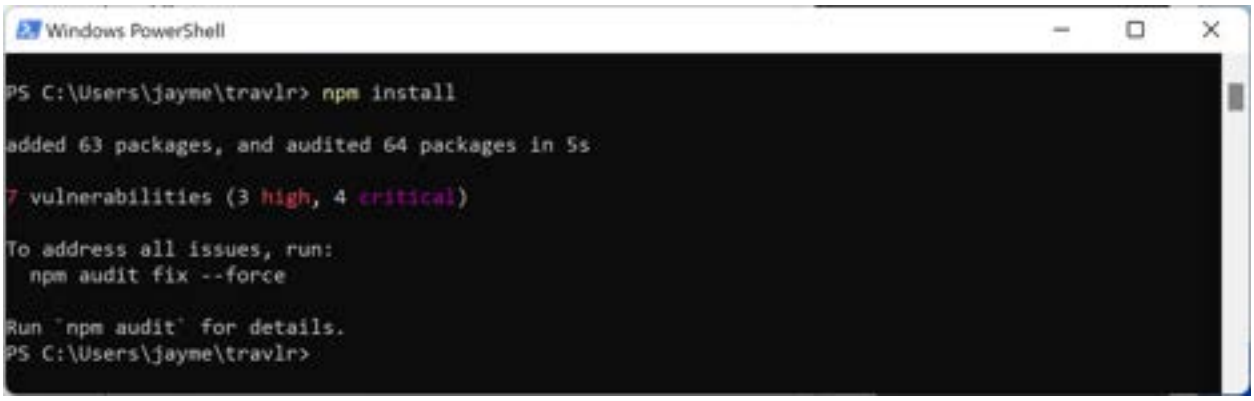
```
.vscode/*
!.vscode/settings.json
!.vscode/tasks.json
!.vscode/launch.json
!.vscode/extensions.json
*.code-workspace
```



Remember to save the file after making the changes!

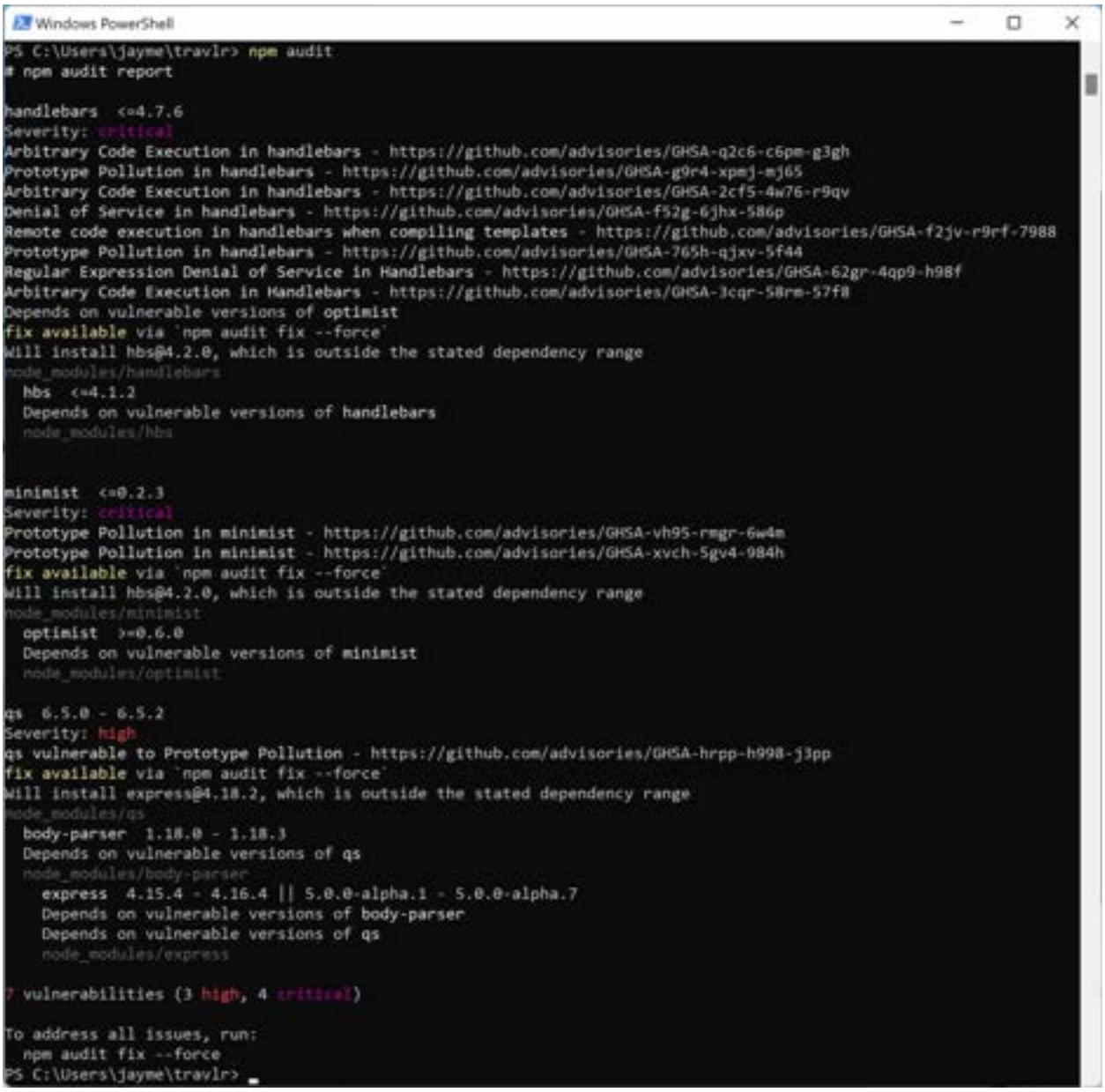
5. Back in the Windows PowerShell command window, install the Node packages automatically included in **packages.json** when the website was generated using the following command:

```
npm install
```



You will note that you may see an indication that some of the packages installed are the subject of current vulnerabilities. It is always a good idea to view the vulnerability report to determine what may need to be done before continuing. We will accomplish this with the following command:

```
npm audit
```

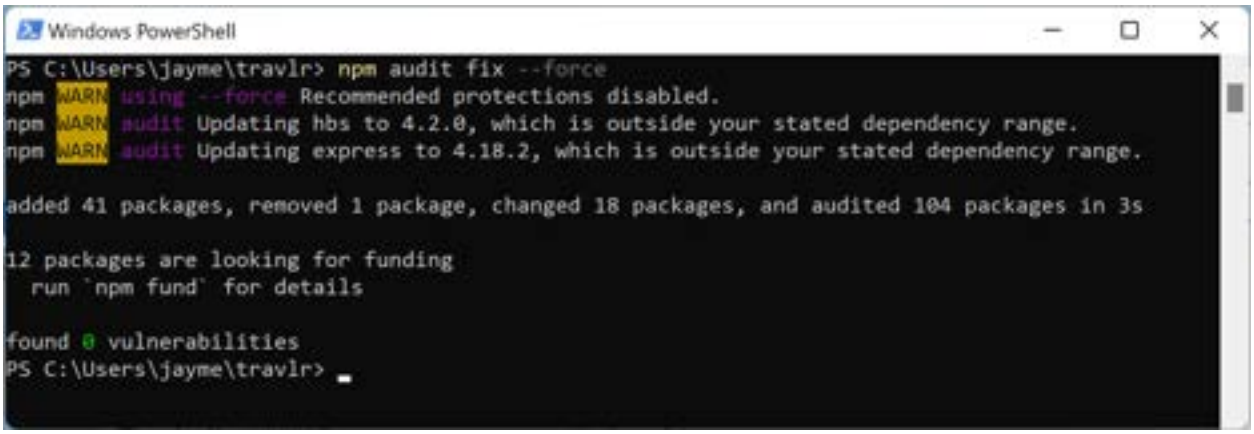


While it is always important to eliminate vulnerabilities in your software wherever and whenever possible, it may not always be possible due to compatibility (or incompatibility) between modules in an application.

Note: It is always a good practice to save your current work to git and perform an update on the impacted packages. If your application breaks, you can blow the directory away, re-run the simple install, and recover your last known good configuration directly from git.

Should you choose to run the package upgrades, you can accomplish this by running:

```
npm audit fix --force
```



Note: This may leave you with some packages that are now newer than the identified dependency ranges, but this may also be due to a lack of time to test. Ultimately, the decision to update or not is up to you, but being familiar with how to accomplish this and how to troubleshoot and address issues that may arise is central to being able to perform these activities in a professional environment.

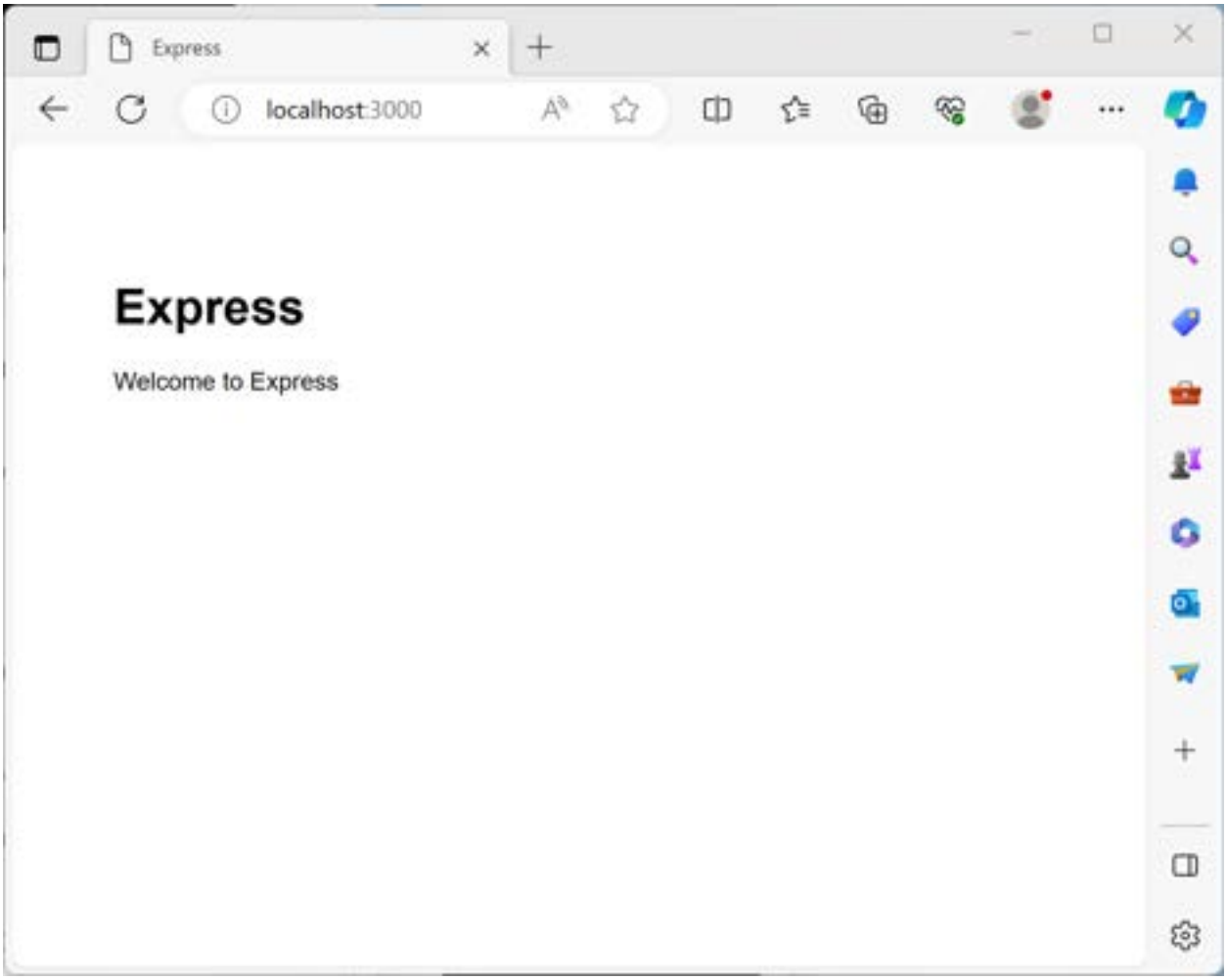
6. Now that we have the necessary packages loaded we are going to start the web server. The command to do this is exactly what was shown when we setup the web site with the express-generator command.

    a. set DEBUG=travlr:*
    b. npm start

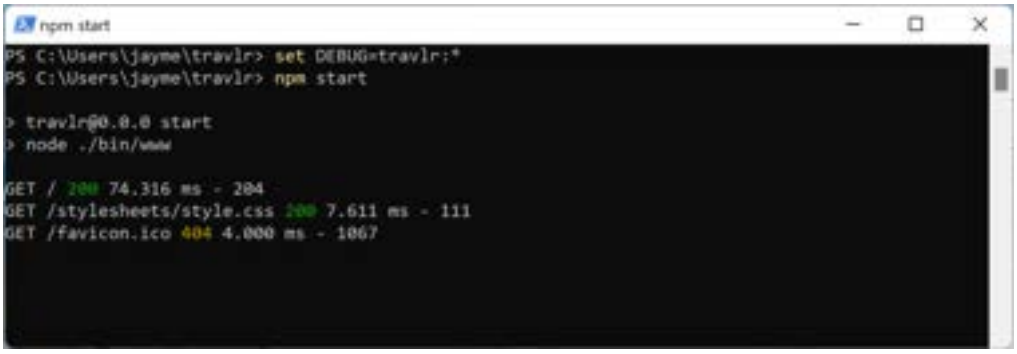
7. When we try to start the web server you may get prompted to allow communication through the Windows firewall. Check both boxes and select 'Allow Access' to continue.

You must do this to allow the web server to properly communicate with the browser on your machine.

8. Open your browser and connect to the following URL: http://localhost:3000. You should see something like this in your browser window:



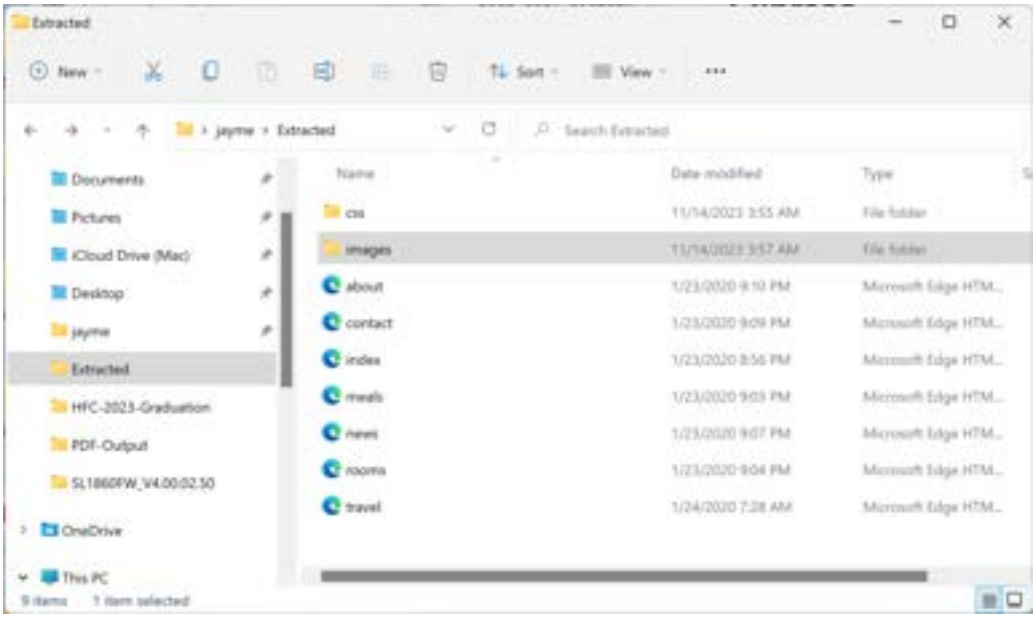
and you should see the access request that your browser made in your NPM window as well:



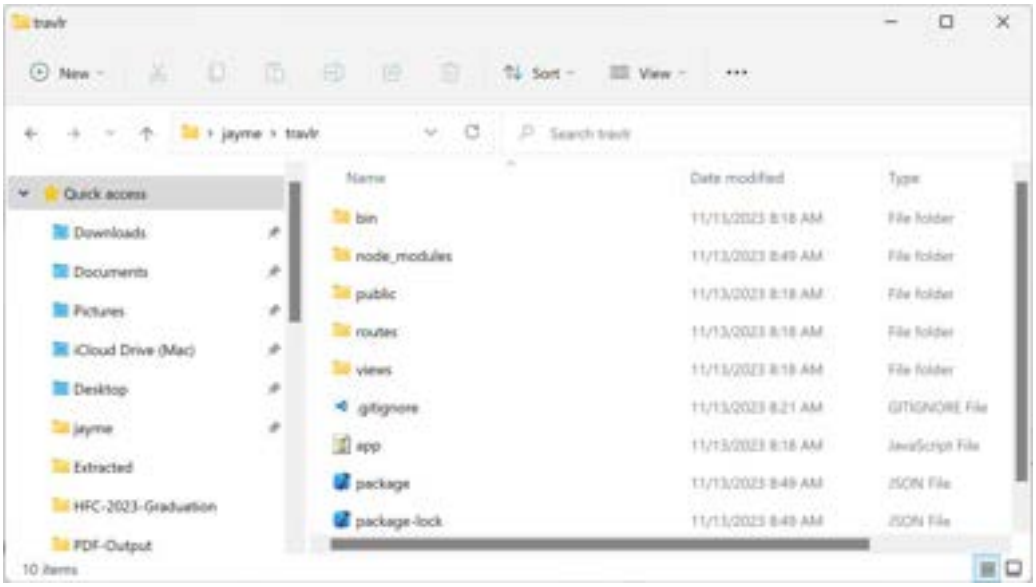
Install Static Web Files

Now that we have the basic web site up and running it is time to add the static web files that you downloaded in the travlr.zip file from the Assignment Guidelines and Rubric for Assignment 1-6.

1. Open two Windows Explorer windows so that you can focus on the directory containing the unzipped files in one window, and the travlr directory in the other.



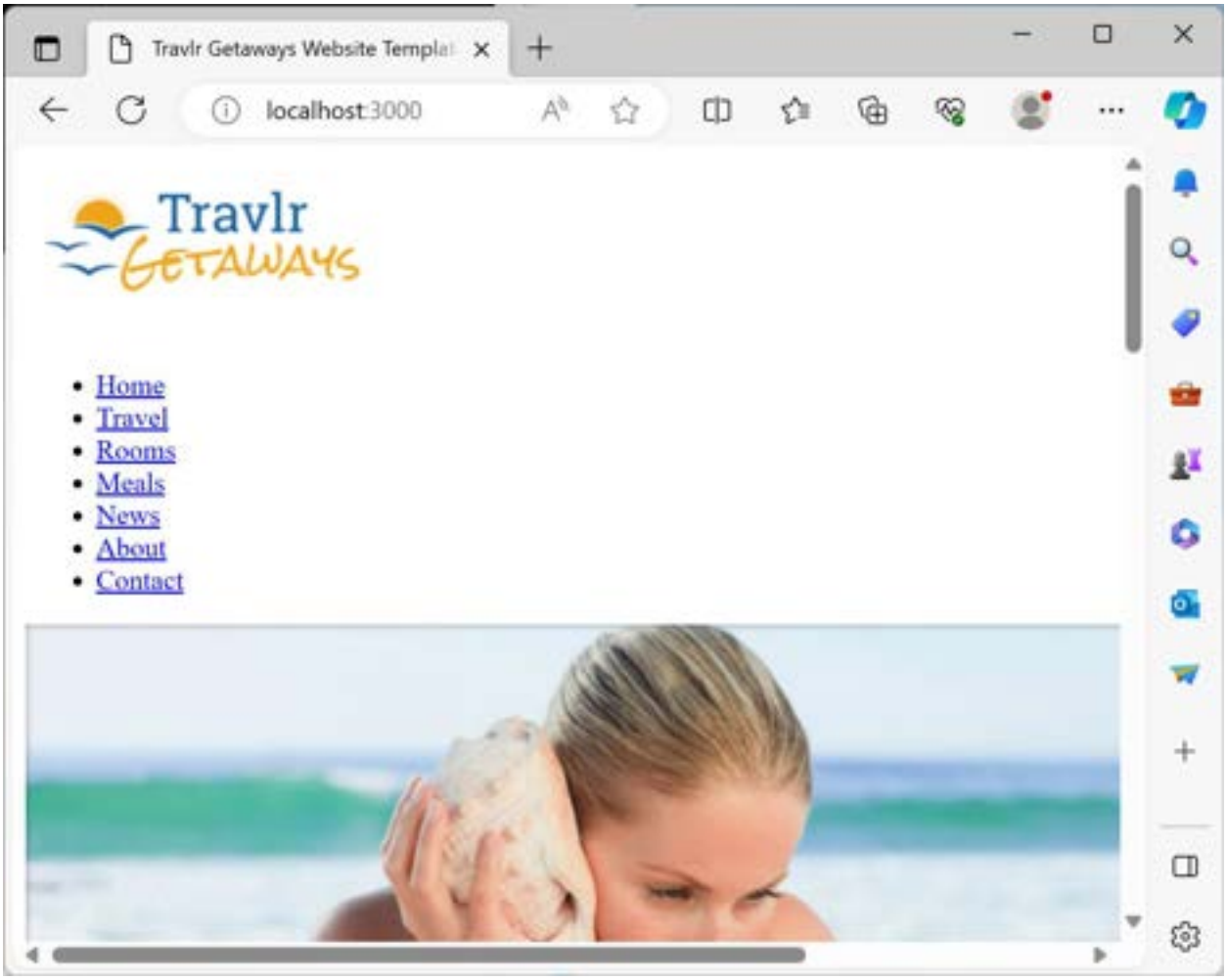
And



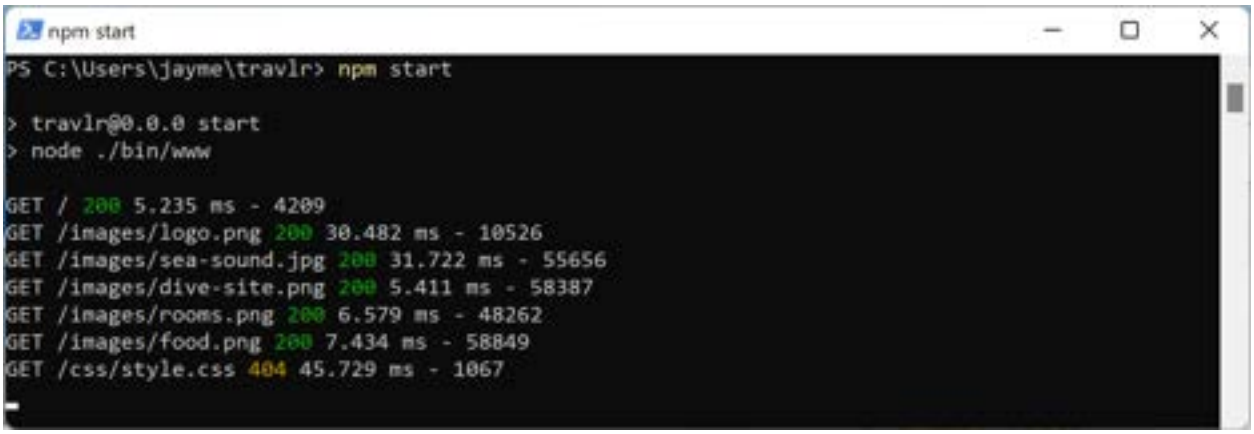
2. Copy the .html files to the public directory beneath travlr (drag and drop).

3. Copy the files from the image directory to the public/images directory in the travlr folder.
4. Copy the stylesheet from the css folder to the public/stylesheets folder.
5. Stop and restart the Node.JS webserver. You can accomplish this by pressing ctrl-C in the Powershell window that is currently running the webserver. This will stop the webserver and then it can be started up again with 'npm start'.

Once you have restarted the webserver, you should check it with your browser:

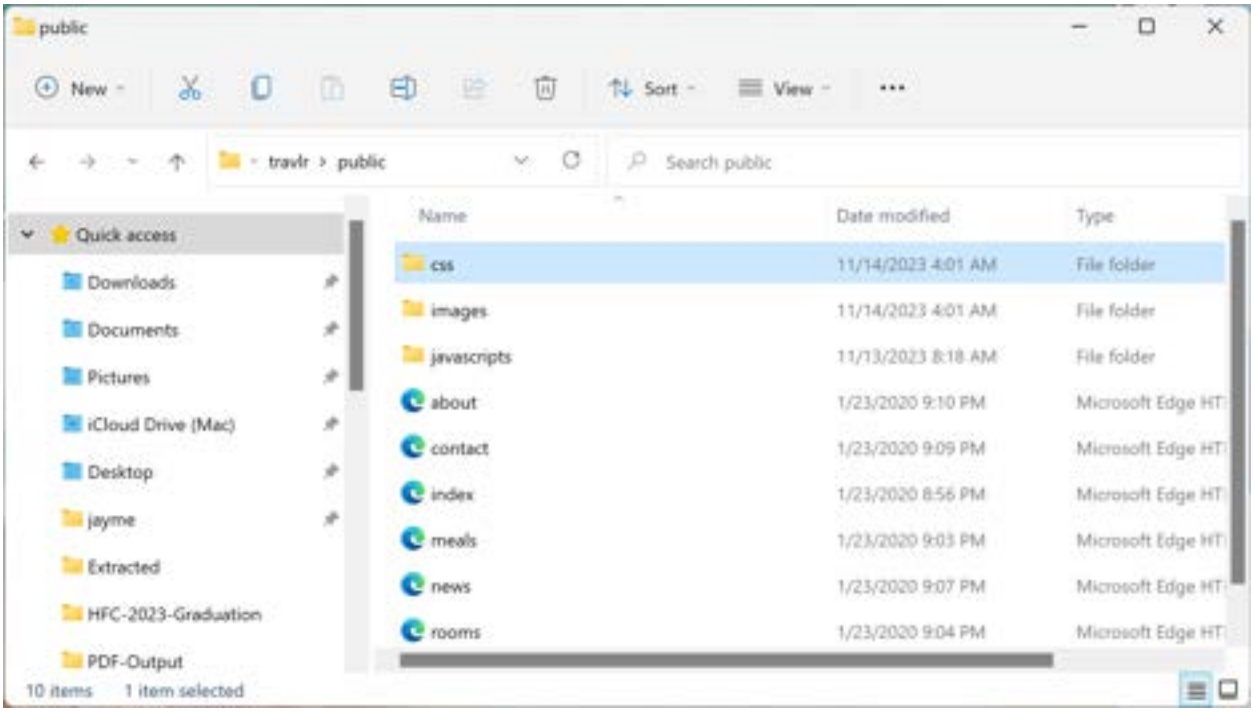


This does not look like what we are expecting. Why? You can see an indicator in the NPM window:

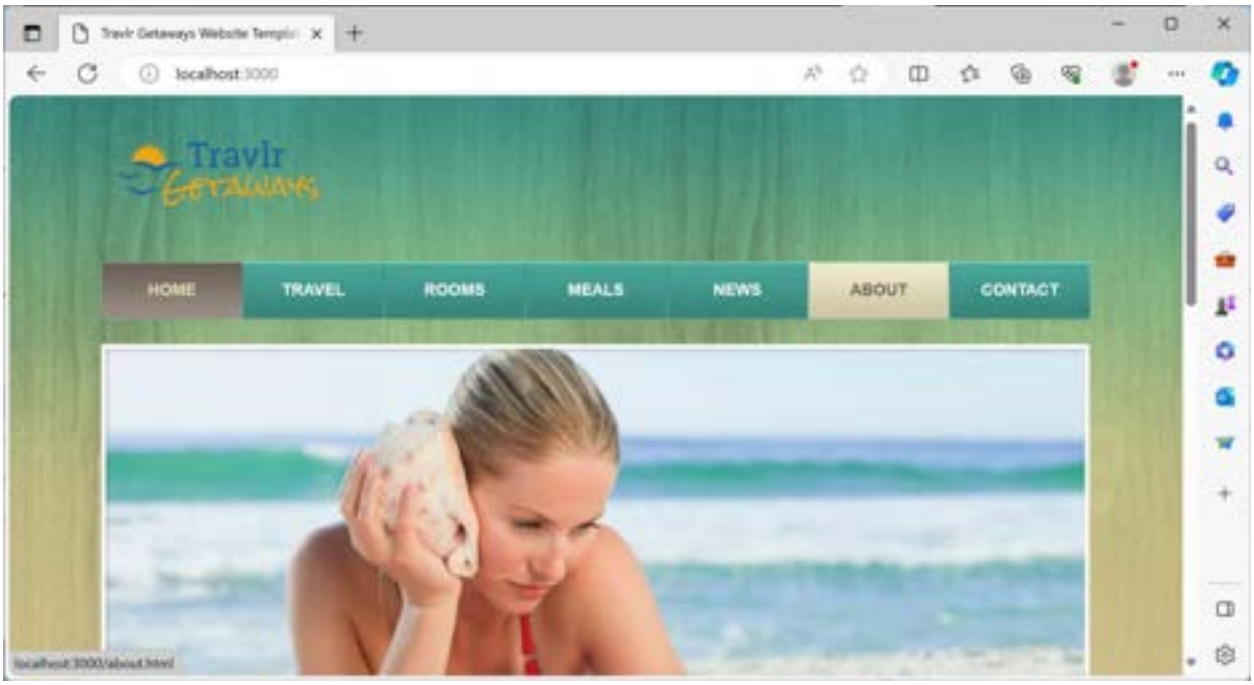


Here, you will see that style.css is not found at all. This is the output from running the npm command. You can see that the application is looking for the style.css file in the /css folder from the website. We can accomplish this by renaming the public/stylesheets folder to css and restarting the webserver.

Please Note: Express assumes that the default directory for the .css files is stylesheets – but it is easier to rename this than it is to edit all of the files in the example website.



Reload the page in your browser and you should see this now:

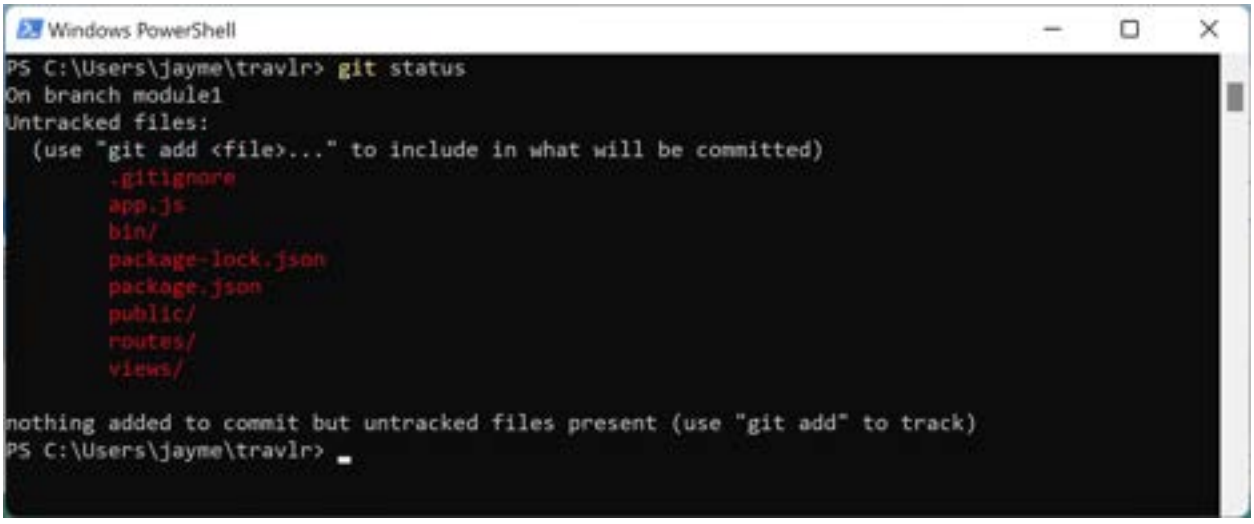


Finalizing Module 1

The final step for Module 1 is to validate that you have all of your code in GitHub and under configuration control.

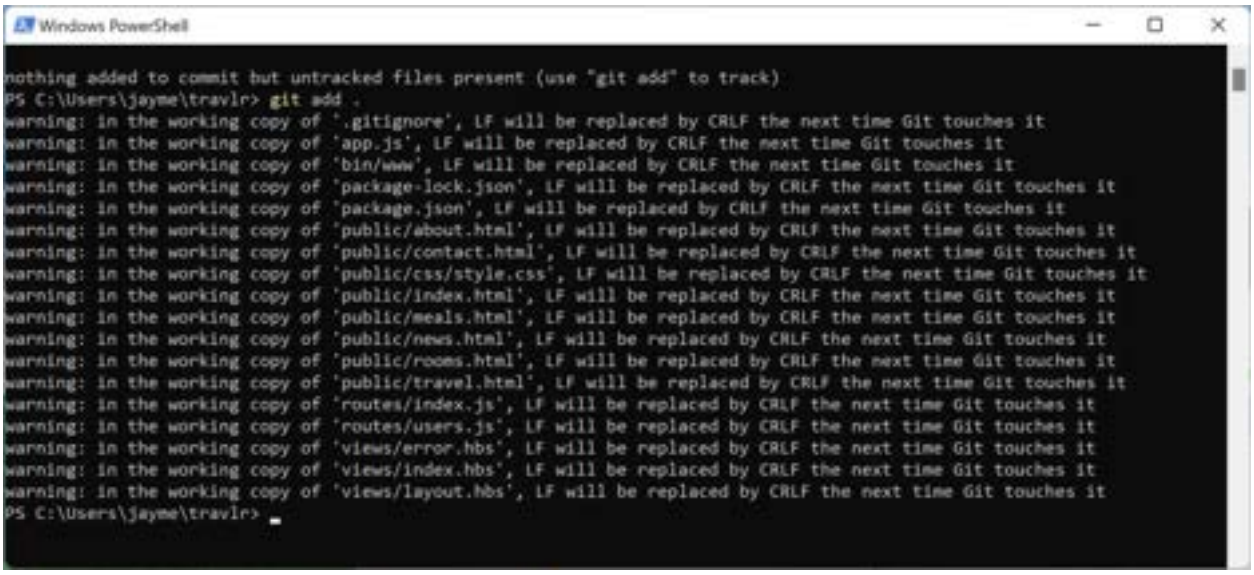
From your PowerShell command window, from the travlr directory, check the status of your files:

```
git status
```



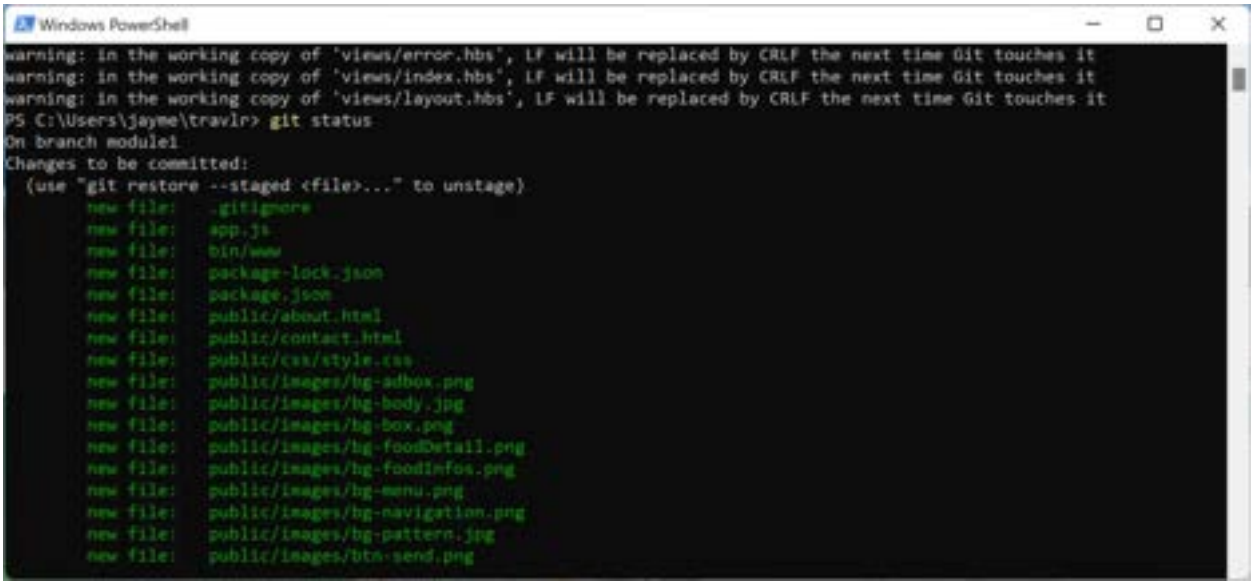
In order to stage and accept all of the un-tracked files into change management execute the appropriate ‘git add’ command to bring everything in the current directory and below the current directory into tracking:

```
git add .
```



Note: based on your initial configuration of git within your environment, you may be notified about changes to the physical format of your files will be handled when pushed to git.

At this point, you can run another ‘git status’ command to see all of the new files that will be added to tracking:

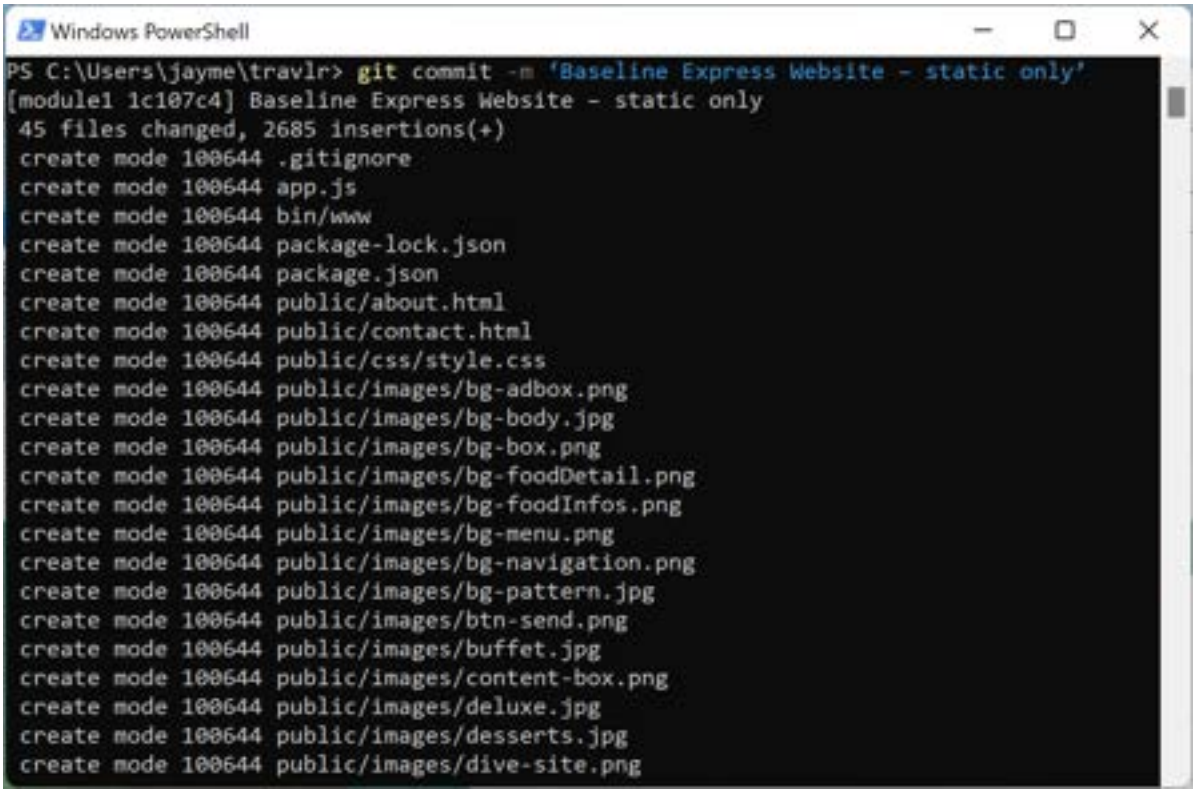


Now that we are ready to commit our files to tracking, we have to first configure two global variables for our git repository. The information that needs to be set are the git user.name and git user.email parameters. We will set these with the following commands (substituting your own values for the contents between the quotation marks. You should use the e-mail address you registered for your GitHub account.) **Please Note:** These commands will set the configuration variables for your entire system. If you need to set them only for the current repository, please omit the '`--global`' flag.

```
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
```

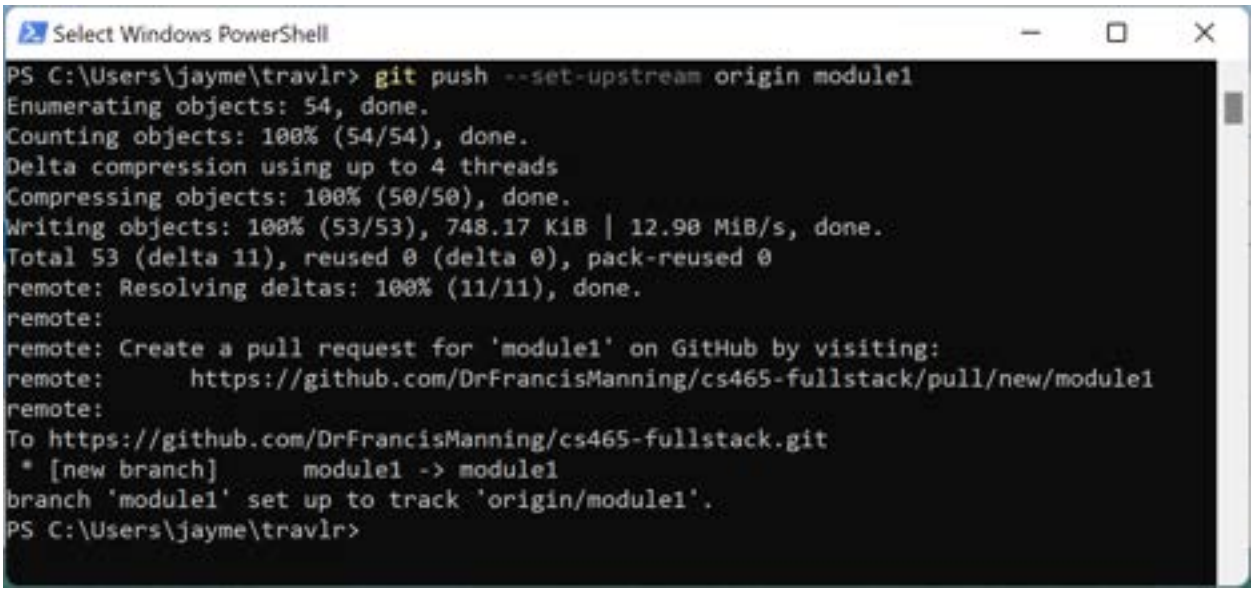
Now that they have been added, it is time to commit them to our repository. This is done with the following command:

```
git commit -m 'Baseline Express Website – static only'
```



Now that the files have been committed, we need to push them to our GitHub repository. This final step in the process will push everything to GitHub and we will have a clean copy that we can revert to if we have any trouble in our development process.

```
git push --set-upstream origin module1
```

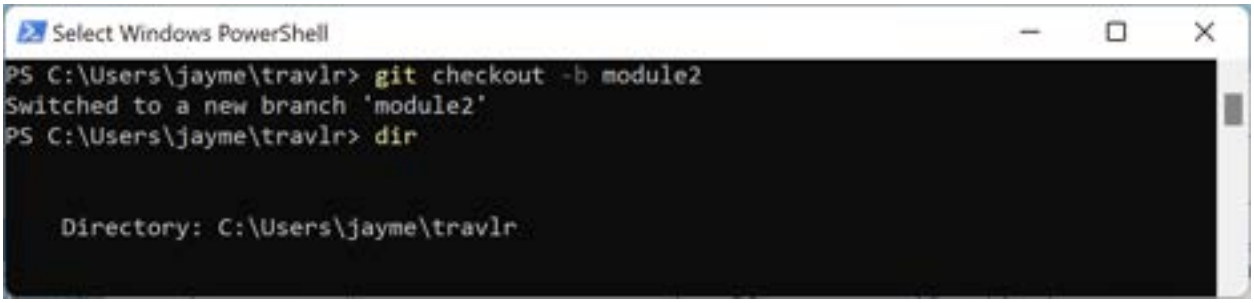


Module 2: Model View Controller (MVC) Routing

Create Git Branch for Module 2

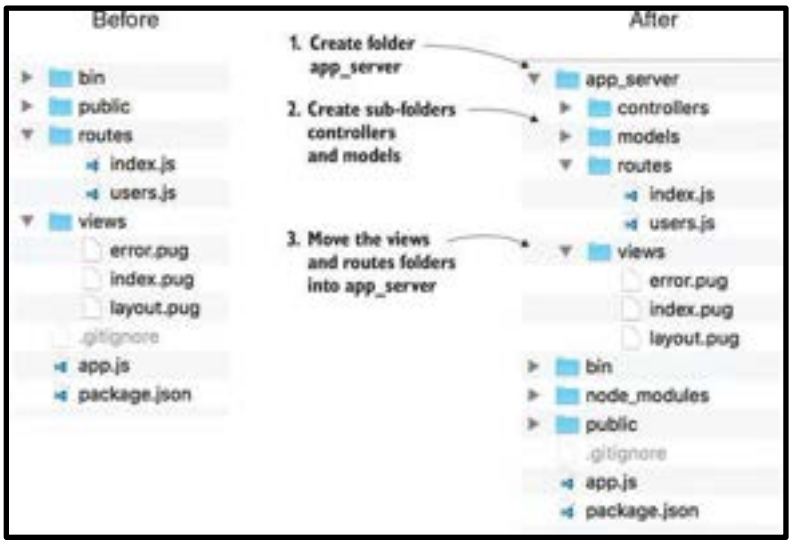
Before you begin, it is important to make sure that you have created your new branch in git for Module 2. To accomplish this, we will perform the following command in a PowerShell window in the travlr project directory:

```
git checkout -b module2
```

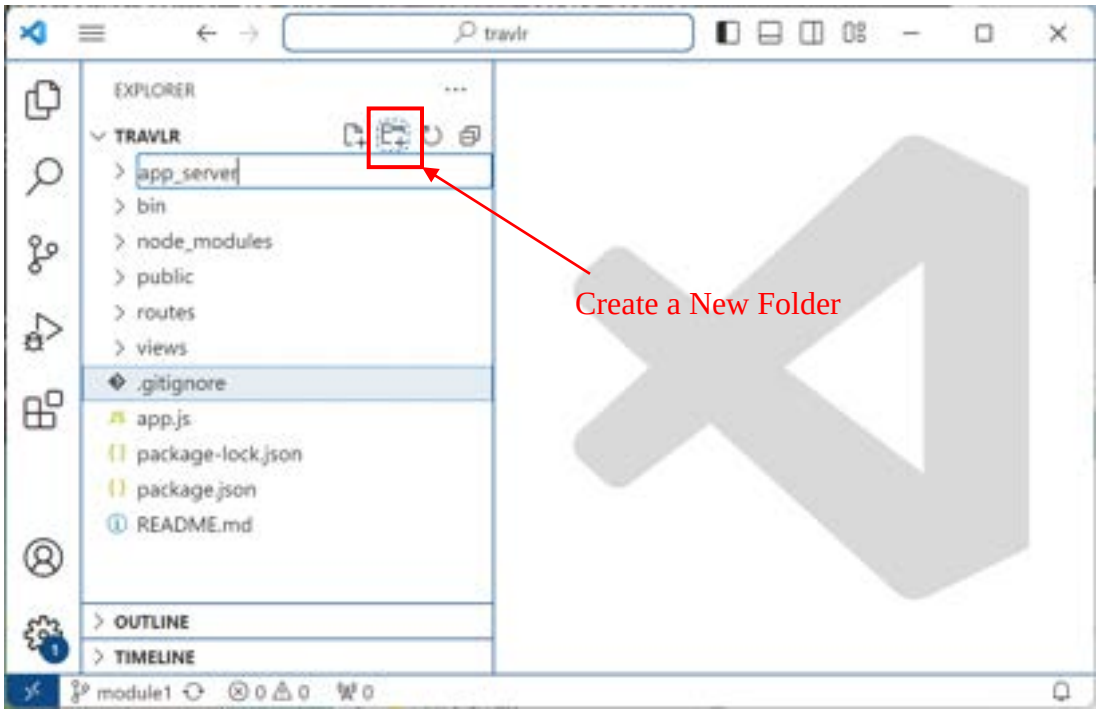


Create Web Application Folders

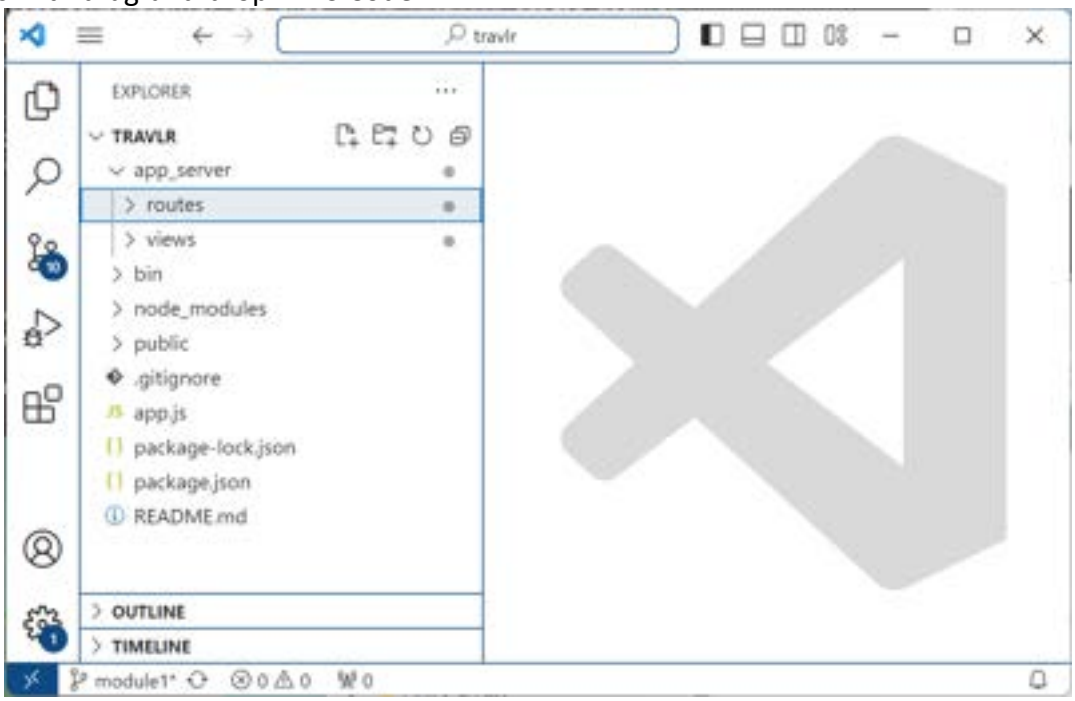
1. Create an **app_server** folder where we'll migrate the static HTML to Handlebars template views, and refactor common functionality such as headers and footers into "partials". This section corresponds to Chapter 3.3 in your textbook. Refer to Figure 3-6 in the textbook to visualize the reorganization of the default folders in the website.



2. In VS Code, create a new folder named 'app_server' under the travlr project.

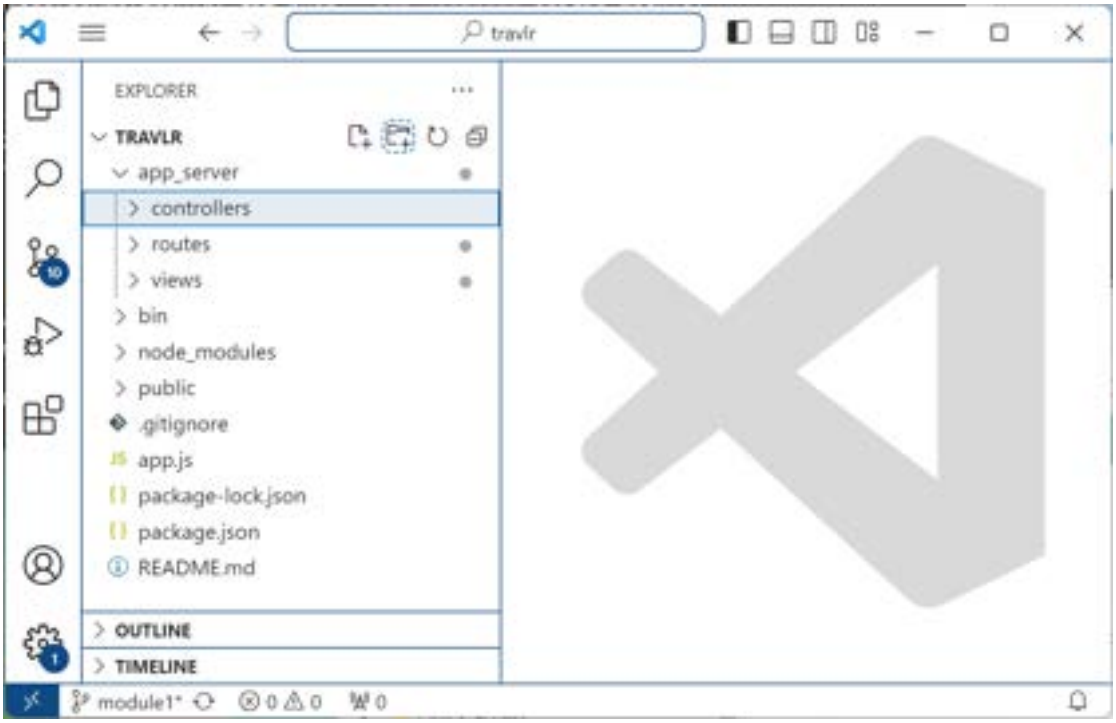


3. Move the existing ***routes*** and ***views*** folders into the new ***app_server*** folder. You can do this with drag-and-drop in VS Code.

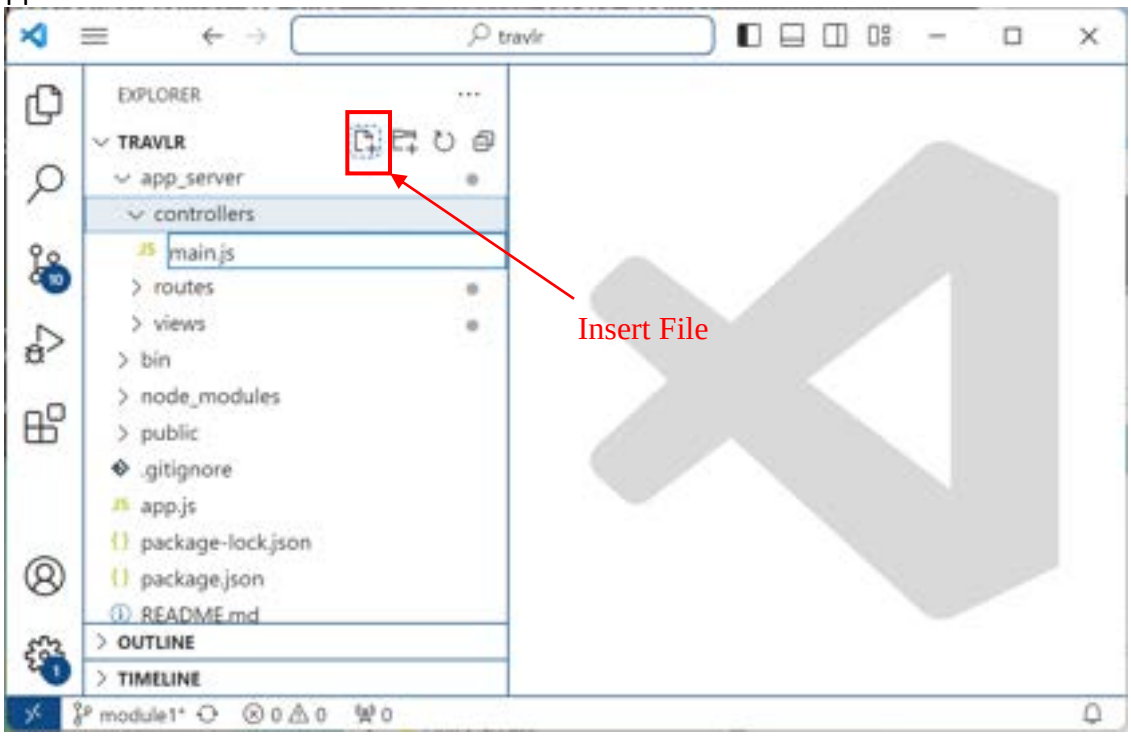


Creating Controllers and Routes

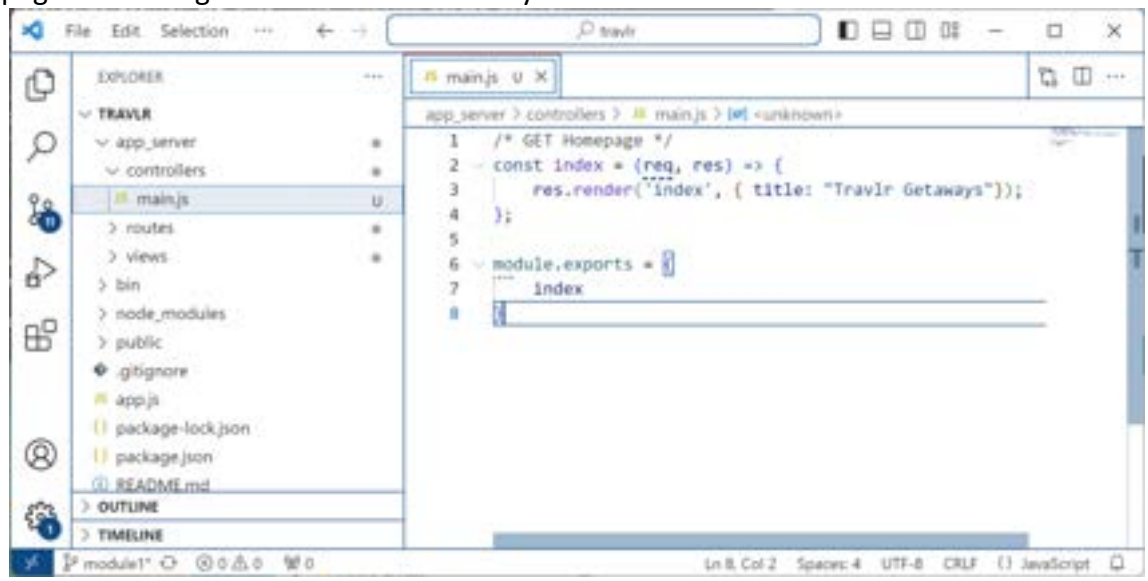
1. With the ***app_server*** folder selected, click the new-folder icon and create a folder named ***controllers***.



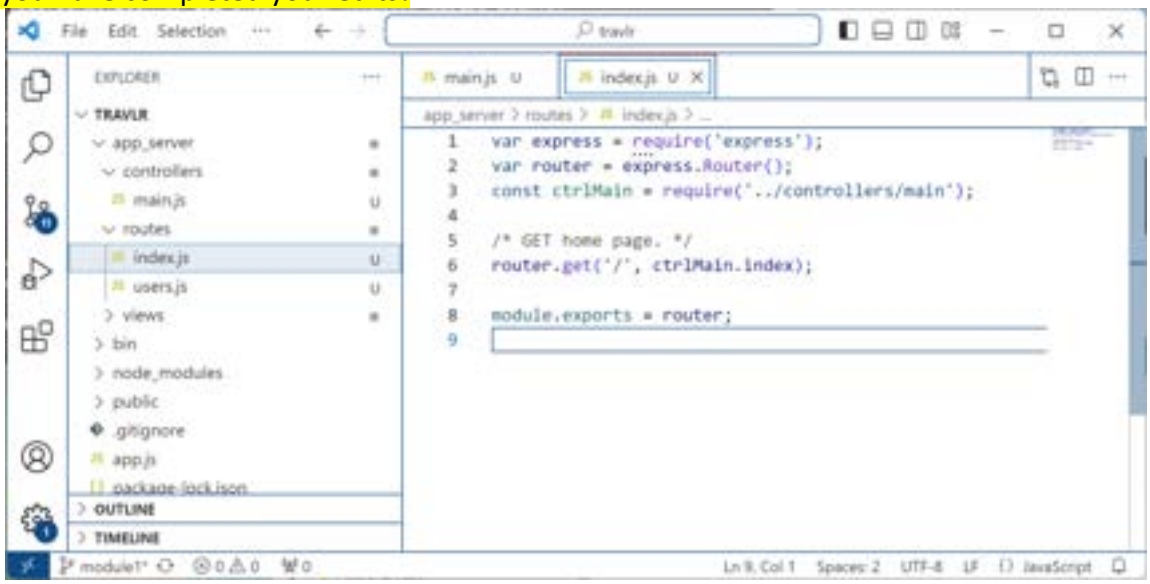
2. Once the ***controllers*** folder is present, select the folder and then use the Create-File icon to create a new file that will be named ***main.js*** that will be the first controller for this application.



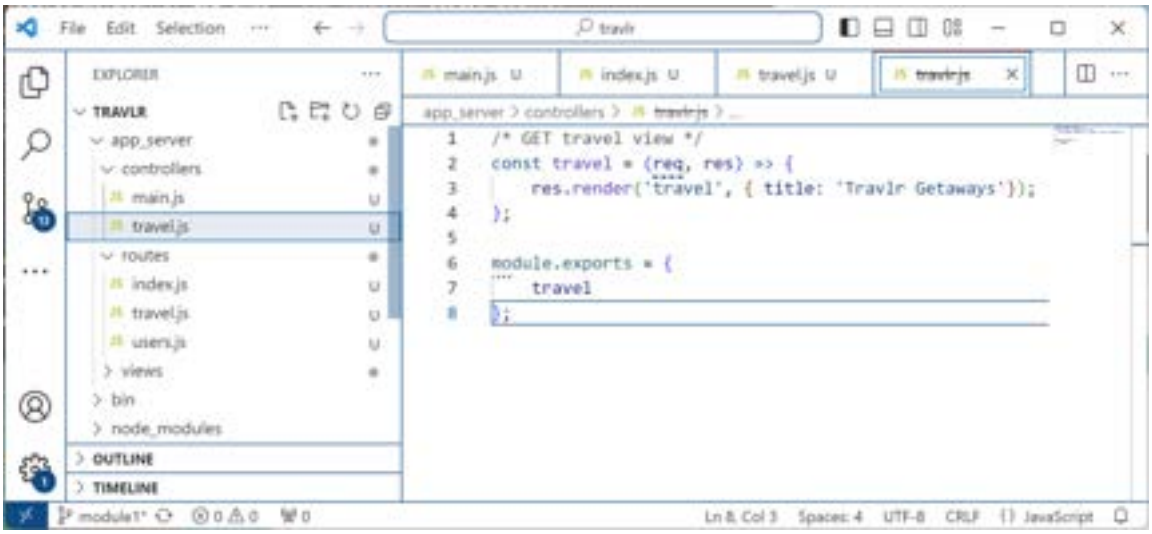
3. Edit the ***main.js*** file – this is where we will wire up the controller to serve the main index page. Don't forget to save the file when you are done!



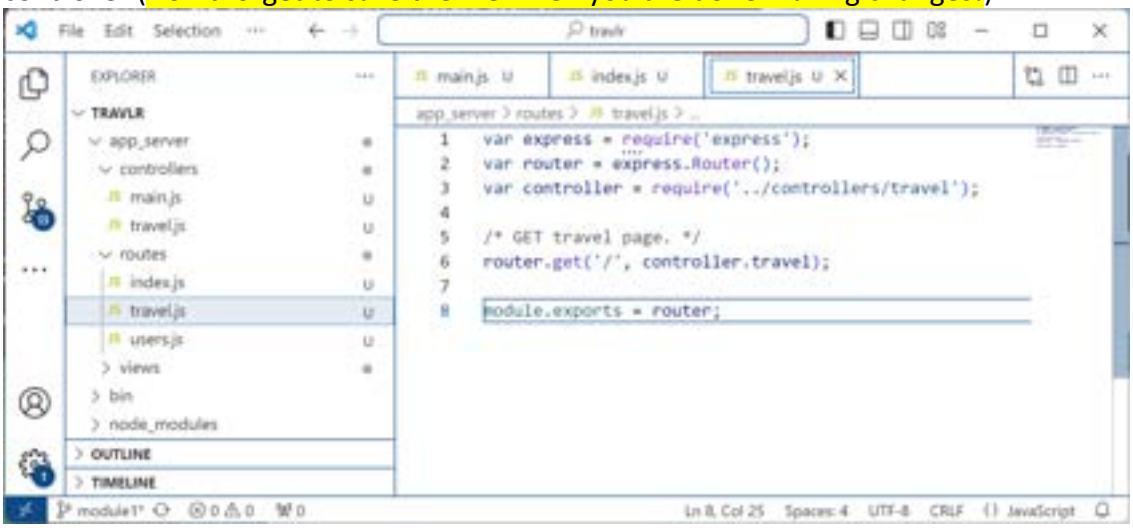
4. Next we need to edit the ***index.js*** file in the ***routes*** directory to pass the request for the site-default starting page (often referred to as the the 'root' or "/") to our new main controller. We will do this by adding a reference to the controller and altering the function that gets passed to the ***get*** call for the index. Don't forget to save the file when you have completed your edits!



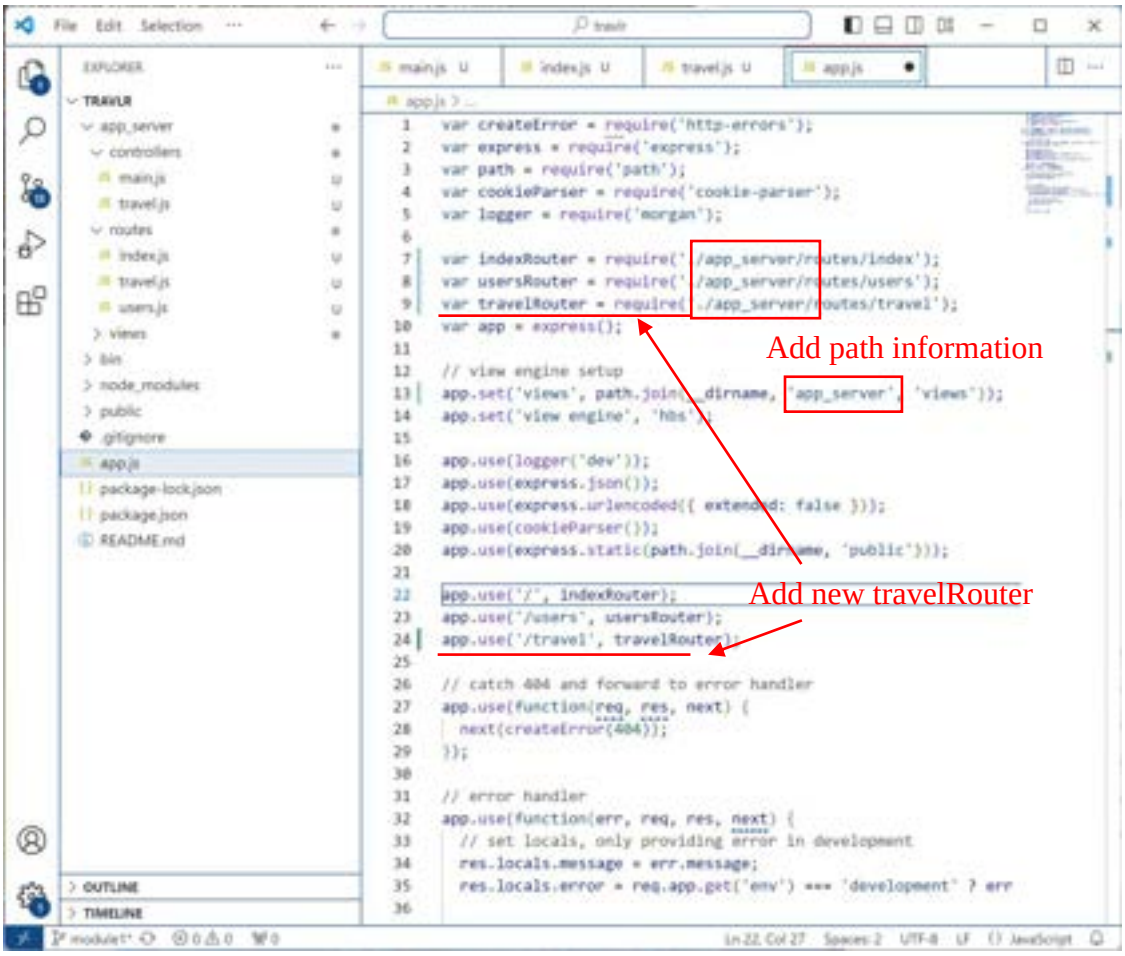
5. Next, we need to create a controller for the ***travlr*** page. This is a repetitive process of what we went through to create the ***main*** controller. Create the controller for the ***travlr*** page (Don't forget to save the file when you are done making changes!):



6. Similar to what we did for the ***main*** controller, we now need to add a route for the ***travlr*** controller (Don't forget to save the file when you are done making changes!):



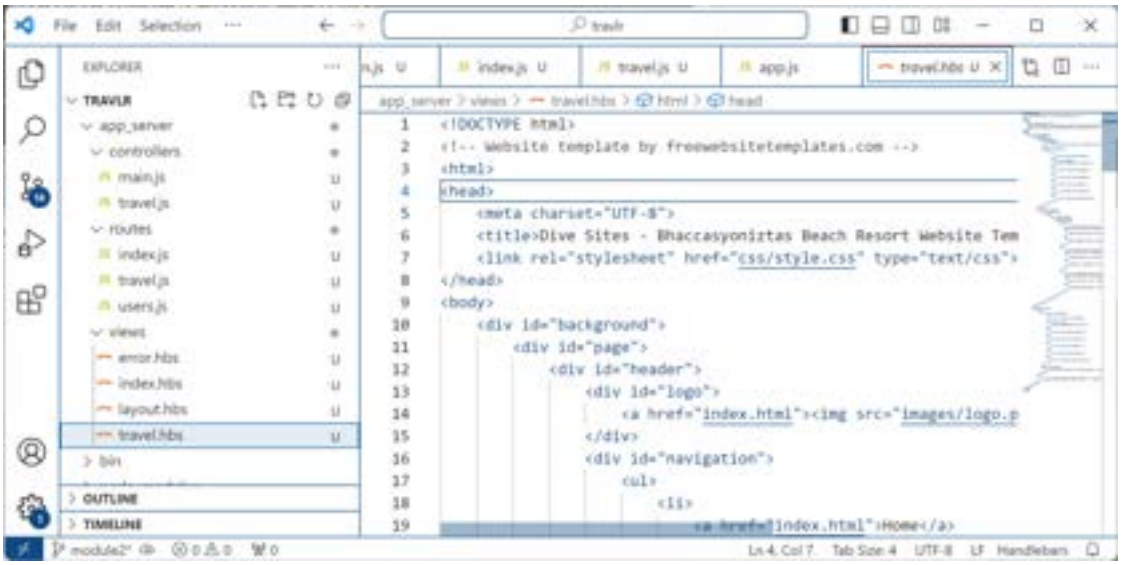
7. Finally, we need to edit the app.js file to add the ***app_server*** folder to the application path, and wire up the ***travlr*** controller connecting it to the route for the '**/travel**' page. Don't forget to save the file when you are done making changes!



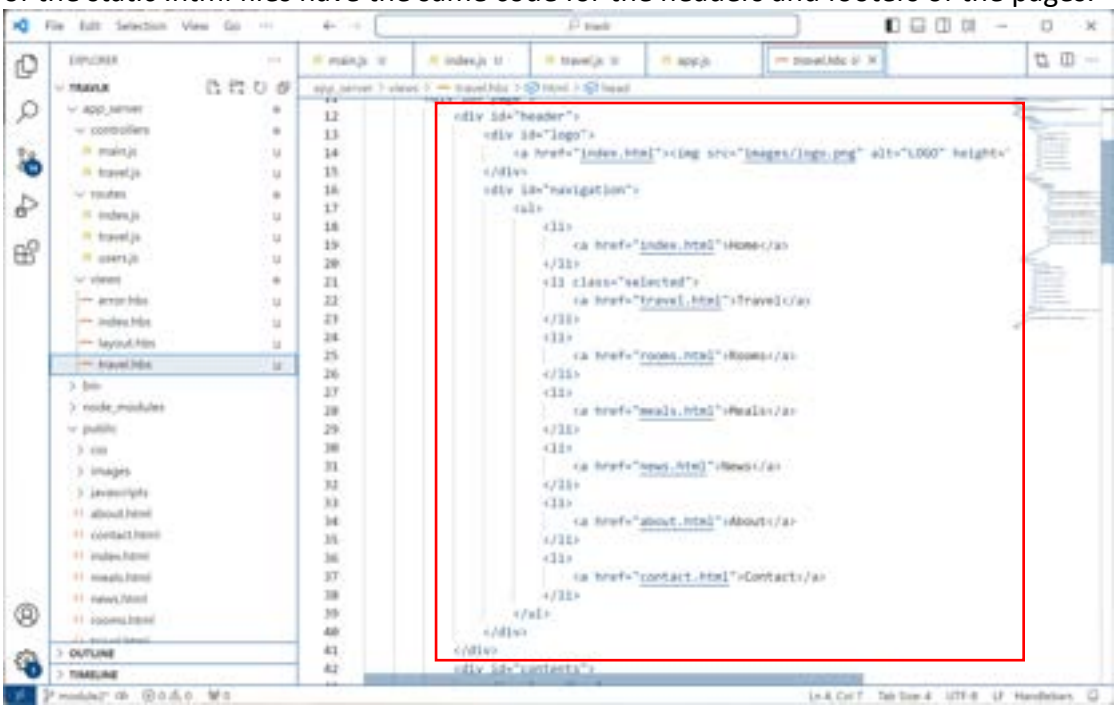
Creating Handlebars Views

The next step will be to create the ***handlebars views*** that are necessary to begin transitioning from static html to generated code.

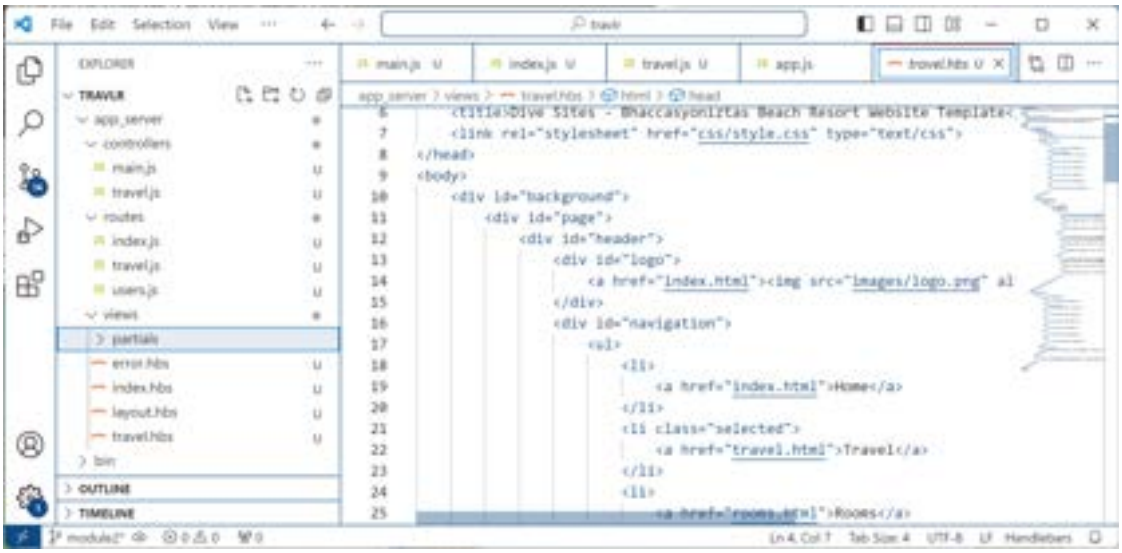
1. The first step in this process will involve taking the ***travel.html*** file found in the public folder of the travlr website and copying it to the ***app_server/views*** folder and changing the extension from .html to .hbs – a designation that will inform the express server engine that it will be rendered with the handlebars processor.



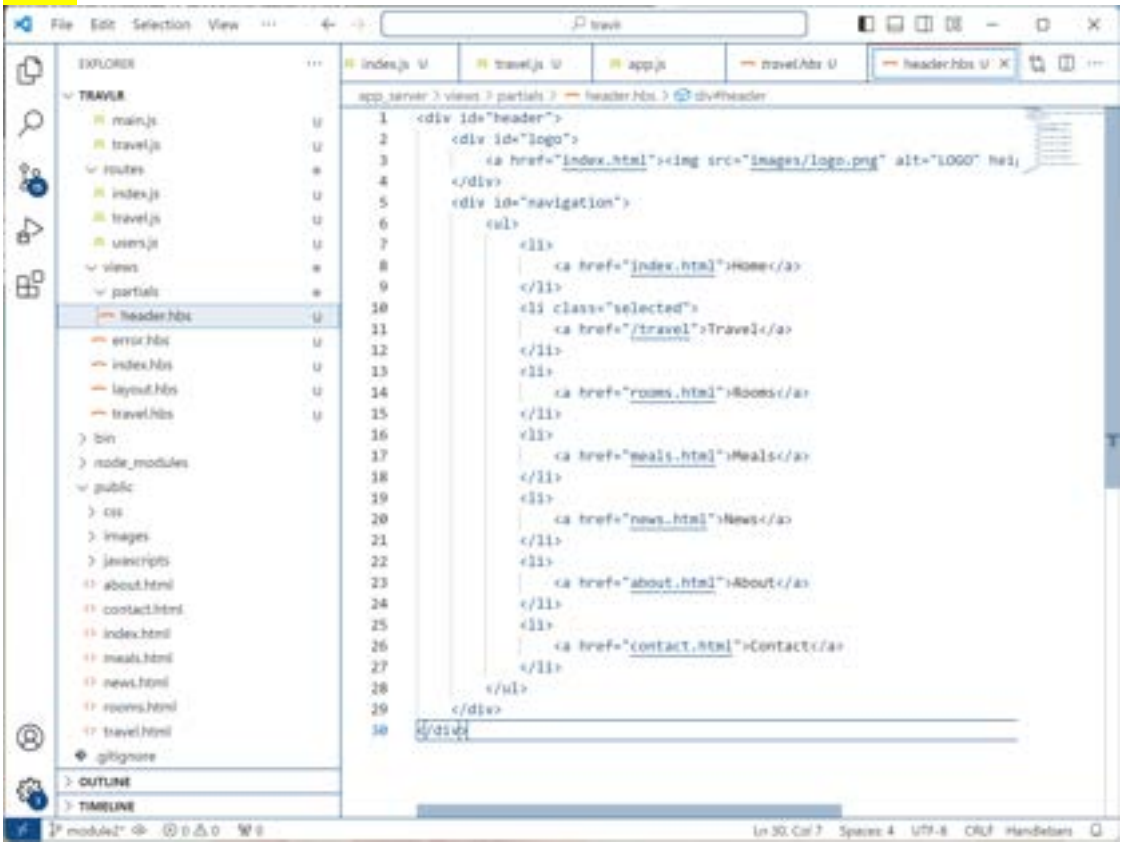
2. Upon examination of the ***travel.html*** file (now named ***travel.hbs***) you will notice that all of the static .html files have the same code for the headers and footers of the pages.



3. One of the features of the ***handlebars*** view engine is support for partial page fragments that hold common html to avoid duplication of code within your website. The eliminates the need to code the same redundant data on each page. To use this capability, we will first create a ***partials*** folder under the ***views*** folder in the website hierarchy.

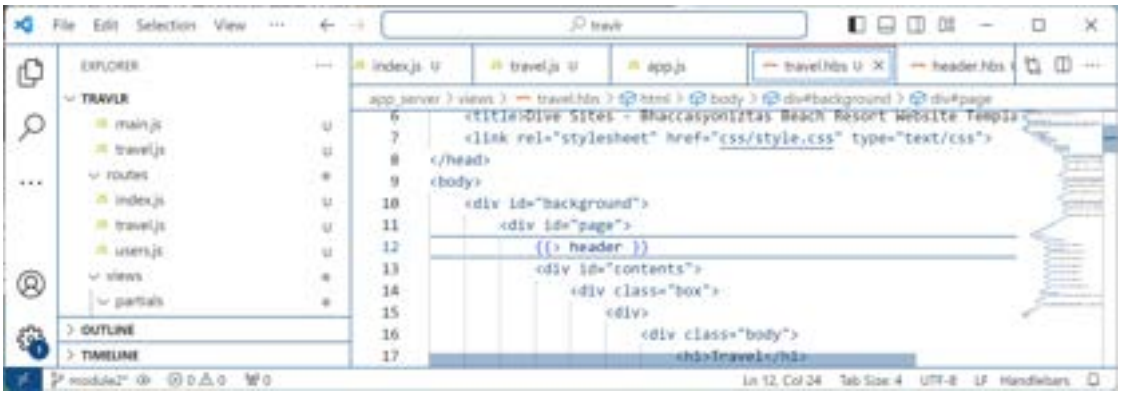


4. In the ***partials*** folder we will create a new file called ***header.hbs*** and we will populate that with the header block from the ***travel.hbs*** file. At the same time, replace the anchor reference ***travel.html*** with ***/travel*** in order to route the link through the handlebars processing engine. Don't forget to save the file when you are done making edits!

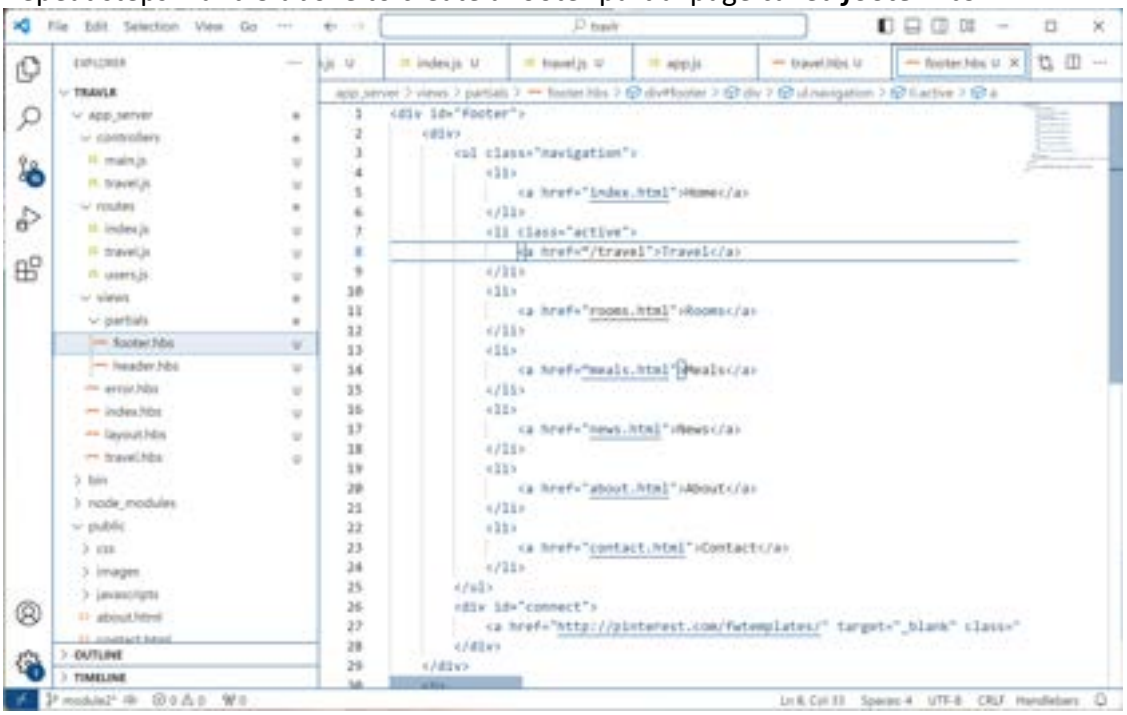


5. Handlebars uses double curly braces to distinguish its code blocks from ordinary HTML. The greater than symbol is an instruction to the templating engine to include the header view, specifically to replace the instruction tag in the ***traveler.hbs*** file with the contents of ***header.hbs***. To accomplish this, we will replace the entire block in the ***travelr.hbs*** file with a single tag referencing our new handlebars partial. (Don't forget to save the file when you are done making edits!)

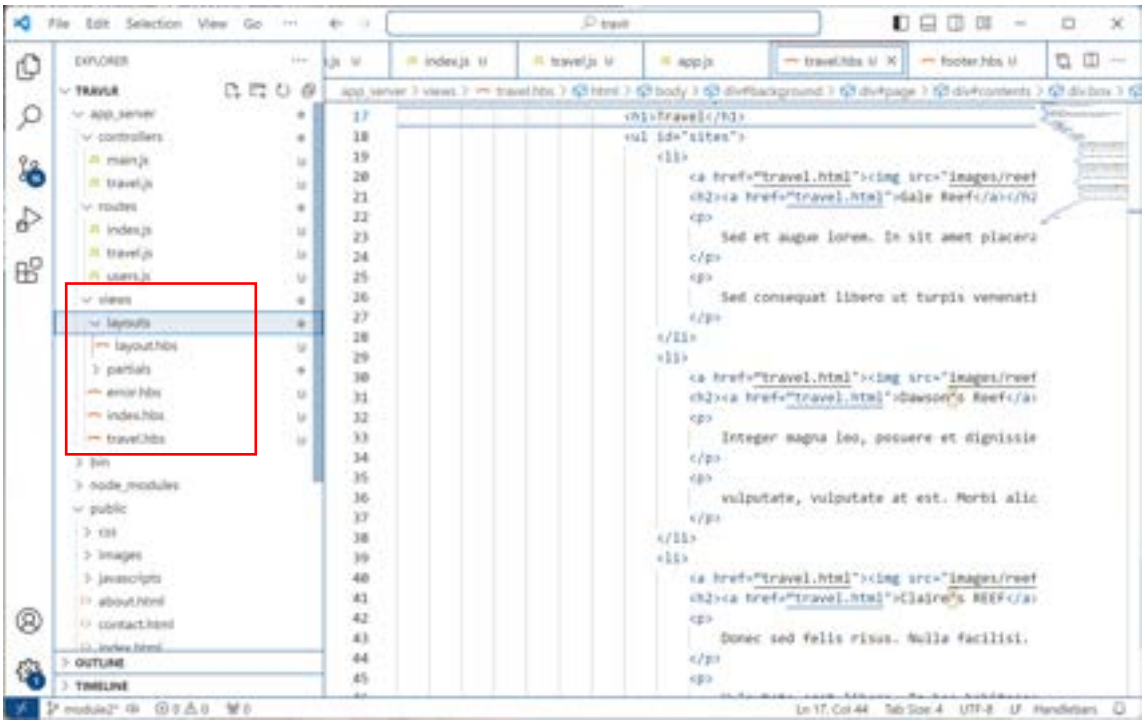
```
{{> header }}
```



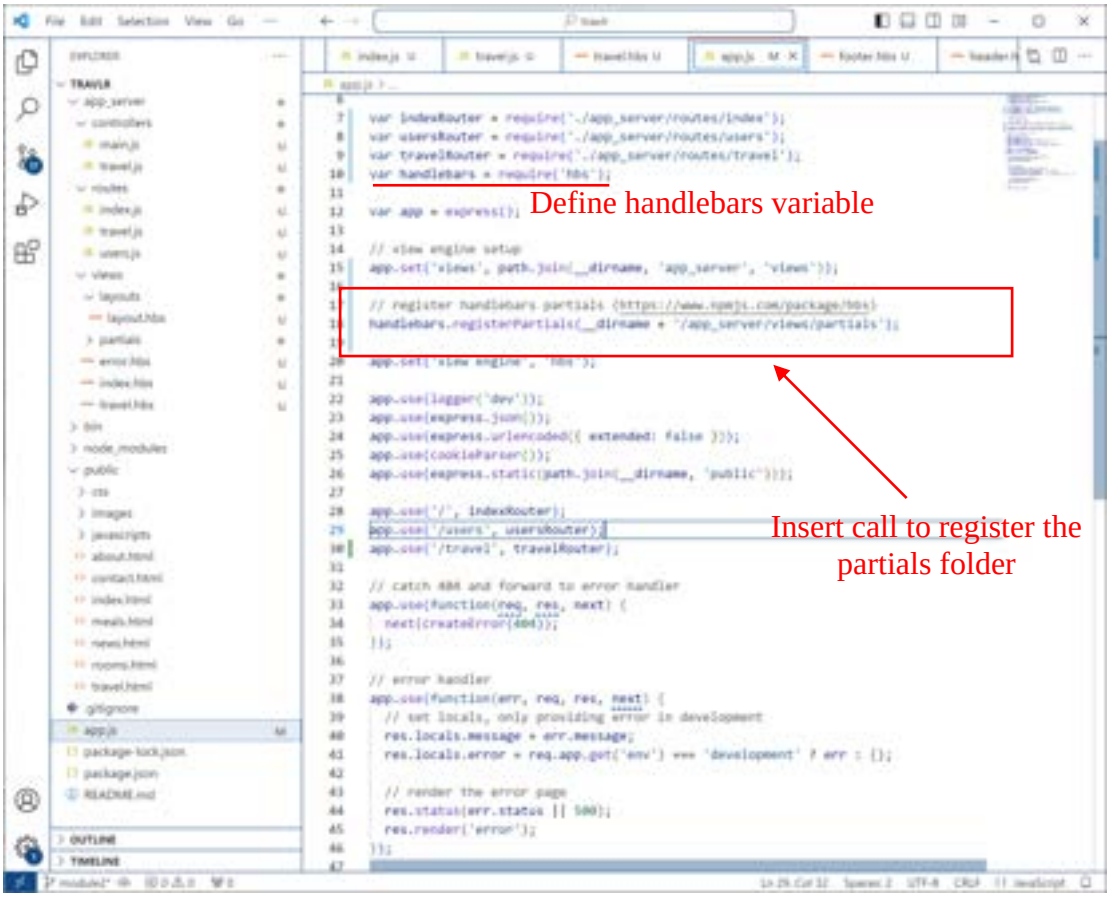
6. Repeat steps 4 and 5 above to create a footer partial page called ***footer.hbs*** .



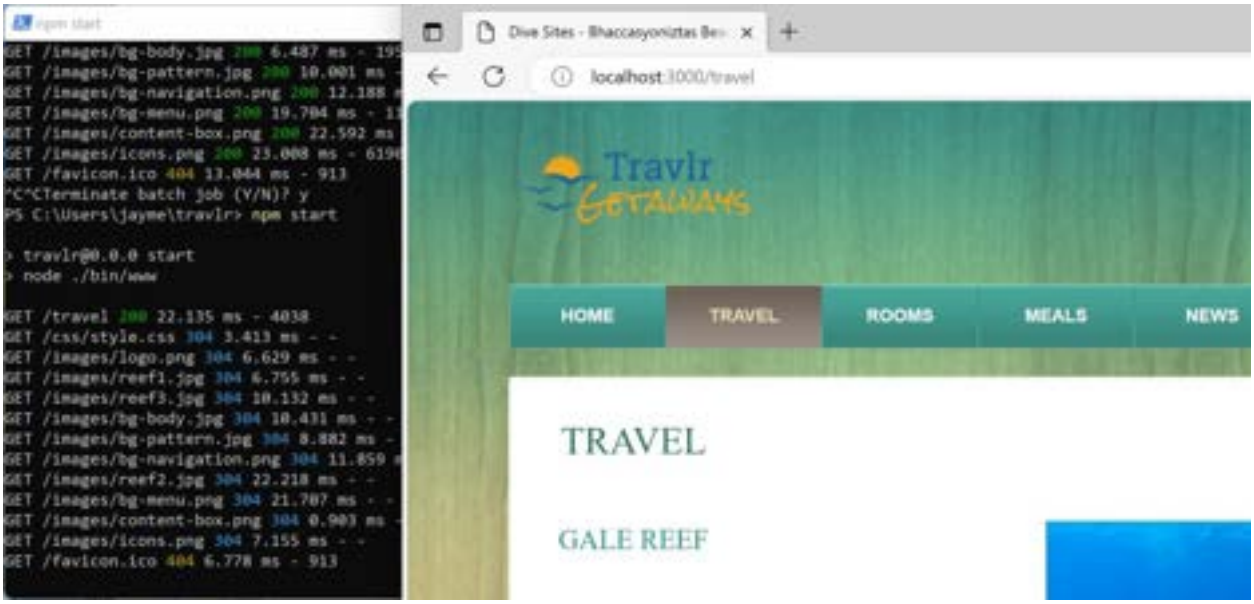
7. For the next step, we are going to create one more sub-folder underneath the views folder. This one will be named layouts. When the folder has been created, we want to move the ***layout.hbs*** file into that layouts folder.



8. The last step necessary to enable the appropriate rendering of these handlebars views is registering the partials directory with the templating engine. To do this, we need to once again edit the app.js file and add a small code block to register the partials directory:



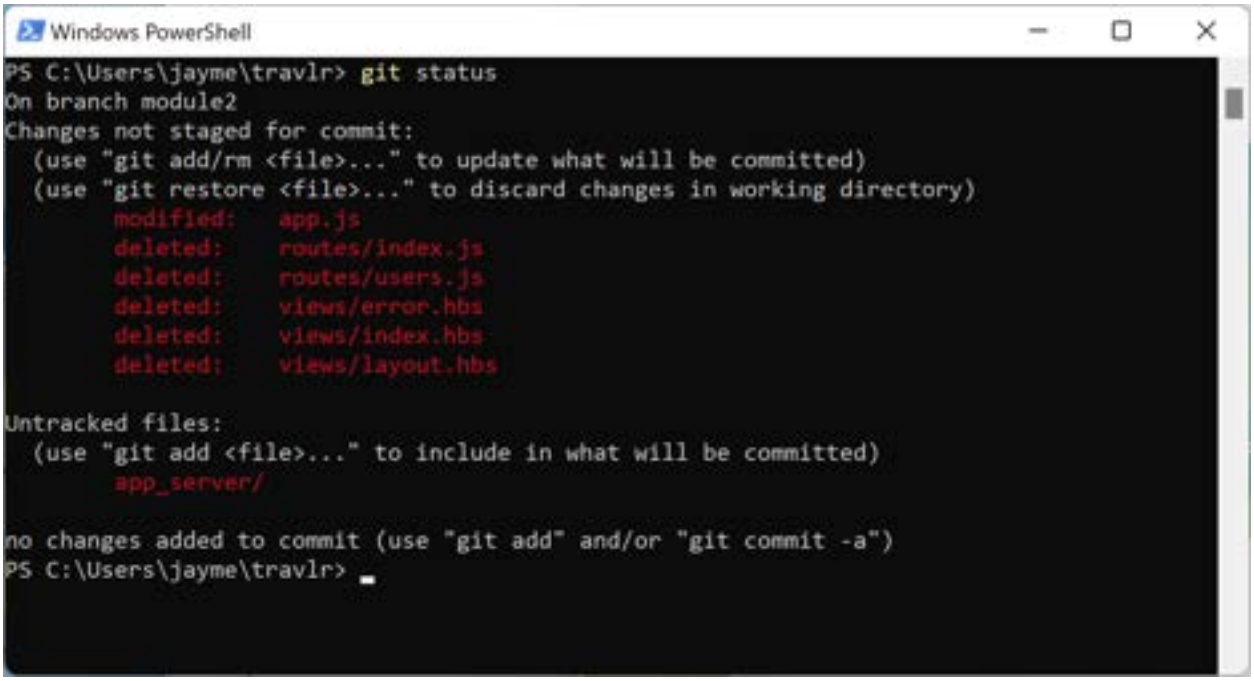
9. Now that we have made many changes, it is time to test this. Restart your webserver (npm start in the travlr sub-directory in PowerShell) and check the webpage.



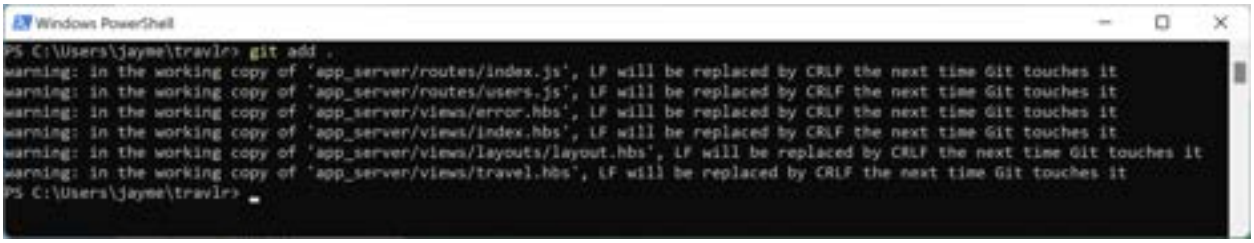
You will note that the webserver served up the dynamic content when we entered the specific URL. We needed to do this manually because we did not remove the ***travel.html*** file from the website, nor did we replace all of the ***travel.html*** references within the code.

Finalizing Module 2

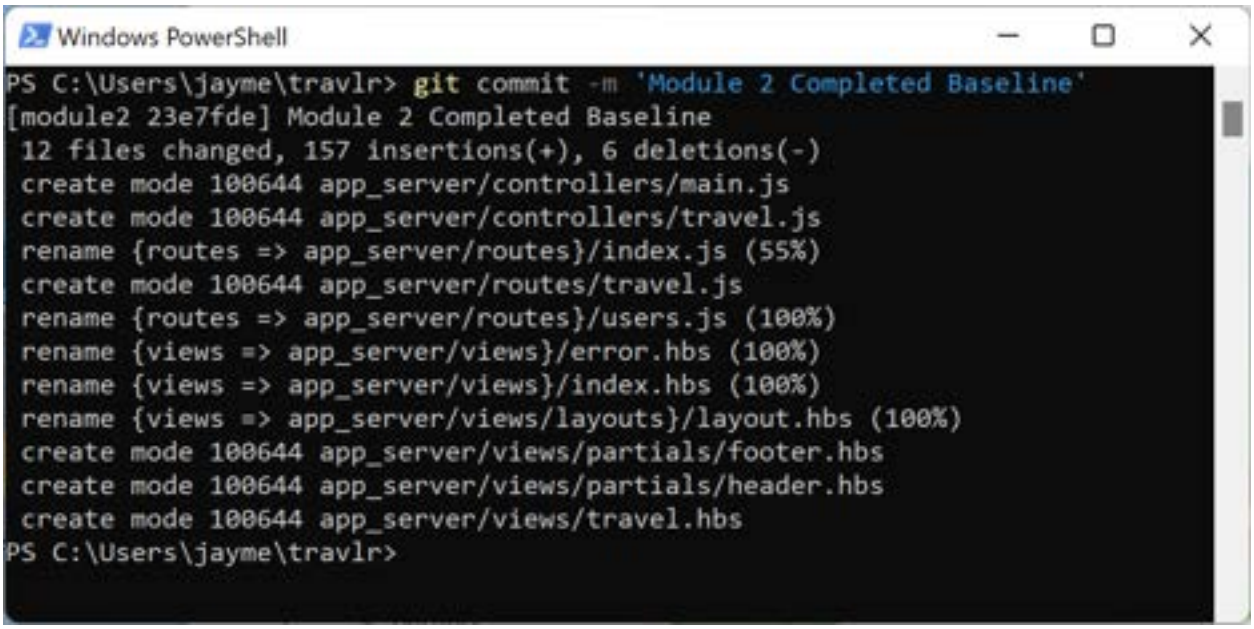
1. Now that we have completed Module 2, we go back to git and make sure that we add everything to tracking. We start by checking the status of what has changed (git status):



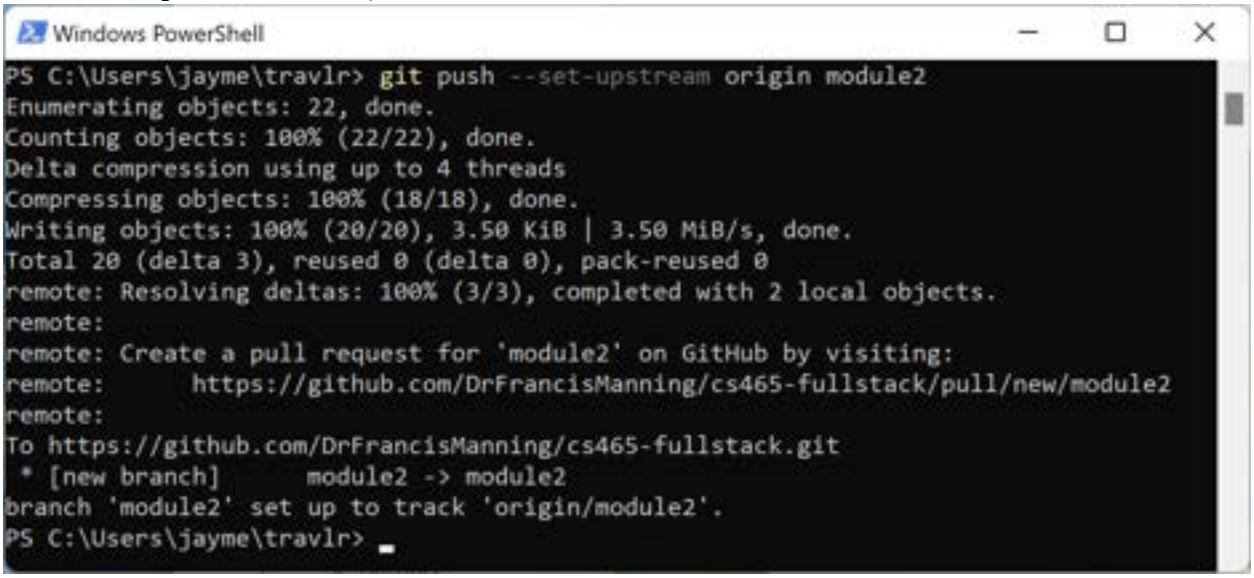
2. Then we add all of those changes into tracking (`git add .`):



3. Now we commit the changes (`git commit -m 'Module 2 completed baseline'`):



4. We push the changes back to GitHub for safekeeping (`git push --set-upstream origin module2`):



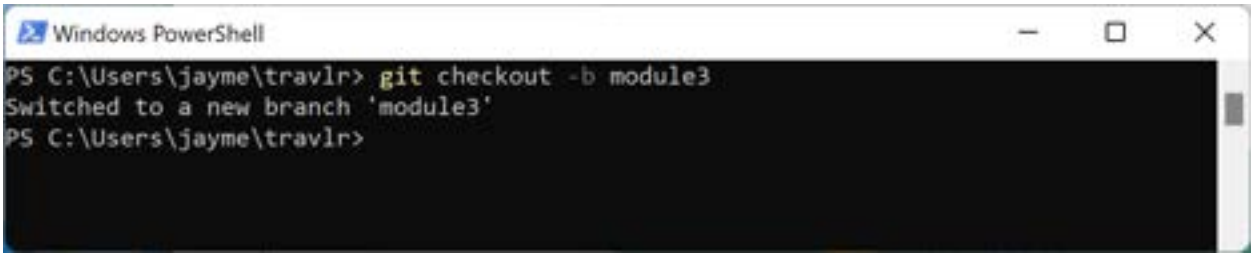
Module 3: Static HTML to Templates with JSON

In this module, we are going to replace static HTML with templates that will utilize JSON to format and display information.

Create Git Branch for Module 3

Before you begin, it is important to make sure that you have created your new branch in git for Module 3. To accomplish this, we will perform the following command in a PowerShell window in the travlr project directory:

```
git checkout -b module3
```



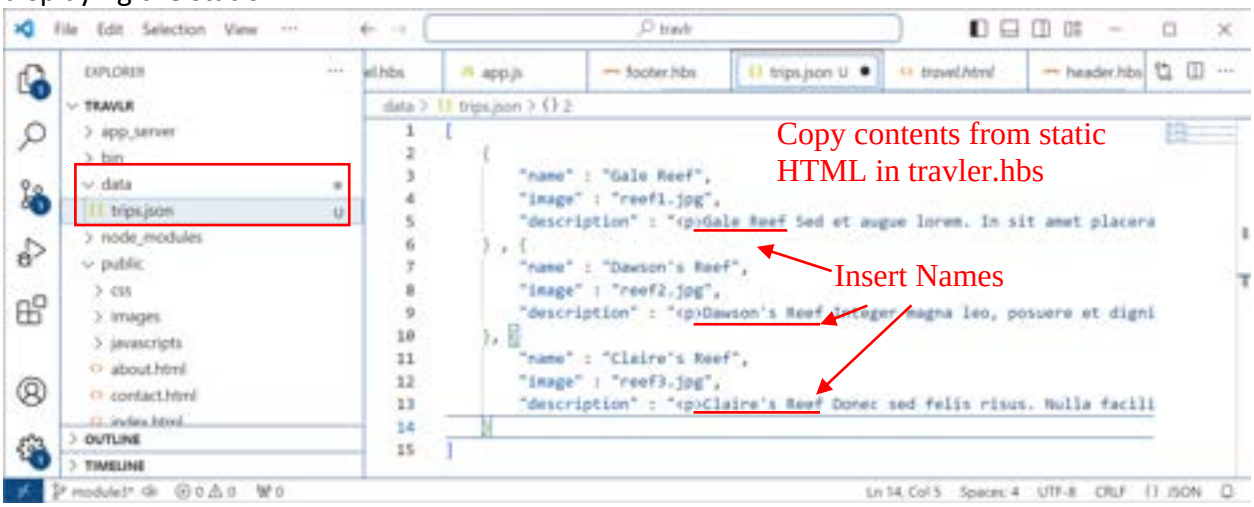
Replacing Static HTML with Templates

We are going to be working on the Trips displayed on the Travel page on the website. We have already started to update the raw travel.html page by turning it into a rendered page using handlebars. We have separated out the header and footer into partials and reduced the code in the remaining template. Now we are going to work on the content of the trips displayed on the page.

To accomplish this, we are going to replace the hard-coded HTML trip content with a loop that will render JSON data using handlebars directives.

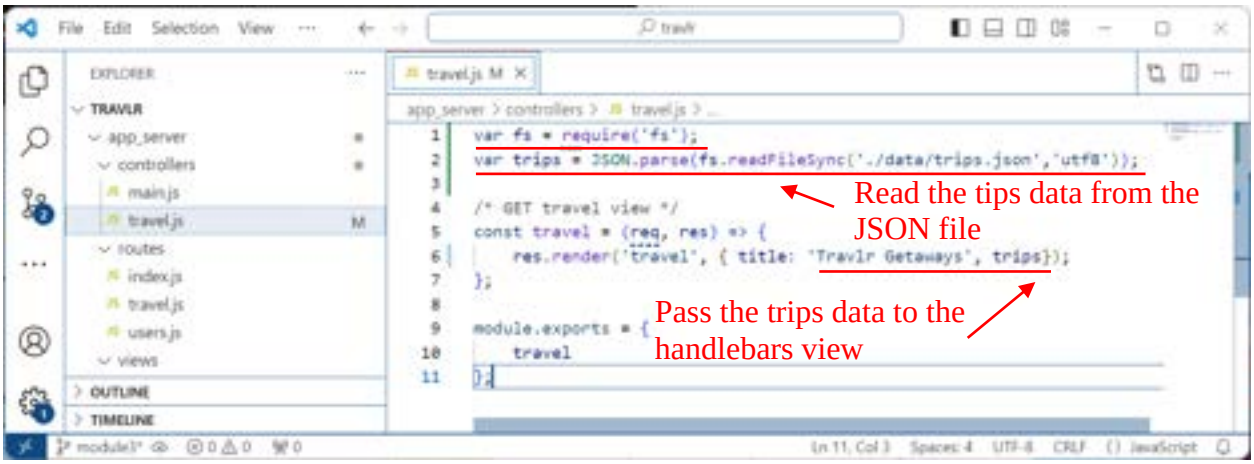
1. This is the display of one of the trips currently embedded as static HTML on the page:

2. To enable the transition to a handlebars loop, we will first begin by creating a ***data*** folder in the main travlr project directory. In the new data folder, we will create a ***trips.json*** file that will contain a JSON description of the trips for the purposes of testing. **Please Note:** We are asking you to add the name of the trip to the description line in each trip instance so that it is evident when the JSON data is being rendered instead of just displaying the static HTML.



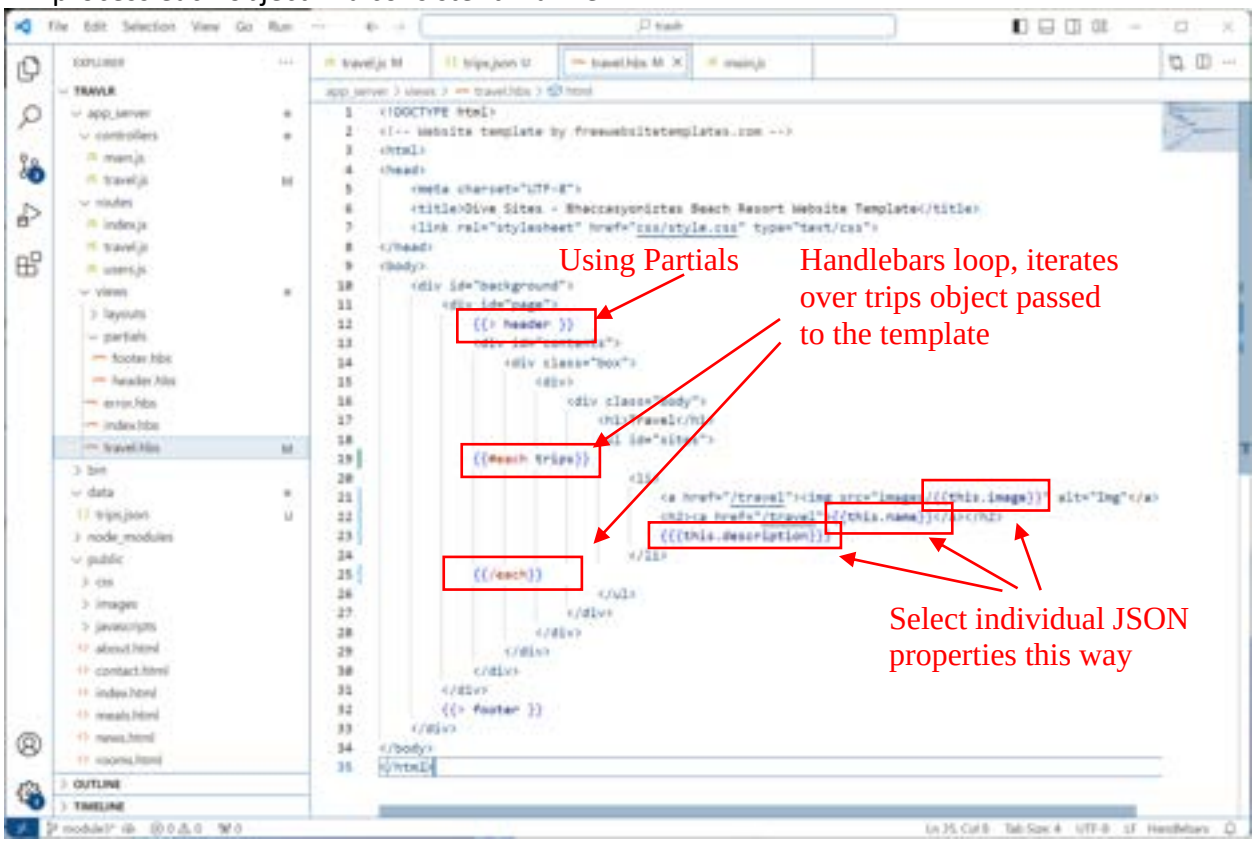
3. The next step will be to edit the ***travel.js*** controller file in order to use the built-in NodeJS filesystem component to read the data file that we just created. We will be using the fs.readFileSync() method to retrieve the JSON.

```
var fs = require('fs');
var trips = JSON.parse(fs.readFileSync('./data/trips.json',
'utf8'));
```

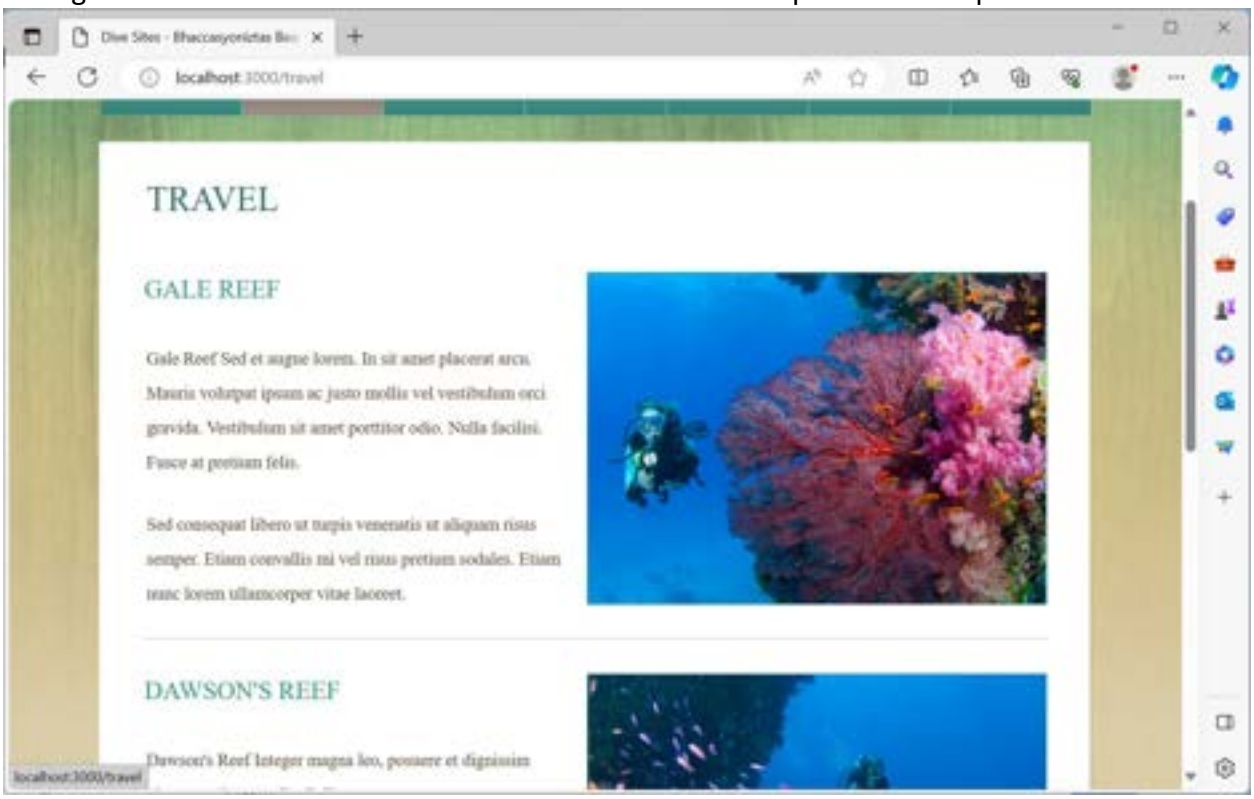


Please Note: It is ***not*** a best practice to read a JSON file every time the webserver processes a request. This is a method used during development to support rapid prototyping and should be replaced before the applications goes into production.

4. The final step in this process is to edit the travel.hbs template and replace the static HTML list entries for the trips with a {{#each trips}} {{/each}} directive. This is handlebars notation to create a loop for each object in the 'trips' data collection, allowing you to process each object in a consistent manner.



At this point, you can restart your webserver, and test to make sure that you are now seeing the rendered code which will have the name of the trip in the description as well.

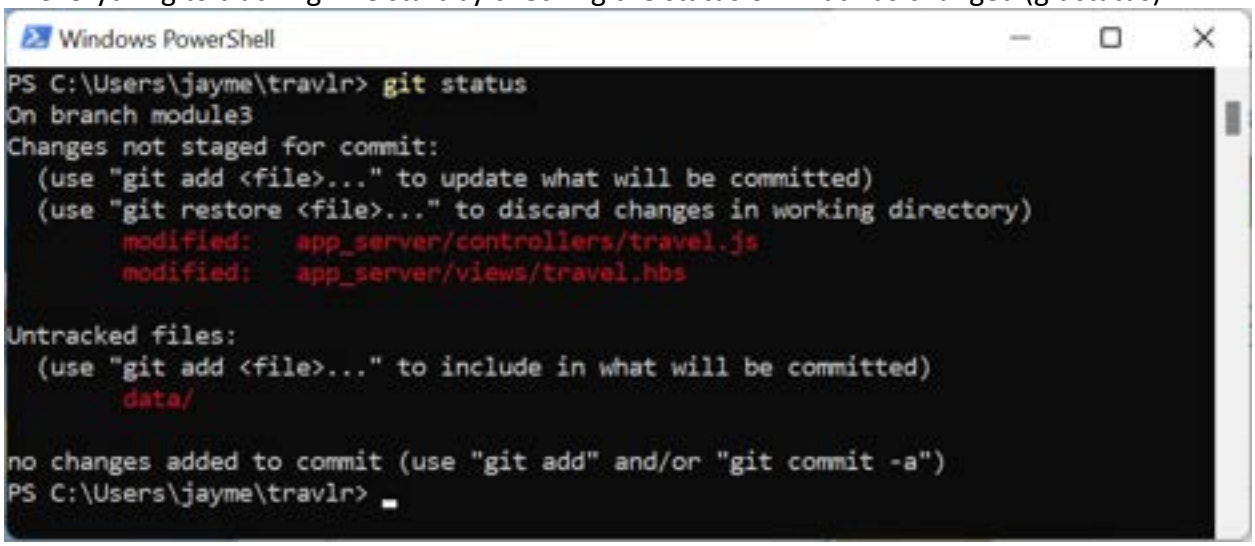


Your effort has replaced 120 lines of static HTML with 35 lines of code including 3 different handlebars directives that allow the rendered page to be driven dynamically with data passed in to the template.

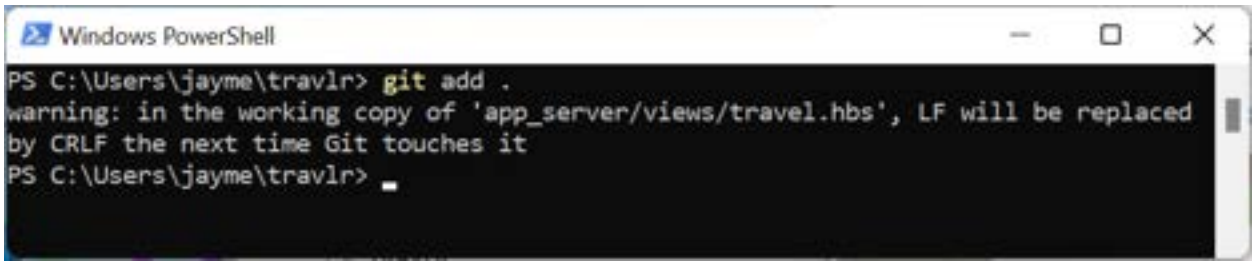
Optional Challenge: Using the techniques that you have just learned, repeat these steps to convert other static HTML pages to Handlebars templates, either with or without JSON data. This is your opportunity to experiment!

Finalizing Module 3

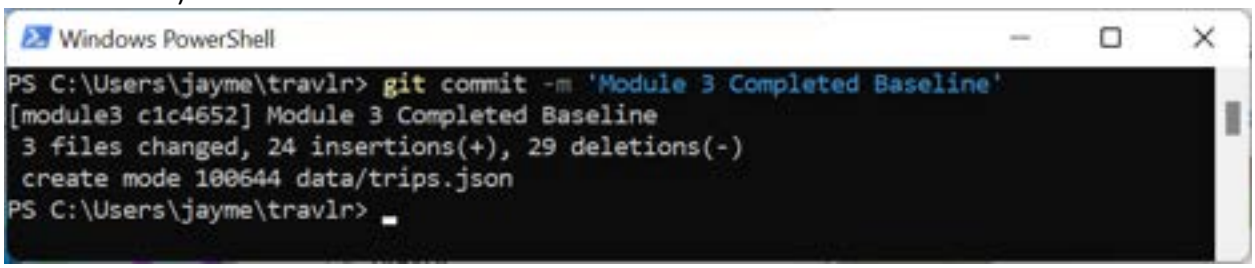
1. Now that we have completed Module 3, we go back to git and make sure that we add everything to tracking. We start by checking the status of what has changed (git status):



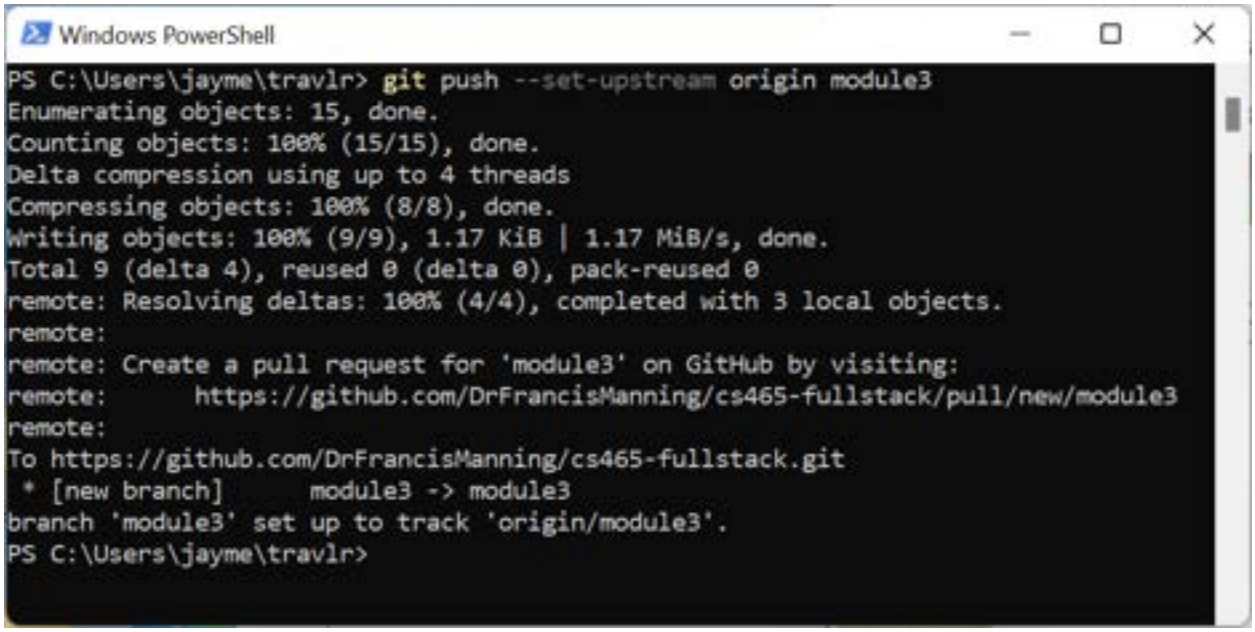
2. Then we add all of those changes into tracking (`git add .`):



3. Now we commit the changes (`git commit -m 'Module 3 completed baseline'`):



4. We push the changes back to GitHub for safekeeping (`git push --set-upstream origin module3`):


Windows PowerShell
PS C:\Users\jayme\travlr> git push --set-upstream origin module3
Enumerating objects: 15, done.
Counting objects: 100% (15/15), done.
Delta compression using up to 4 threads
Compressing objects: 100% (8/8), done.
Writing objects: 100% (9/9), 1.17 KiB | 1.17 MiB/s, done.
Total 9 (delta 4), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (4/4), completed with 3 local objects.
remote:
remote: Create a pull request for 'module3' on GitHub by visiting:
remote: https://github.com/DrFrancisManning/cs465-fullstack/pull/new/module3
remote:
To https://github.com/DrFrancisManning/cs465-fullstack.git
* [new branch] module3 -> module3
branch 'module3' set up to track 'origin/module3'.
PS C:\Users\jayme\travlr>

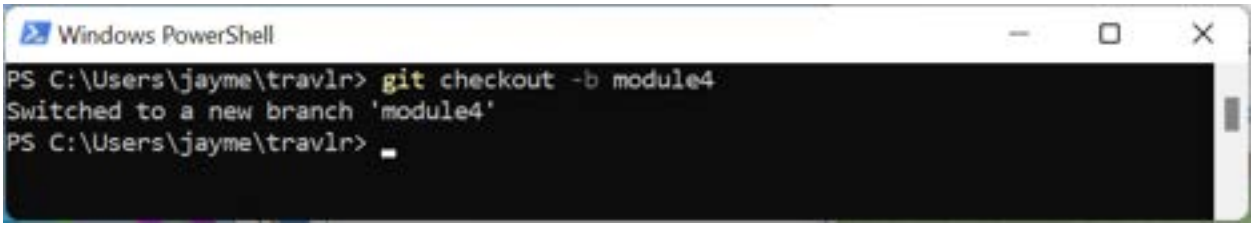
Module 4: NoSQL Databases, Models, and Schemas

In this module, we are going to take the next step and integrate our code with MongoDB, a NoSQL database. This will be much more efficient for storing JSON documents than trying to read them from the file-system every time a request is made to the database.

Create Git Branch for Module 4

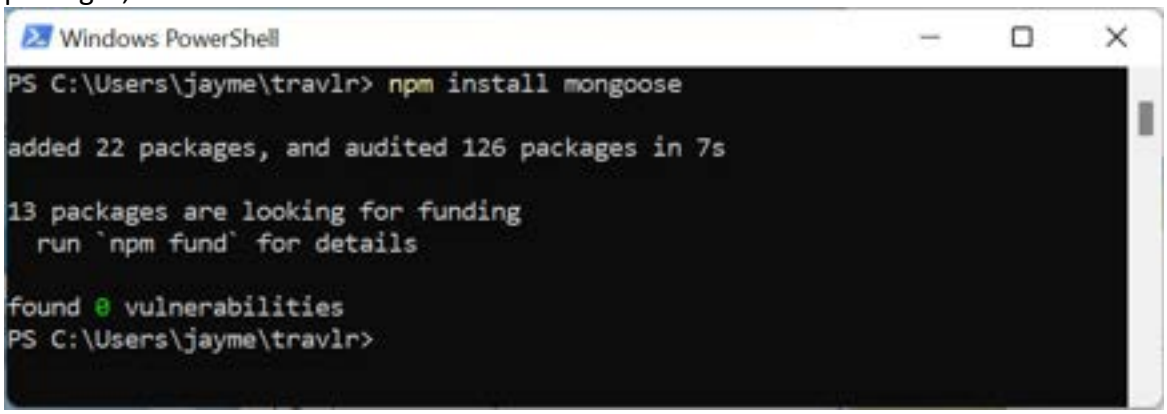
Before you begin, it is important to make sure that you have created your new branch in git for Module 4. To accomplish this, we will perform the following command in a PowerShell window in the travlr project directory:

```
git checkout -b module4
```

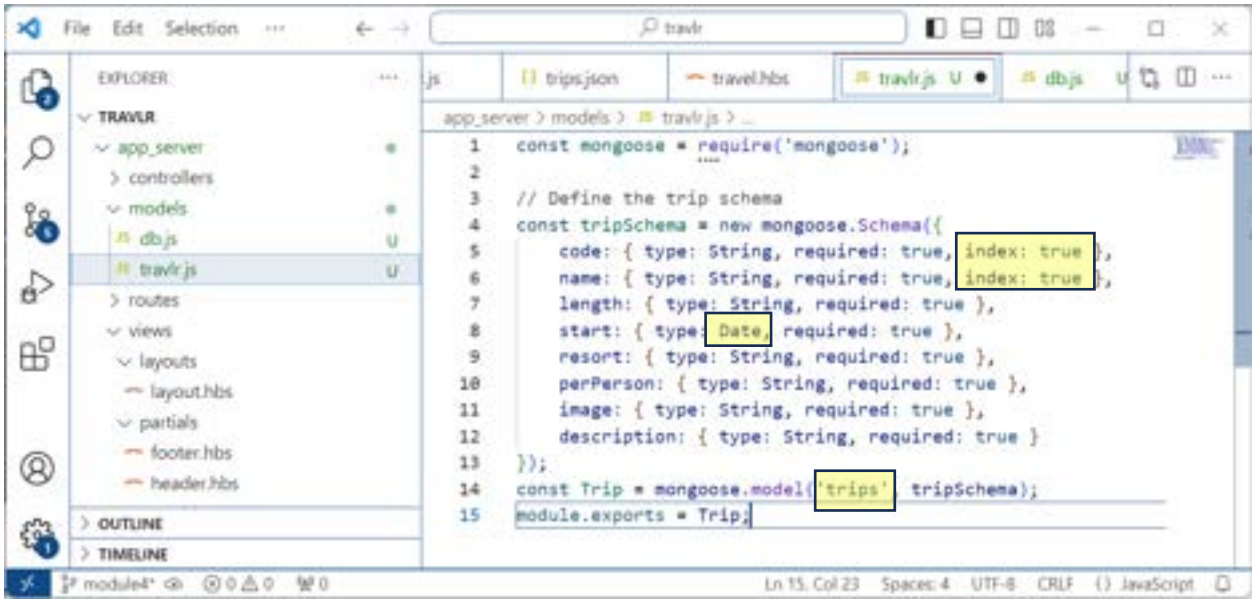


Install and Configure Mongoose

1. Begin by installing Mongoose. Mongoose is the NodeJS package that enables interaction with a MongoDB database. This will be installed similarly to any of the other NodeJS packages, via NPM.



2. Now we will create a new ***models*** folder underneath the ***app_server*** folder for our application. This is where we will create a module that will hold the schema for our trips. This module will be named ***travlr.js***.



Please Note the highlighted portion of the previous image. The trip code and name will be indexed in MongoDB for faster retrieval. The start date will be stored using the ISO standard date format, and the collection will be named 'trips'.

The code for the schema is presented here:

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

// Define the trip schema
const tripSchema = new mongoose.Schema({
    code: { type: String, required: true, index: true },
    name: { type: String, required: true, index: true },
    length: { type: String, required: true },
    start: { type: Date, required: true },
    resort: { type: String, required: true },
    perPerson: { type: String, required: true },
    image: { type: String, required: true },
    description: { type: String, required: true }
});
const Trip = mongoose.model('trips', tripSchema);
module.exports = Trip;
```

3. We can use the db.js module from Chapter 5, section 1 of your textbook with only minor changes for our project. The code is reflected here:

```
const mongoose = require('mongoose');
const host = process.env.DB_HOST || '127.0.0.1';
const dbURI = `mongodb://${host}/travlr`;
```

```
const readLine = require('readline');

// Build the connection string and set the connection timeout.
// timeout is in milliseconds.
const connect = () => {
    setTimeout(() => mongoose.connect(dbURI, {
    }), 1000);
}

// Monitor connection events
mongoose.connection.on('connected', () => {
    console.log(`Mongoose connected to ${dbURI}`);
});

mongoose.connection.on('error', err => {
    console.log('Mongoose connection error: ', err);
});

mongoose.connection.on('disconnected', () => {
    console.log('Mongoose disconnected');
});

// Windows specific listener
if(process.platform === 'win32'){
    const r1 = readLine.createInterface({
        input: process.stdin,
        output: process.stdout
    });
    r1.on('SIGINT', () => {
        process.emit("SIGINT");
    });
}

// Configure for Graceful Shutdown
const gracefulShutdown = (msg) => {
    mongoose.connection.close(() => {
        console.log(`Mongoose disconnected through ${msg}`);
    });
};

// Event Listeners to process graceful shutdowns

// Shutdown invoked by nodemon signal
process.once('SIGUSR2', () => {
    gracefulShutdown('nodemon restart');
```

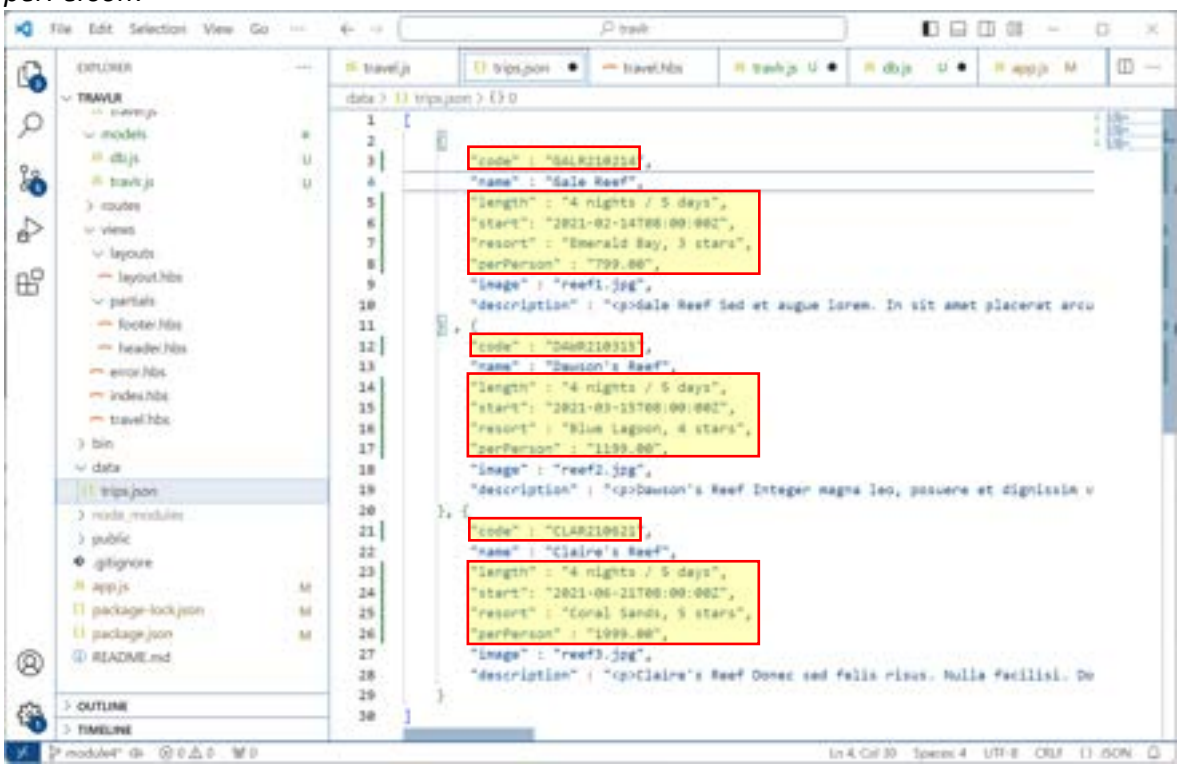
```
    process.kill(process.pid, 'SIGUSR2');
});

// Shutdown invoked by app termination
process.on('SIGINT', () => {
    gracefulShutdown('app termination');
    process.exit(0);
});


// Shutdown invoked by container termination
process.on('SIGTERM', () => {
    gracefulShutdown('app shutdown');
    process.exit(0);
});

// Make initial connection to DB
connect();

// Import Mongoose schema
require('./travlr');
module.exports = mongoose;
```

Seeding the Database

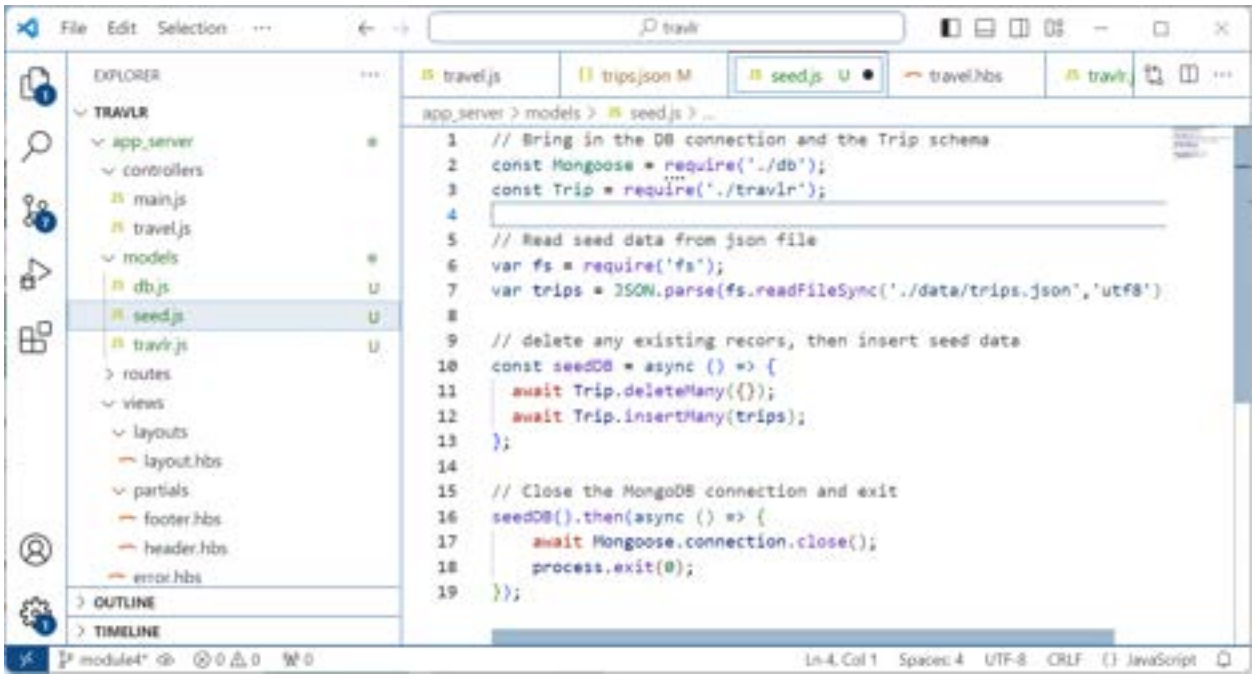
The next step in the connection of our application to the database is putting some seed data into the Database. There are multiple ways that we can accomplish this – you can use Mongo Compass to create the database, the collection, and add seed data, and you could do the same in DBeaver. However, we are going to take a different route and seed the database by leveraging some of the code that we have already built and creating a small Node JS script to insert data into the database.

1. The first step we need to take is to adjust the ***trips.json*** data file that we created earlier in order to provide the additional properties that we identified for our schema. In this case, we are going to add key-value pairs for the new attributes: *code, length, start, resort, and*

perPerson.

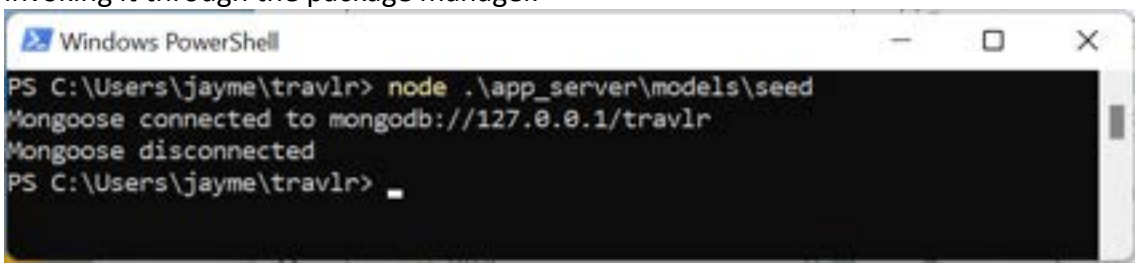


2. Next, we need to create a new file in the ***models*** directory in the ***travlr*** application named ***seed.js***. This will be the script that we utilize to enter the seed data into our MongoDB instance.

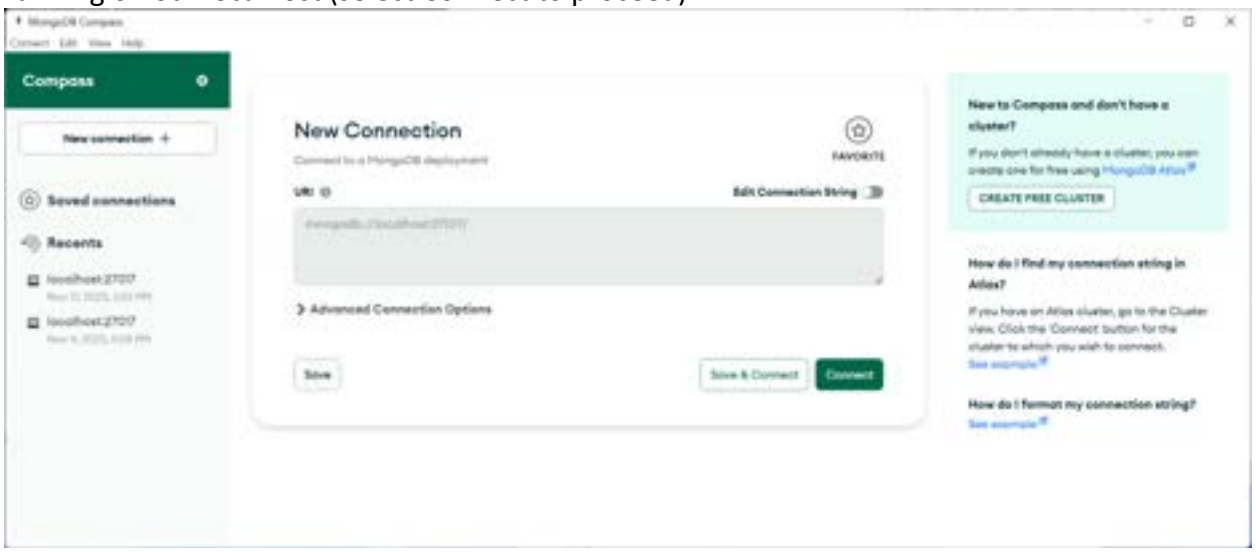


Using this method to seed the database, we will remove any existing records each time we re-run the seed script. You can modify the number of records seeded by editing the ***trips.json*** file.

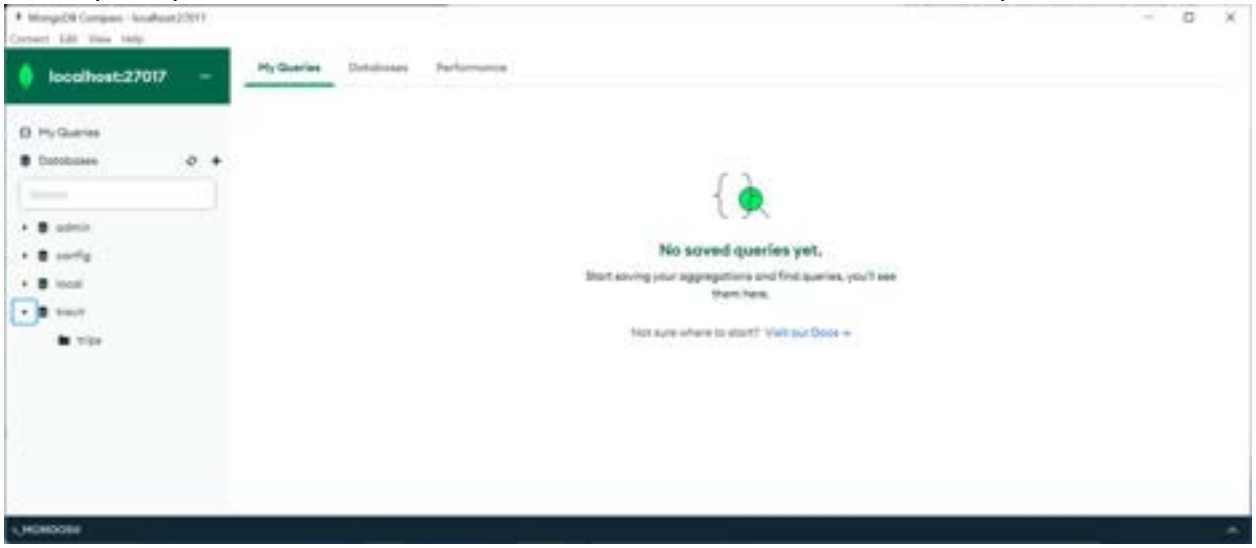
3. Now that we have the seed script created, we will execute the script. The execution is a little bit different than what we have used before because we will be using node directly, and not invoking it through the package manager.



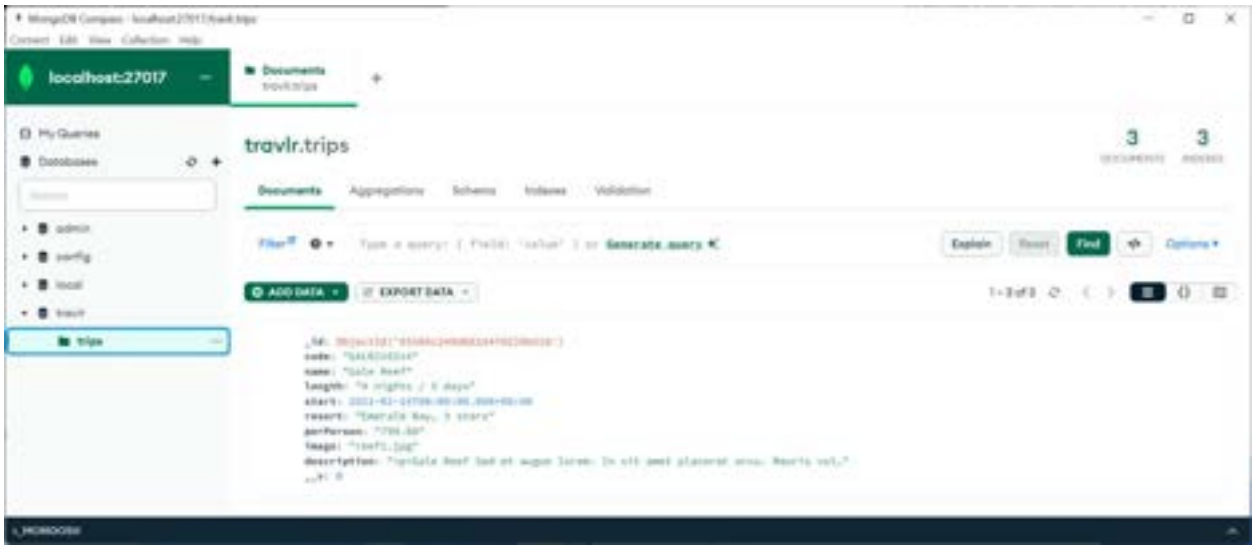
4. The next step is to use Mongo Compass or DBeaver to verify that the data has been loaded in the database. Using Mongo Compass, we will connect to the instance of MongoDB running on our localhost (select Connect to proceed):



5. At this point you should see that the ***travlr*** database exists and contains a 'trips' collection:

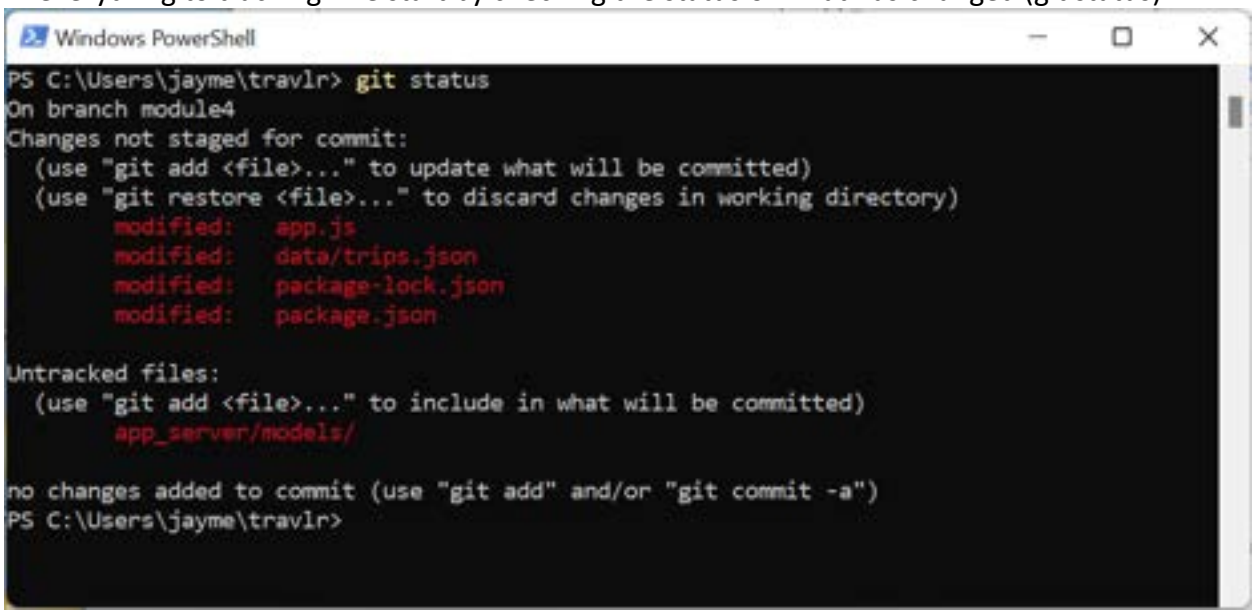


6. Selecting the '***trips***' collection will allow you to see that three records have been added to the database.



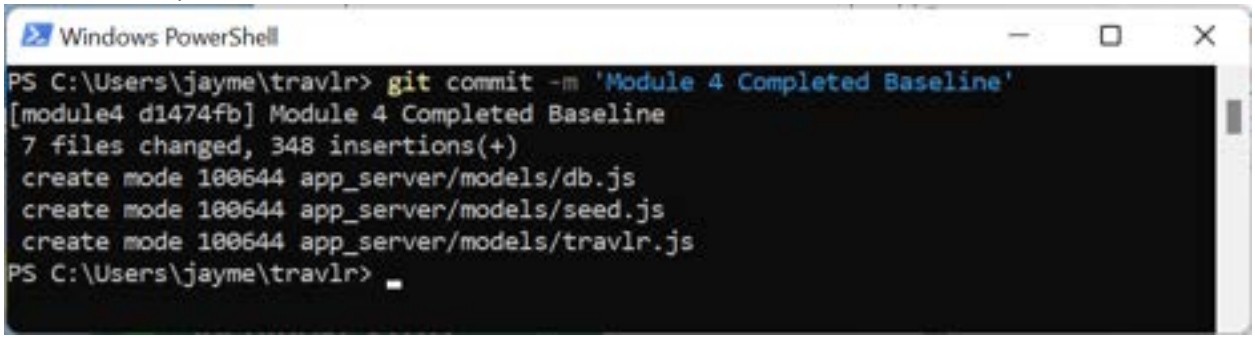
Finalizing Module 4

1. Now that we have completed Module 4, we go back to git and make sure that we add everything to tracking. We start by checking the status of what has changed (git status):

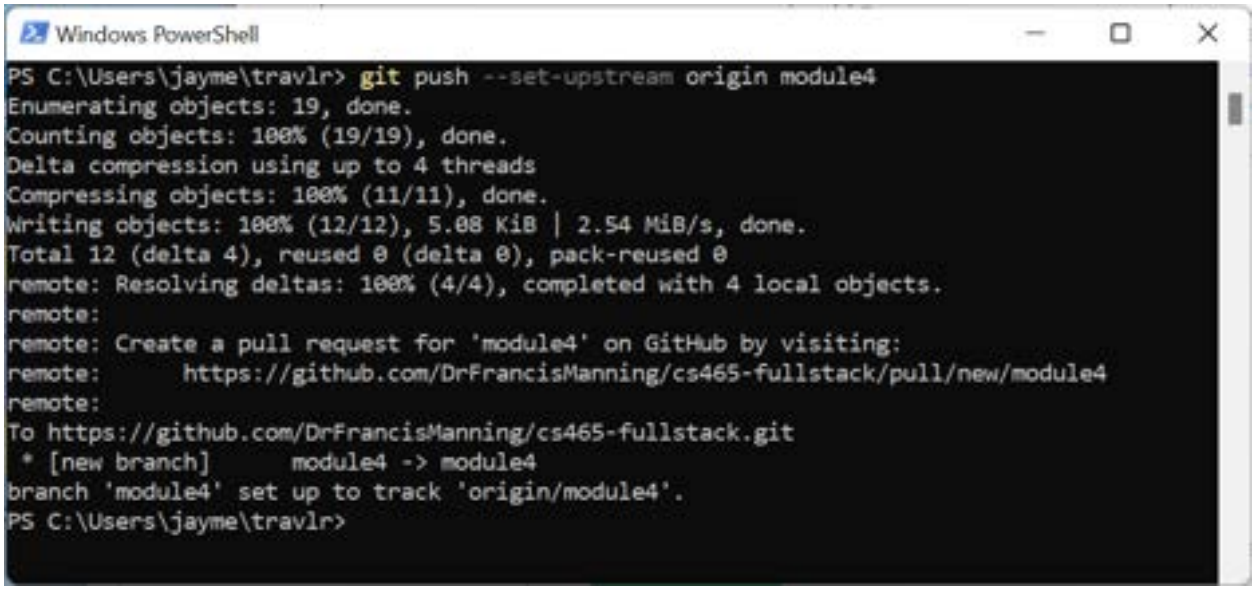


2. Then we add all of those changes into tracking (`git add .`):

3. Now we commit the changes (`git commit -m 'Module 4 completed baseline'`):



4. We push the changes back to GitHub for safekeeping (`git push --set-upstream origin module4`):


Windows PowerShell
PS C:\Users\jayme\travlr> git push --set-upstream origin module4
Enumerating objects: 19, done.
Counting objects: 100% (19/19), done.
Delta compression using up to 4 threads
Compressing objects: 100% (11/11), done.
Writing objects: 100% (12/12), 5.08 KiB | 2.54 MiB/s, done.
Total 12 (delta 4), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (4/4), completed with 4 local objects.
remote:
remote: Create a pull request for 'module4' on GitHub by visiting:
remote: https://github.com/DrFrancisManning/cs465-fullstack/pull/new/module4
remote:
To https://github.com/DrFrancisManning/cs465-fullstack.git
* [new branch] module4 -> module4
branch 'module4' set up to track 'origin/module4'.
PS C:\Users\jayme\travlr>

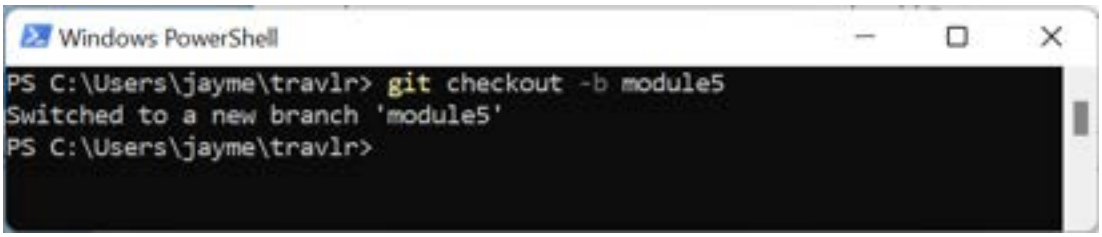
Module 5: RESTful API

In this module, we are going to take the next step and begin creating RESTful API endpoints for our code while also integrating with MongoDB. The purpose of this is to setup for Module 6 when we transition the application into a Single-Page Application (SPA) that utilizes the API calls to fetch and manipulate data.

Create Git Branch for Module 5

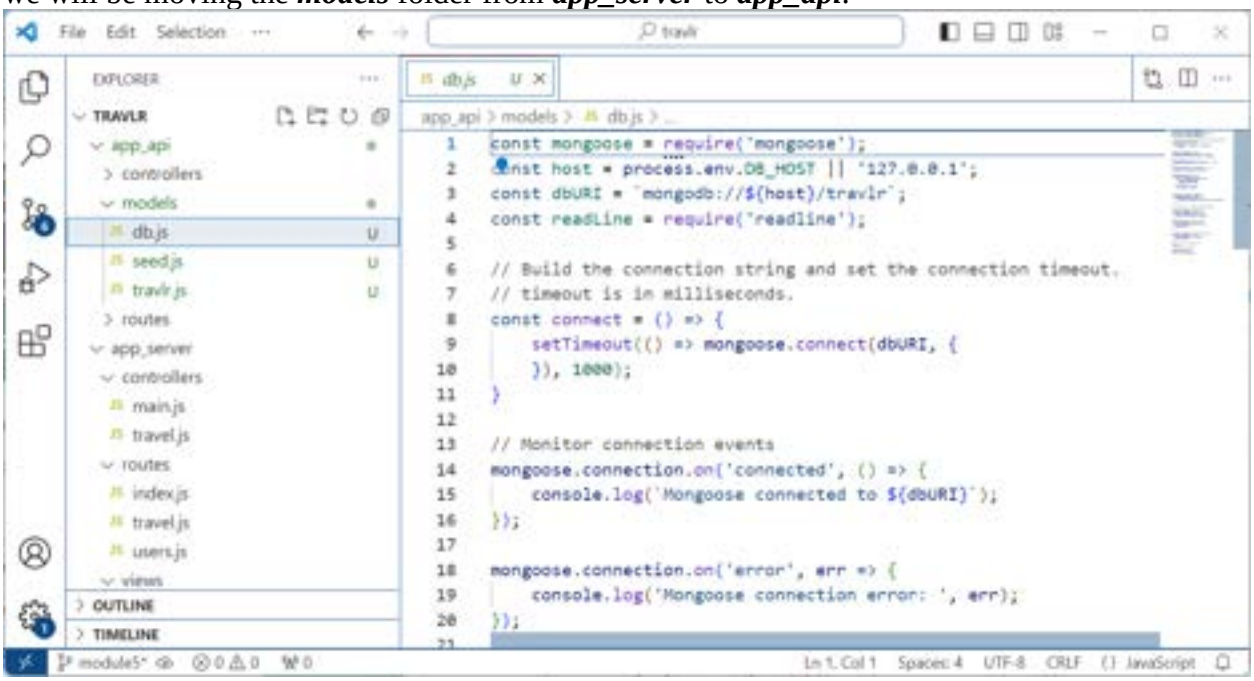
Before you begin, it is important to make sure that you have created your new branch in git for Module 5. To accomplish this, we will perform the following command in a PowerShell window in the ***travlr*** project directory:

```
git checkout -b module5
```



Creating the Structure for a RESTful API

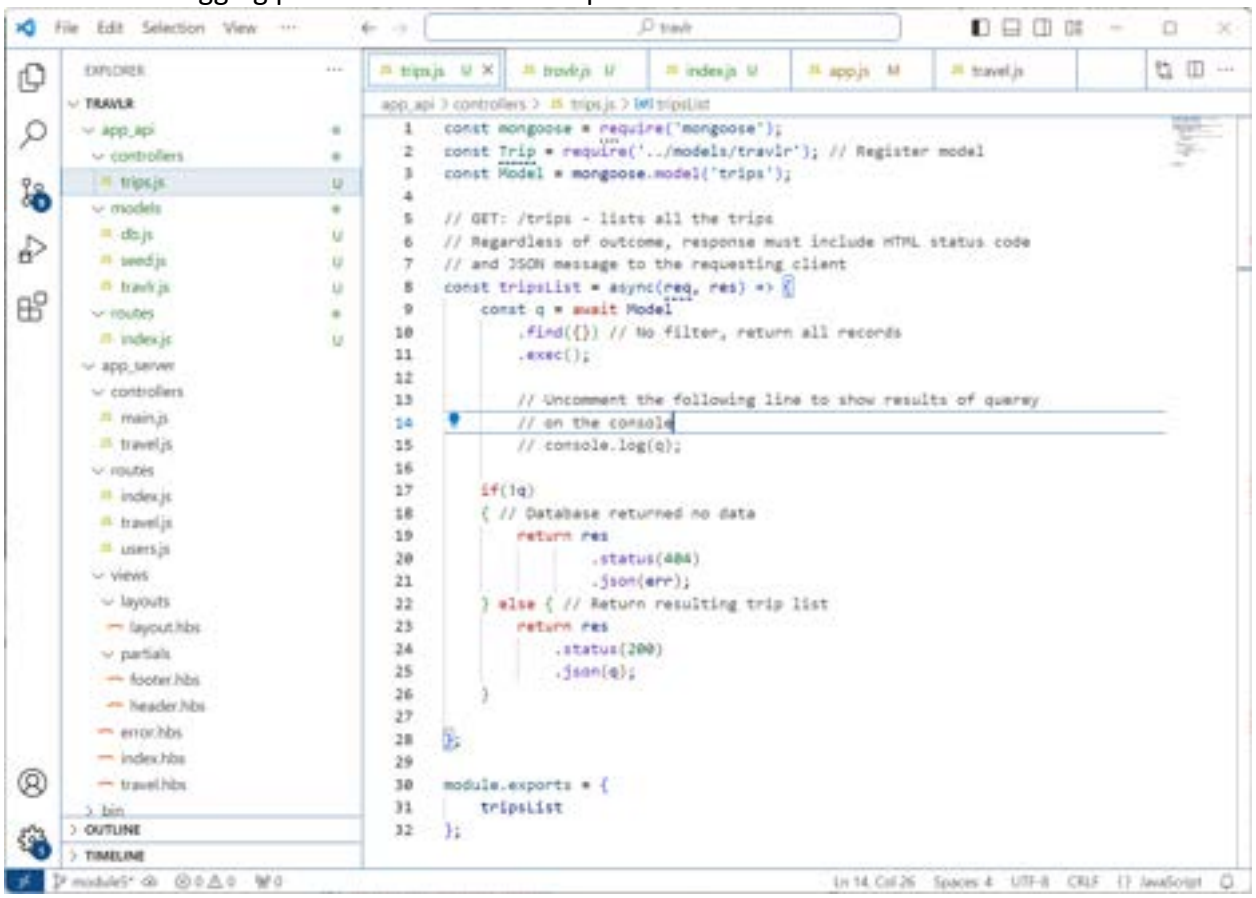
We will begin this step by creating a top-level folder in our ***travlr*** project directory named ***app_api***. In this folder, we will be creating two sub-folders: ***controllers*** and ***routes***. These will serve largely the same purpose as the corresponding folders in the ***app_server*** tree. Additionally, we will be moving the ***models*** folder from ***app_server*** to ***app_api***.



Creating the Mongoose Schema and DB Access Code

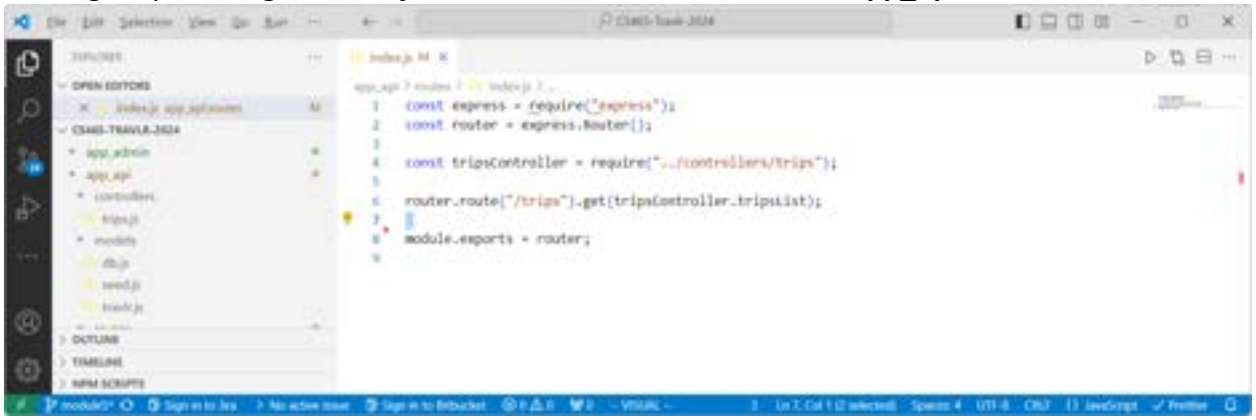
The next step in this process is to create an initial controller that can be utilized to extract trip information from the ***travlr*** database in MongoDB. Once we have the controller built and add the necessary routes to our application, we can perform some preliminary testing and then refactor our web-site to pull data from the database instead of the JSON file.

1. Begin by creating a file named ***trips.js*** in the ***controllers*** folder beneath ***app_api***. This will be where we create the first methods that will integrate with MongoDB to retrieve data for our application. We will start by building a single controller that will retrieve a list of all available trips. While we will be adding additional controllers, we are starting with one controller to make the debugging process easier to accomplish.

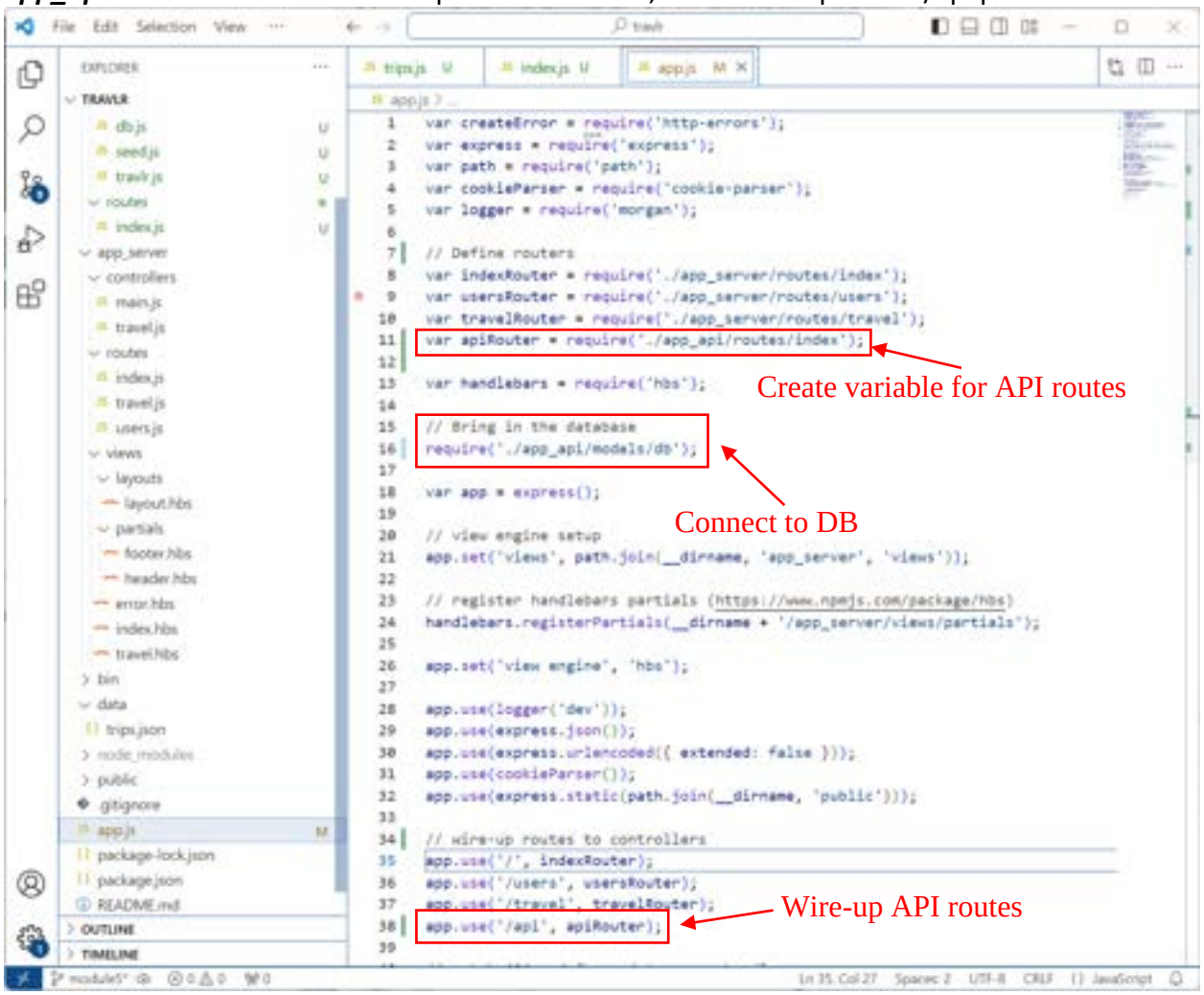


2. As we have already learned, in order activate a controller we need to add a route to the controller so that webserver will know how to process a specific request. In this step, we are going to add a router for our API work. We are going to simplify how we are approaching the routing because there are likely to be additional API endpoints that we will want to enable.

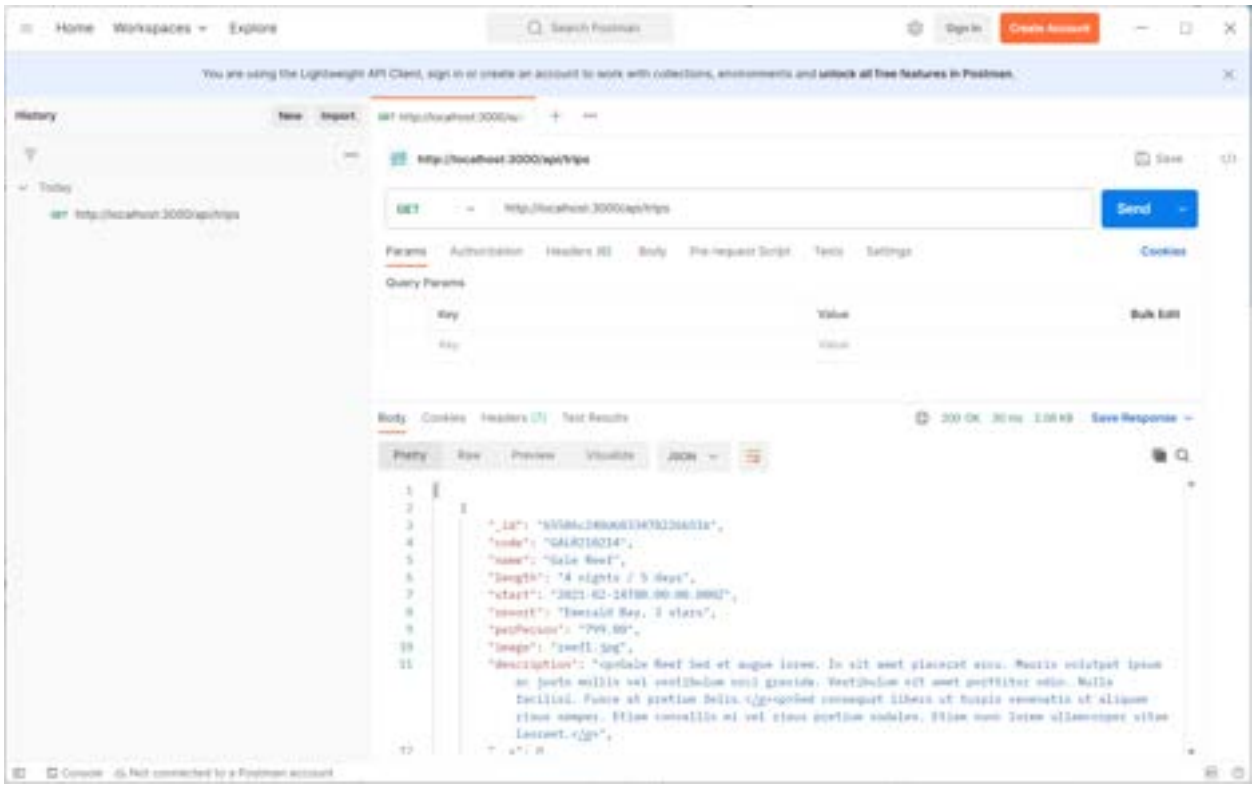
We begin by creating an ***index.js*** file in the ***routes*** folder beneath ***app_api***.



3. Now that we have the routes, we need to adjust the ***app.js*** to pull the necessary items into our application. We need to repoint the ***db*** module because we moved it underneath ***app_api***. We will then define our apiRouter variable, and wire it up to the /api path.



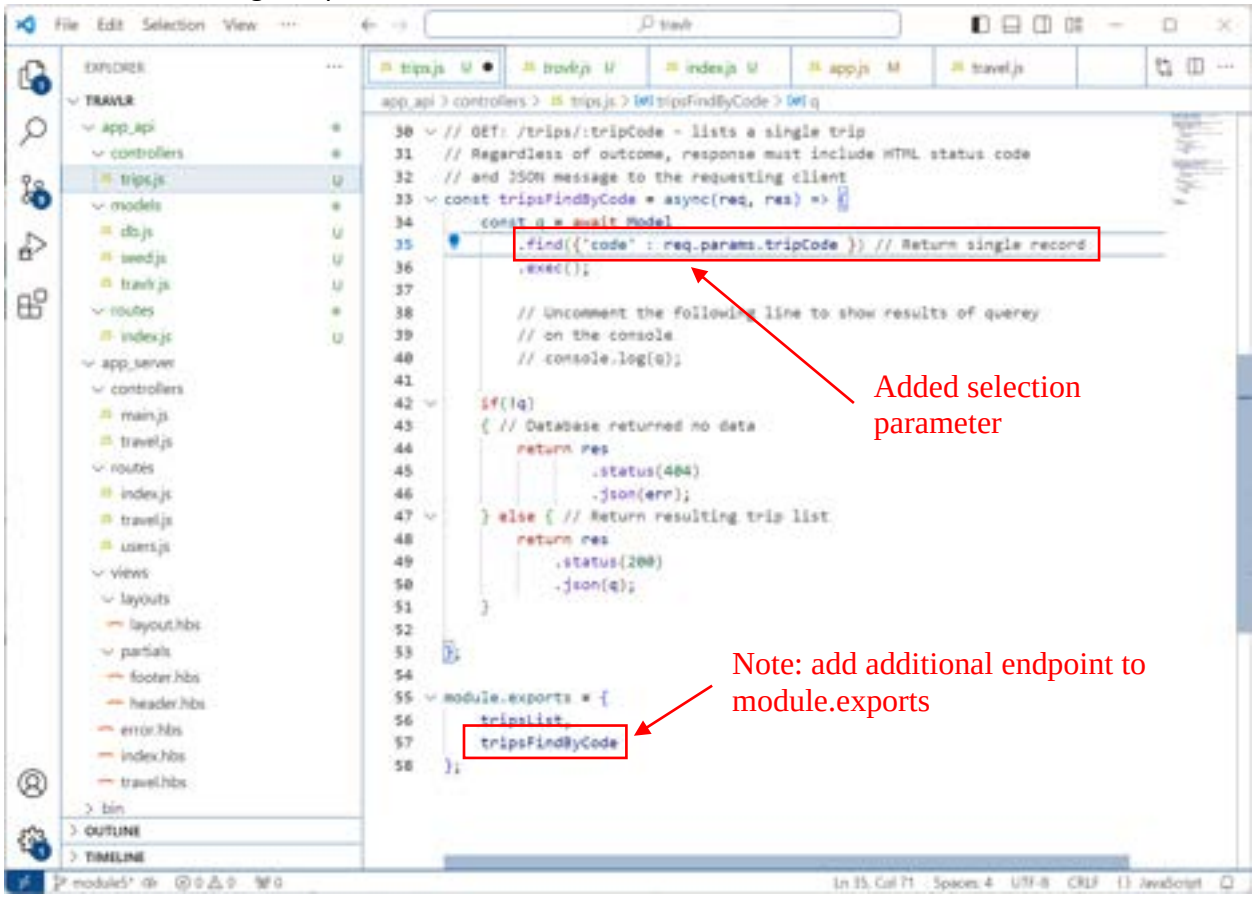
4. Now that we have the route wired-up in our app, it is a good idea to run the webserver and test. While this is something that can certainly be tested just by pointing your browser at http://localhost:3000/api/trips, this is a good opportunity to use postman to look at the API transaction. Open Postman, select continue with the lightweight API client. Enter your URL and select ***Send*** to continue.



You should be able to clearly see, and scroll through, the JSON output of the mongoose call – and this should match the data that you used to seed the database.

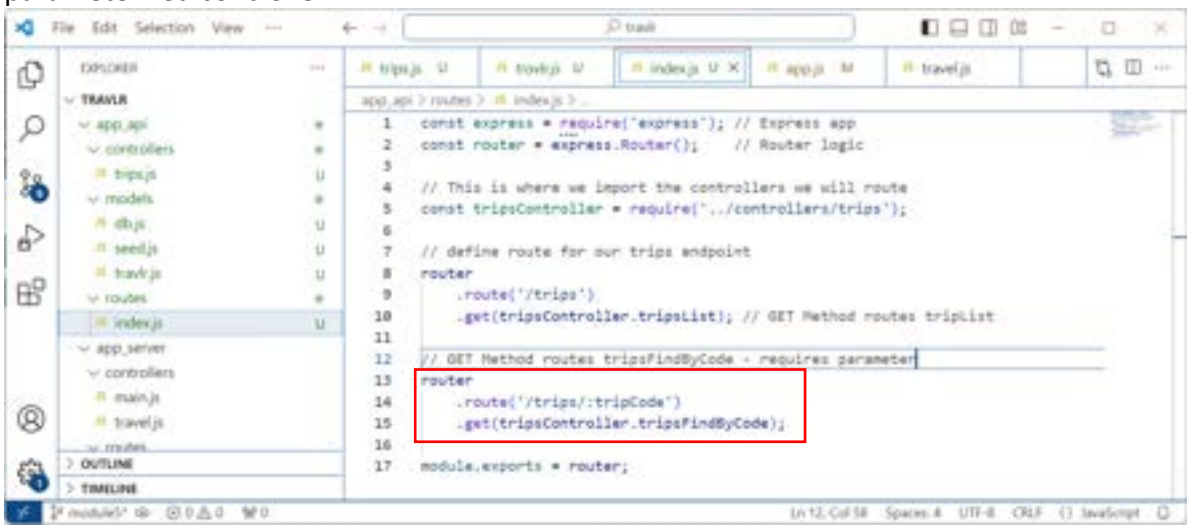
5. Now that we have shown that our controller and route is successfully accessing the database, it is time to add an additional route to our API. Here we will re-open the ***trips.js*** file in our ***app_api/controllers*** directory. We will duplicate and modify our ***tripsList*** endpoint to create an endpoint that will take a single parameter (the 'code' field) and use that as a

filter to select a single trip from the database.

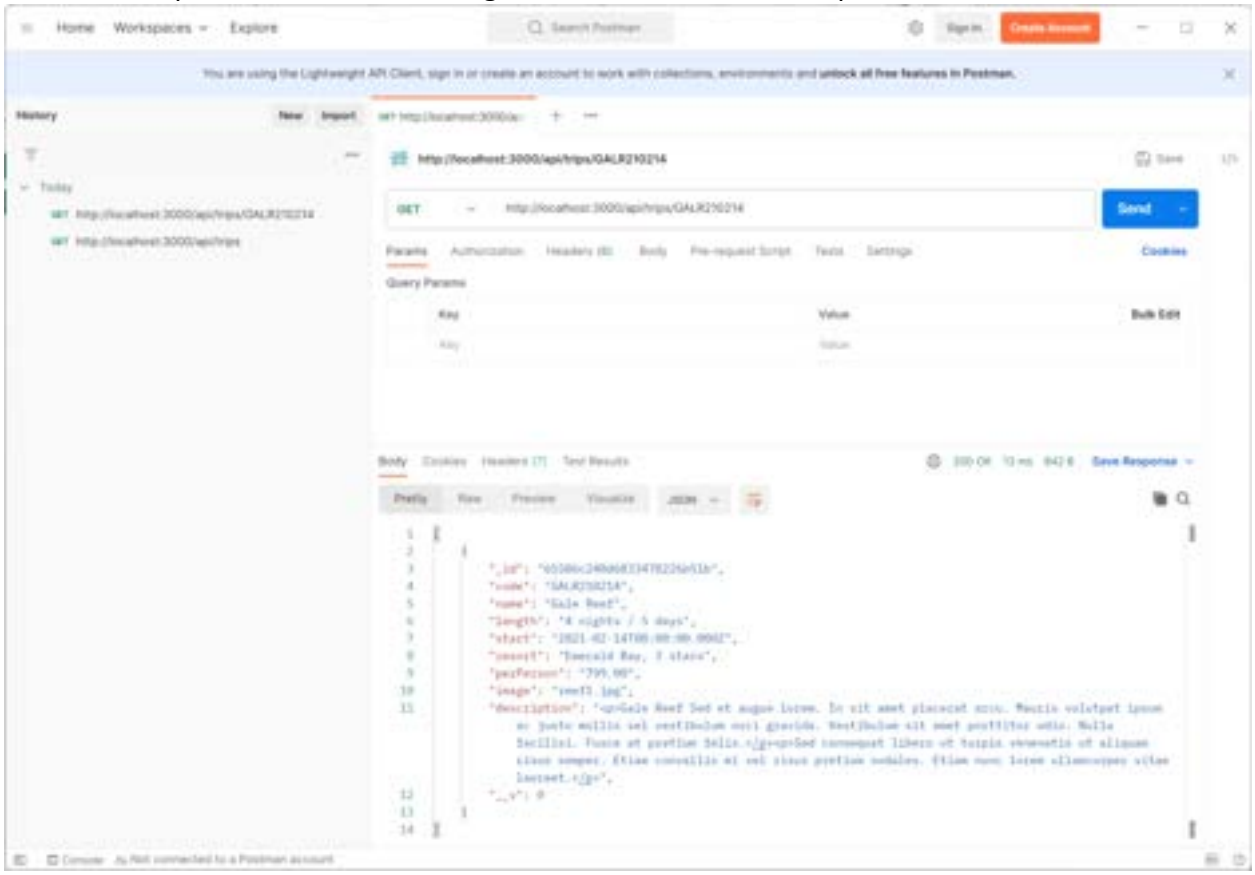


You will see that we are making a very small change to pull in the tripCode parameter from the API endpoint.

6. Before we can utilize the new controller, we need to make a small edit to the ***index.js*** file in the ***app_api/routes*** folder. This is where we will add a route to take advantage of the parameterized controller.

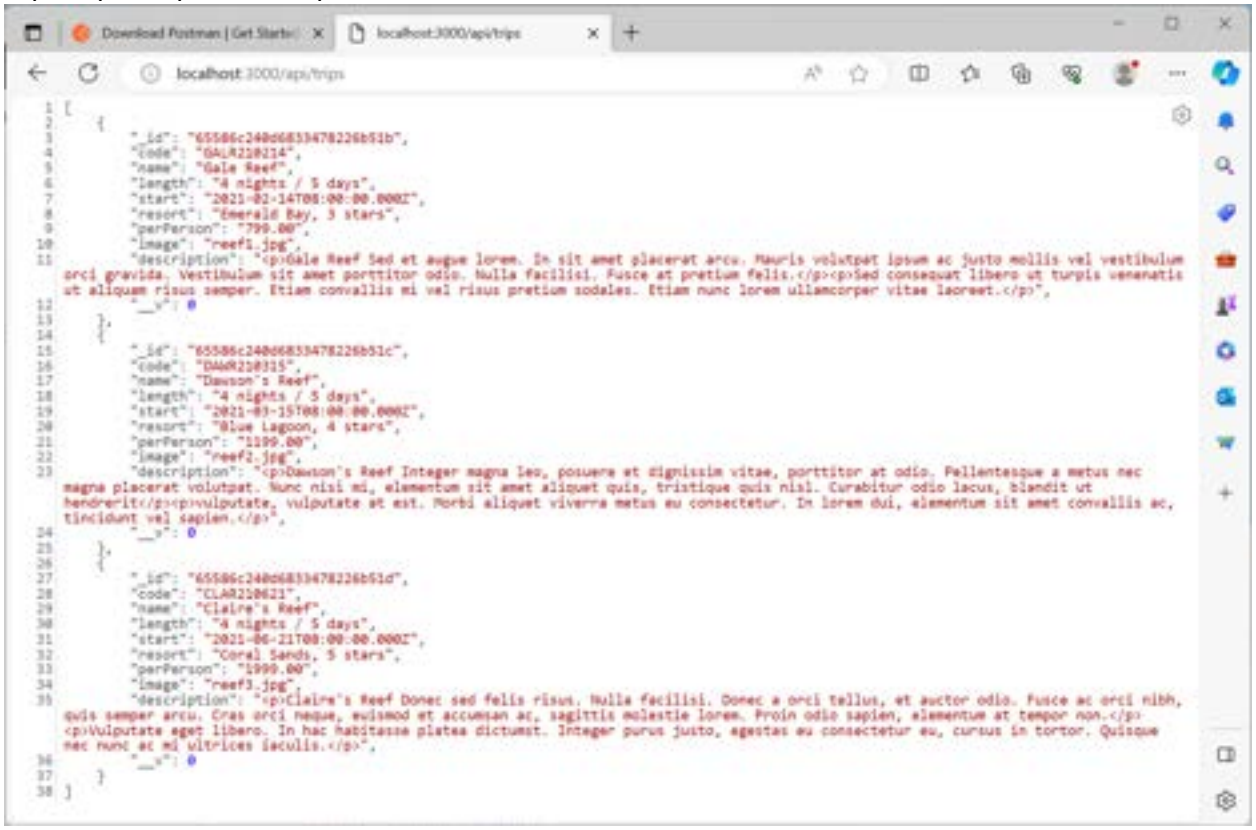


7. Now that we have made changes, we recycle the application and test to see that the new instance of the controller will return a single record. We will test this in Postman by adding one of the trip codes to the URL string for the call to our API endpoint:

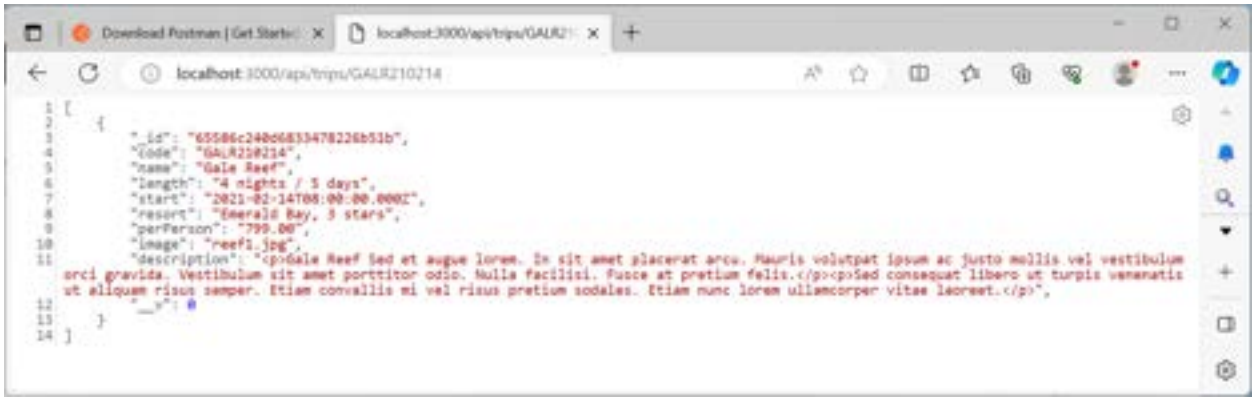


8. Another method that you can utilize to test your API (at least for GET methods) is to enter the URL directly into your browser. This is an easy way to do quick testing, but it is more difficult to manage this for larger numbers of tests – which is why we introduced Postman for API testing. Here is an example with the /api/trips endpoint, and then the

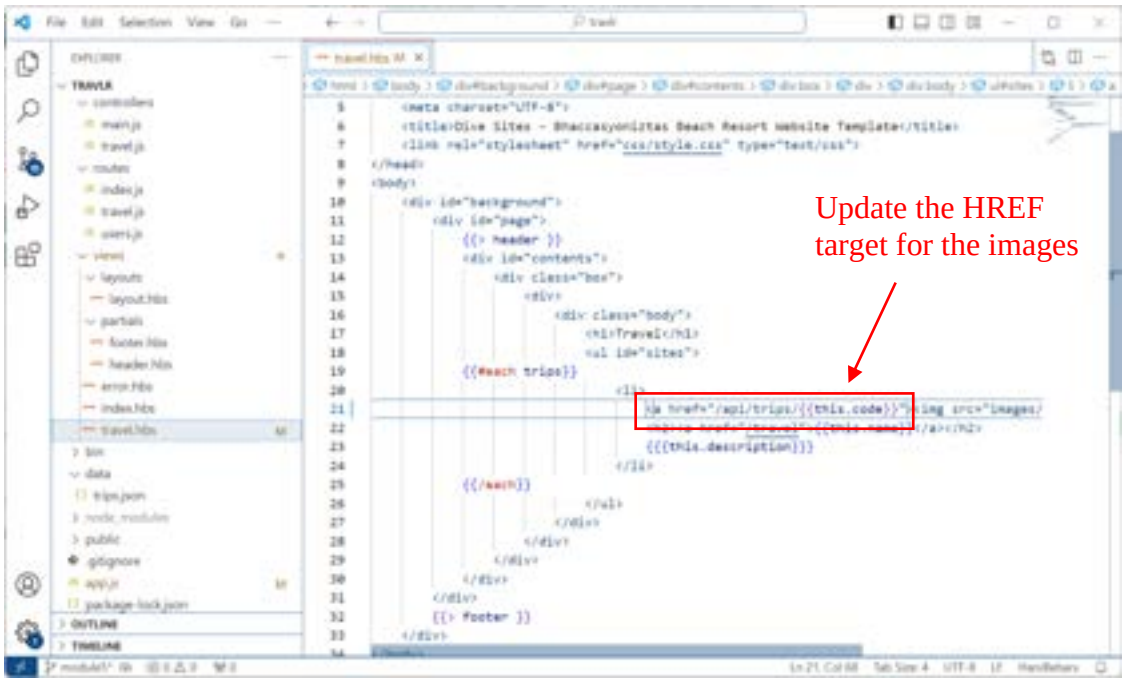
/api/trips/:tripCode endpoint.



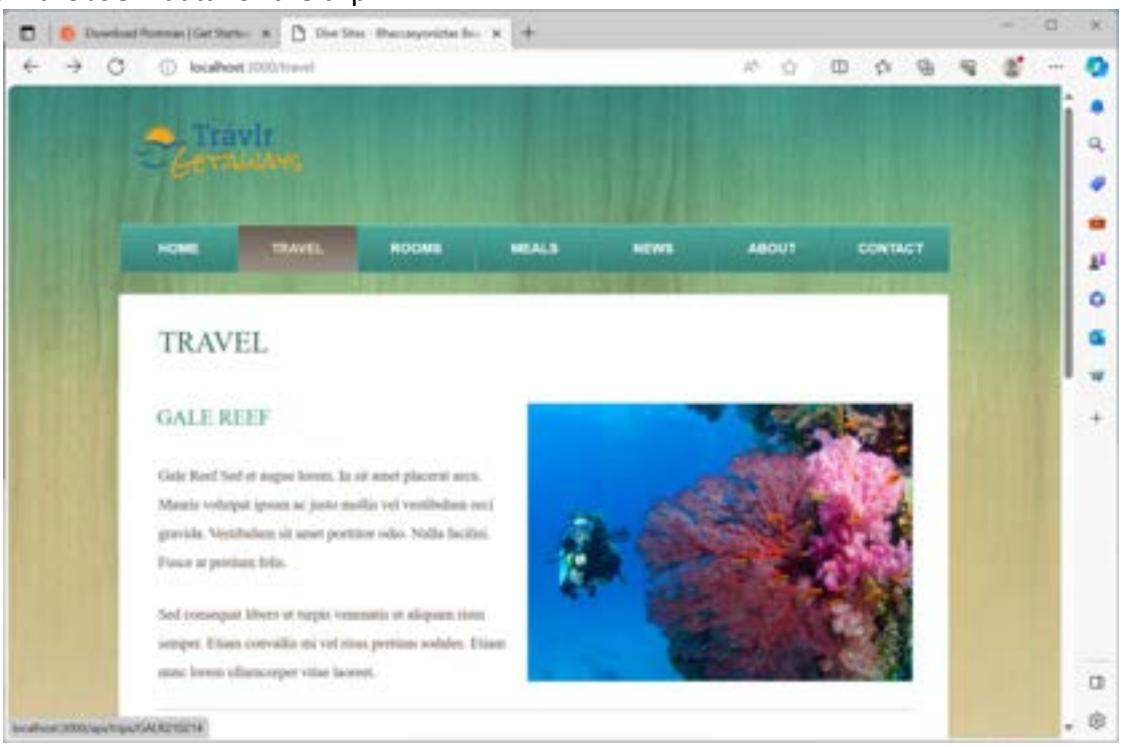
and with a filtered API call:



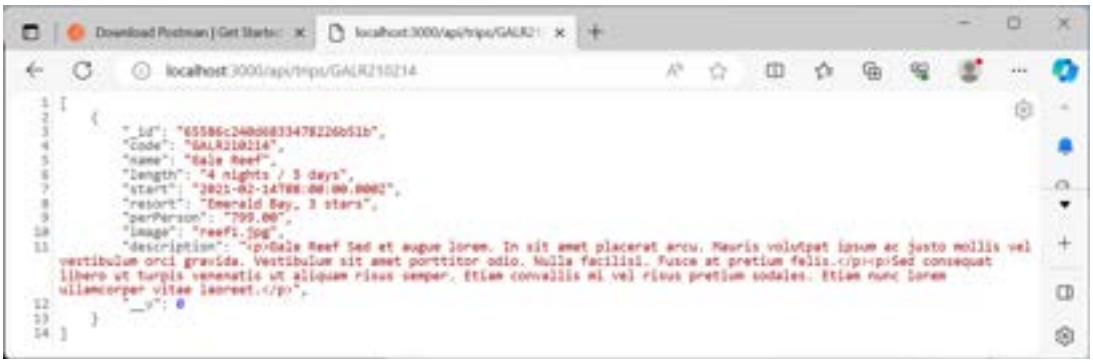
9. There is one more area that we should modify to help us with the utilization of our new API calls. We can adjust the ***travel.hbs*** template to change the HREF target to utilize the API, and further show that our filtered API call can be successful.



And testing in your browser you can see that selecting the image for each travel location will show the JSON data for the trip.



And clicking on the image brings up the JSON for the trip record.



While this is not an example of what you would normally do in a production system, it is a good example of what is possible with the API methodology. It can also be a good method to use to test your environment during the build process.

Modify Public Website to Use API Endpoints

In the previous section we did run across another item that needs to be refactored. The rendered view is still using static JSON from a file, so we need to update this to utilize the new API endpoints. We will be utilizing Chapter 7 of your textbook for guidelines to wire up the view to the new API endpoints to pull data from the database.

1. First, we need to determine how we are going to pull the data in from the APIs. Previously, we would have used the ***request*** Node module, but that has been deprecated and is no longer being maintained and its use in a production environment should be eliminated. In this case, we will be using the ***fetch*** api that is now built into Node JS. This is going to require that we modify the controller in ***app_server/controllers/travel.js***. There are a number of ways to accomplish this, but we are going to use a method that makes minimal changes to the original controller. The original controller is shown here (from our GitHub archive):

```
var fs = require('fs');
var trips = JSON.parse(fs.readFileSync('./data/trips.json','utf8'));

/* GET travel view */
const travel = (req, res) => {
    res.render('travel', { title: 'Travlr Getaways', trips});
};

module.exports = {
    travel
};
```

2. Since we originally generated the data from our JSON seed file, we need to add some additional information to attach to our API endpoint. This is going to include the URL, and the options that we are going to need to set on the connection (which HTML verb to use

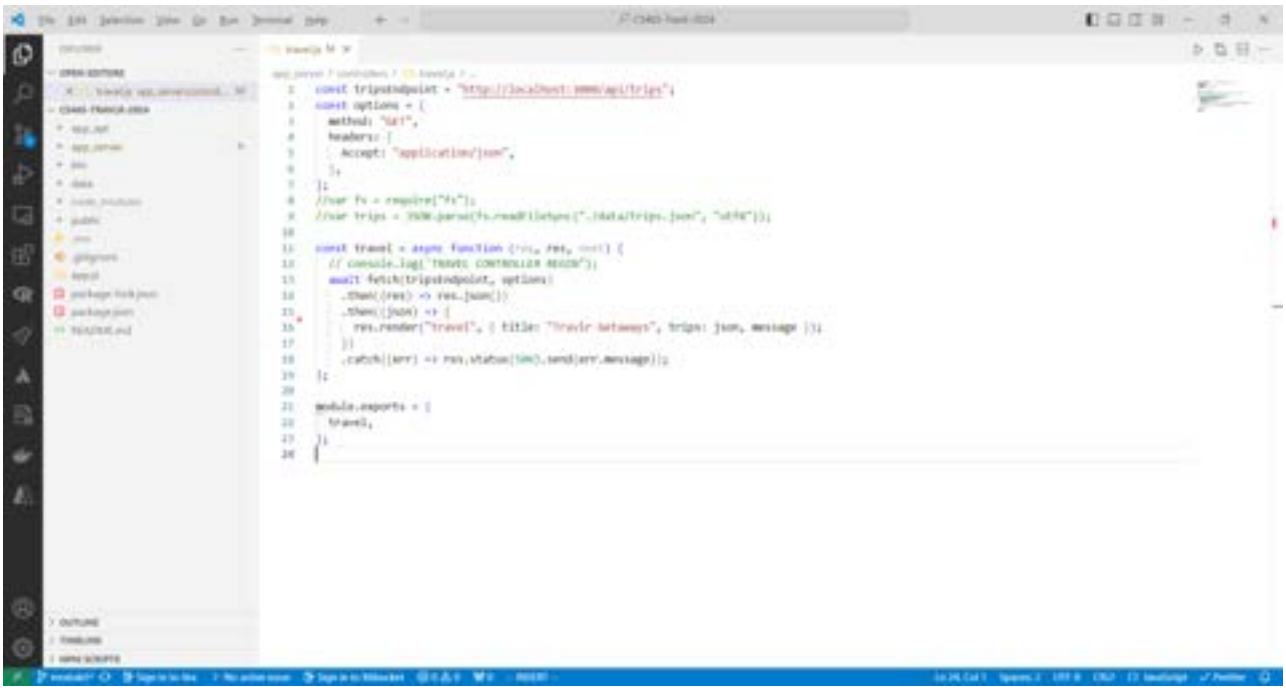
(GET), and any headers that needed identified. We will start by creating a variable for our API endpoint, and one for our list of options.

```
const tripsEndpoint = 'http://localhost:3000/api/trips';
const options = {
    method: 'GET',
    headers: {
        'Accept': 'application/json'
    }
}
```

3. Next, we will comment out the two lines that we used to read the data in from our seed file.

```
// var fs = require('fs');
// var trips = JSON.parse(fs.readFileSync('./data/trips.json','utf8'));
```

4. Now we need to replace our definition of the travel object with an asynchronous function. This is necessary as client interactions with an API endpoint are not synchronous by nature. By defining a function as async we create the ability to use the '***await***' keyword that synchronizes the communication and simplifies our interaction with the endpoint.



You will notice in the above screenshot that we have added three separate logging statements that we have commented out. These are to allow you to experiment with the function and the order in which things occur in the Node JS environment. If you remove the ***async*** and ***await*** tokens, you can run the application and it will not work as you might expect. All the data will be collected, but not necessarily in the order you would expect.

Utilizing the console.log capability is a good way to provide yourself with some debugging information when you are trying to determine what is happening with your code. This will all be displayed in the PowerShell window where you started your application.

The ***fetch*** method has three major components. First, is the main portion of the execution:

```
fetch(URL, Options)
```

This is the initial configuration of the routine – and you can see how we wire up both the URL and Options in the above section. The second part is the ***.then*** clause for the method. You can chain multiple clauses and they will be executed in order. We will use two ***.then*** clauses here:

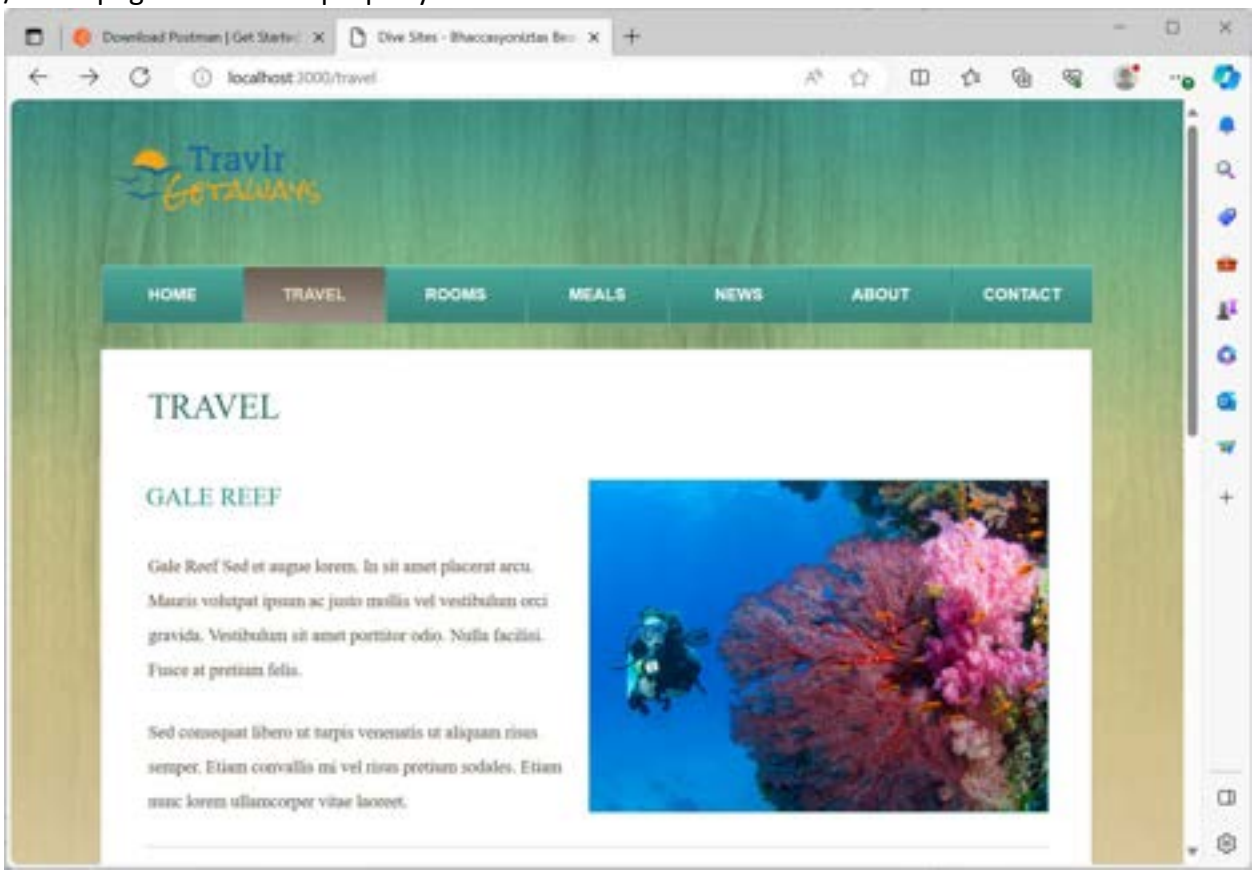
```
.then(res => res.json())
.then(json => {
      // console.log(json);
      res.render('travel', {title: 'Travlr Getaways', trips: json});
})
```

The first of these clauses takes the result from the fetch command and provides the output as JSON. The second of these clauses takes the json object from the previous clause and passes it to the render method. **Please Note:** We changed the 'trips' token at the end of the render line (it used to be the output of reading the datafile) and replaced it as a key-value pair where ***trips:*** is the key and ***json*** is the collection of values retrieved from the API.

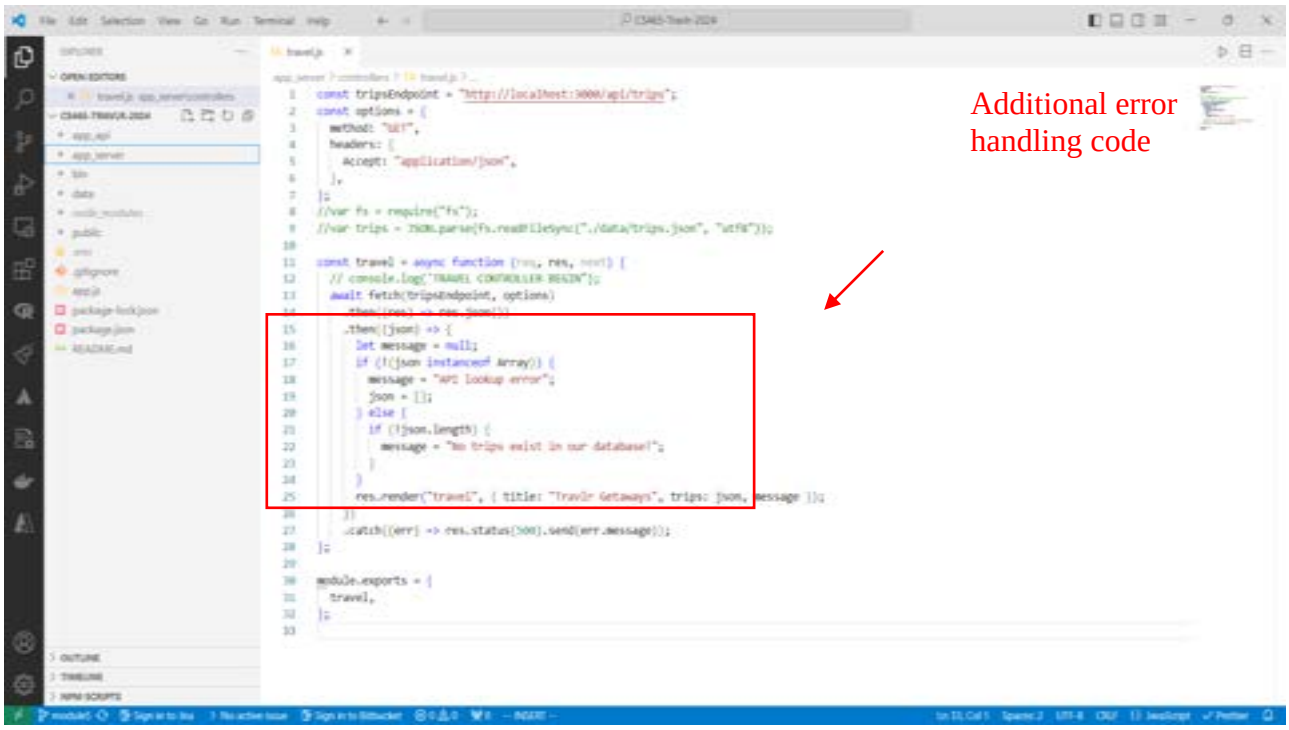
The third part of the fetch command is the ***.catch*** clause that we can use to tag any errors in the communication with the API. We add that here:

```
.catch(err => res.status(500).send(err.message));
```

5. Restarting your webserver at this point will allow you to test, and you should see that the /travel page is rendered properly.



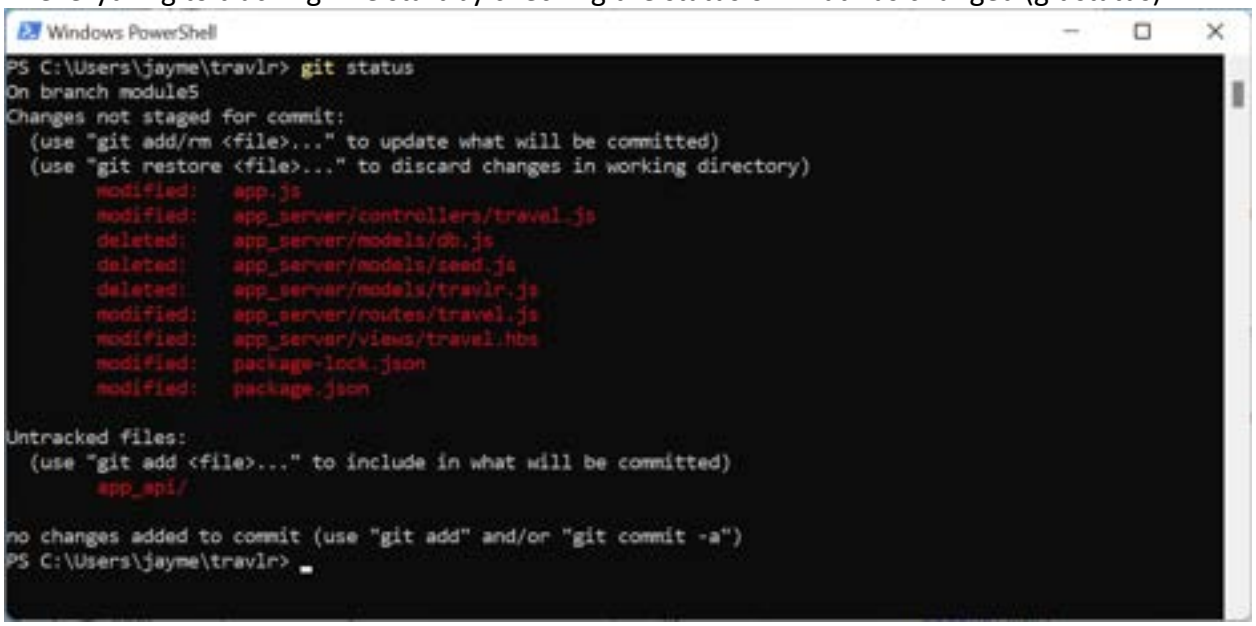
6. There is one more thing that we need to adjust in our controller, because there are some additional error conditions that may occur pulling data from the database through our API. The first, is a condition where the response does not contain any data – specifically it isn't an Array of JSON objects. The second is a condition where the response is an Array of JSON objects, but the array is of length 0. In this case, there is no data in our database to support the call. We want to add some tests and appropriate messaging in these cases.



7. Optional: You can repeat the process shown above to create methods specific to travelDetails.

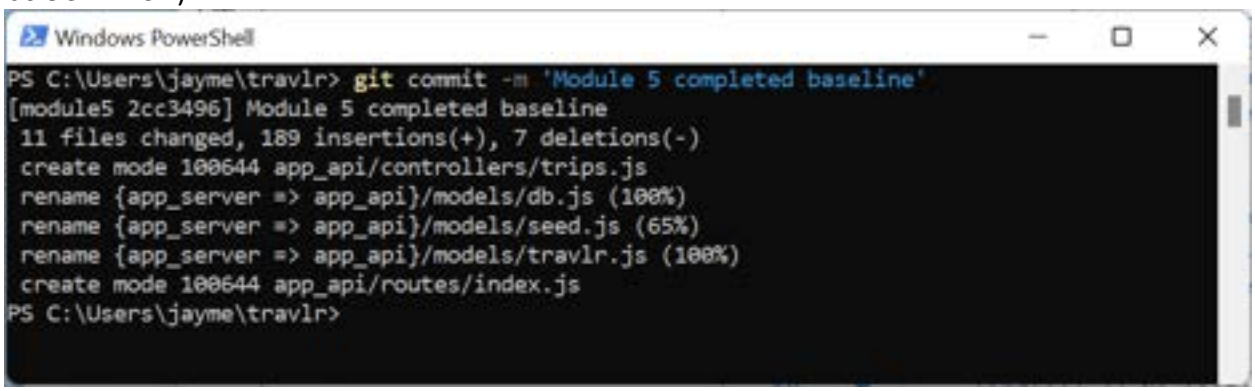
Finalizing Module 5

1. Now that we have completed Module 5, we go back to git and make sure that we add everything to tracking. We start by checking the status of what has changed (git status):

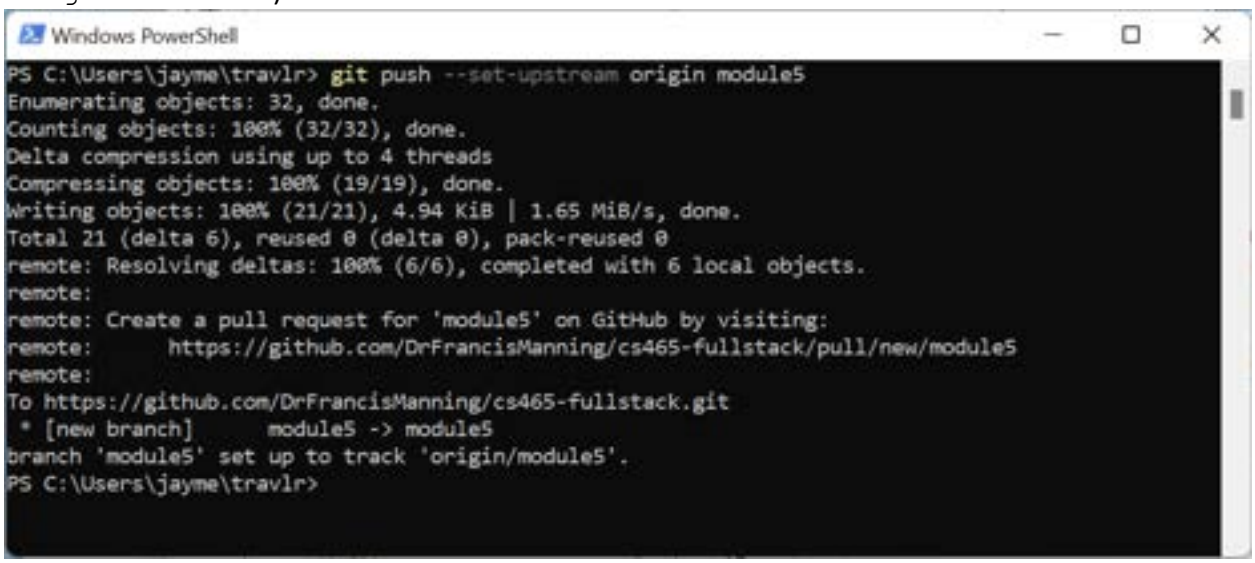


2. Then we add all of those changes into tracking (`git add .`):

3. Now we commit the changes (`git commit -m 'Module 5 completed baseline'`):



4. We push the changes back to GitHub for safekeeping (`git push --set-upstream origin module5`):



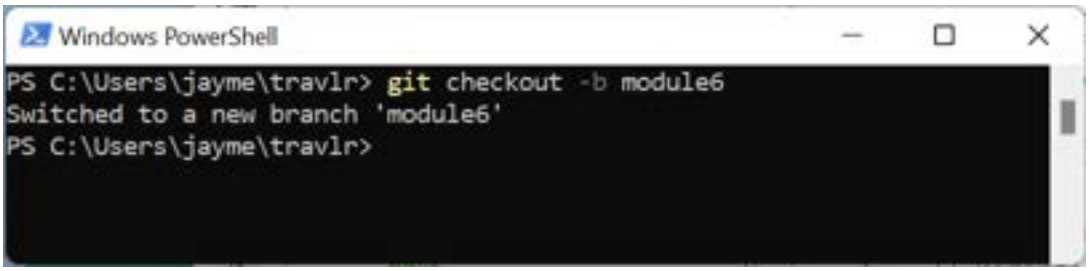
Module 6: SPA (Single Page Application)

In this module, we are going to take the next step forward and begin creating a Single-Page Application (SPA) that utilizes the API calls to efficiently fetch and manipulate data. This will use Angular (https://angular.io) – and industry standard for front-end web-application development as the bases for the coding framework for developing our SPA.

Create Git Branch for Module 6

Before you begin, it is important to make sure that you have created your new branch in git for Module 6. To accomplish this, we will perform the following command in a PowerShell window in the ***travlr*** project directory:

```
git checkout -b module6
```



Creating the Angular Admin Site

This next phase of development is going to change the model for how the application is displayed to the user. Up to this point all of the application logic has been handled by the server, with webpages rendered server-side and then presented to the client's browser for display to the user. In this phase of development, we are going to create what is called a Single Page Application or SPA. This is a common form of web-app development where all of that application rendering work happens on the client-side and effectively minimizes client-server traffic.

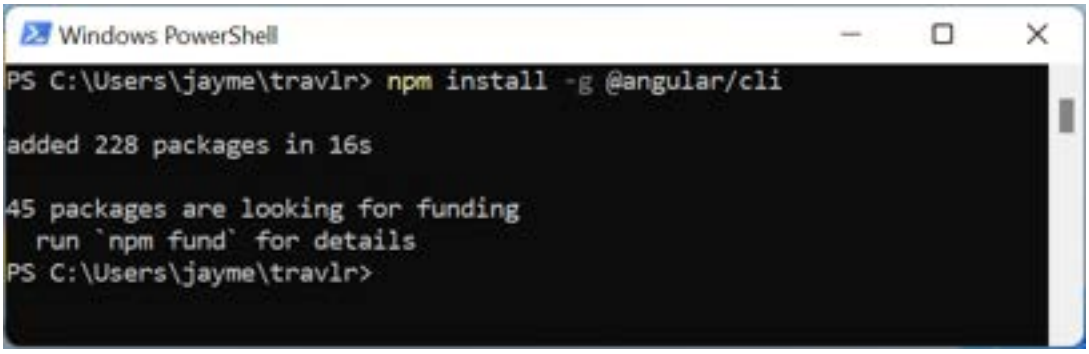
The first thing that needs to be accomplished in order to begin this development process is to install Angular and create our Admin site.

1. Install the Angular Command Line Interface (CLI). The current version of this tool is v17, and that is what this guide is based on. You should be able to continue to use this guide with newer versions of Angular by carefully reading the release notes to see if any of the concepts or methodologies used in this guide have been impacted by changes to the Angular API (this is something a developer would be responsible for when building and maintaining an application in a professional environment).

   Please Note: the @ sign in the command to install the Angular command line interface. This is different from what we have used in the past when installing a package in Node JS. This designates the Angular namespace which is a packaging/protection mechanism in Node JS.

When installing a package beneath @angular, you should have confidence that the package was developed and published by the Angular development team.

```
npm install -g @angular/cli
```



2. Now that we have angular installed in our Node.JS environment, we are going to create a new Angular application. To do this, we are going to run a command to create a new tree within our existing development tree.

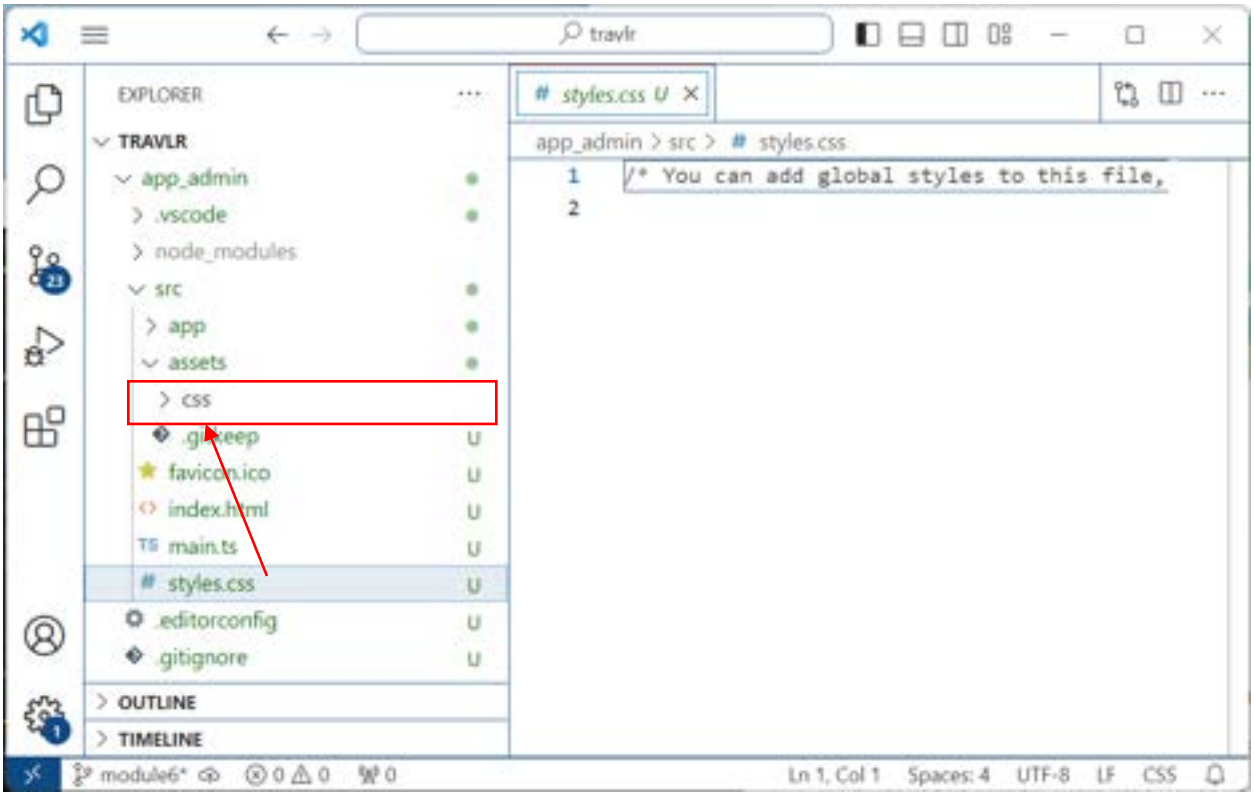
```
ng new travlr-admin --defaults=true --directory app_admin
```

You will be asked if you want to share usage data for what you are building with Google for the purpose of analytics. It is safe to answer this question with an ‘n’ for no.

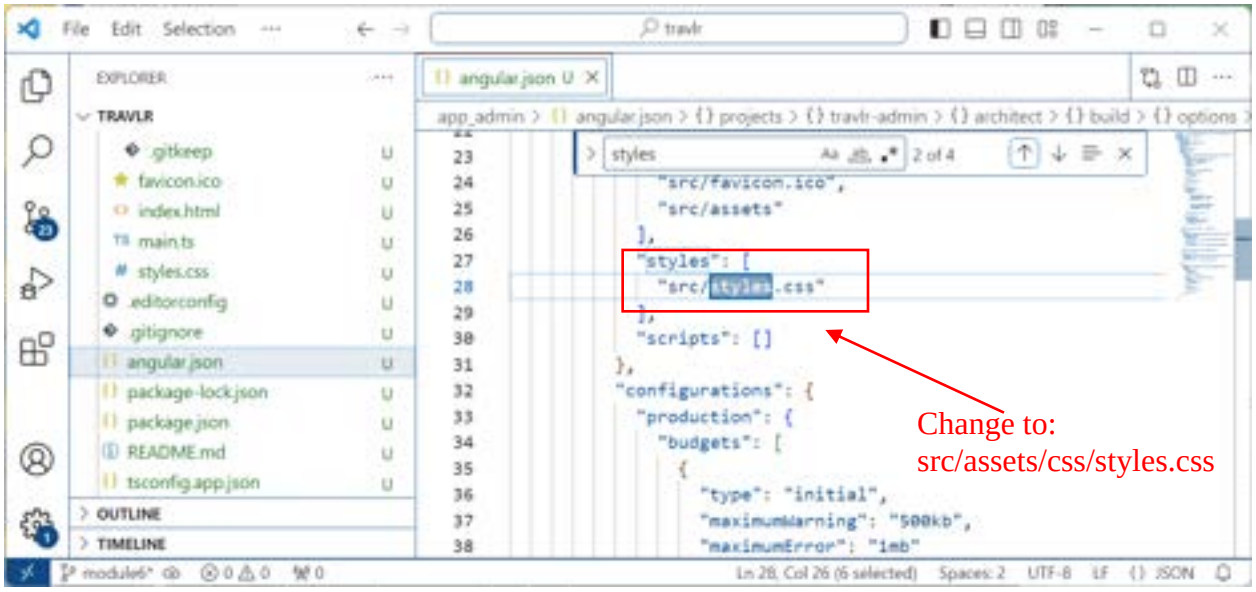
```
Windows PowerShell
PS C:\Users\jayme\travlr> ng new travlr-admin --defaults=true --directory app_admin
? Would you like to share pseudonymous usage data about this project with the Angular Team
at Google under Google's Privacy Policy at https://policies.google.com/privacy. For more
details and how to change this setting, see https://angular.io/analytics. No
Global setting: disabled
Local setting: No local workspace configuration file.
Effective status: disabled
CREATE app_admin/angular.json (2622 bytes)
CREATE app_admin/package.json (1043 bytes)
CREATE app_admin/README.md (1065 bytes)
CREATE app_admin/tsconfig.json (877 bytes)
CREATE app_admin/.editorconfig (274 bytes)
CREATE app_admin/.gitignore (548 bytes)
CREATE app_admin/tsconfig.app.json (263 bytes)
CREATE app_admin/tsconfig.spec.json (273 bytes)
CREATE app_admin/.vscode/extensions.json (130 bytes)
CREATE app_admin/.vscode/launch.json (470 bytes)
CREATE app_admin/.vscode/tasks.json (938 bytes)
CREATE app_admin/src/main.ts (250 bytes)
CREATE app_admin/src/favicon.ico (15086 bytes)
CREATE app_admin/src/index.html (297 bytes)
CREATE app_admin/src/styles.css (80 bytes)
CREATE app_admin/src/app/app.component.html (20884 bytes)
CREATE app_admin/src/app/app.component.spec.ts (934 bytes)
CREATE app_admin/src/app/app.component.ts (370 bytes)
CREATE app_admin/src/app/app.component.css (0 bytes)
CREATE app_admin/src/app/app.config.ts (227 bytes)
CREATE app_admin/src/app/app.routes.ts (77 bytes)
CREATE app_admin/src/assets/.gitkeep (0 bytes)
√ Packages installed successfully.
    Directory is already under version control. Skipping initialization of git.
PS C:\Users\jayme\travlr>
```

3. Now that the ***Angular*** app has been created in our ***travlr*** project tree, there are two structural changes we need to make to conform to industry best practices for application organization. We will first open our project in VS Code and see the new ***app_admin*** folder. We will want to descend the ***app_admin*** tree and create a folder ***src/assets/css*** where we will store the stylesheets for our application.

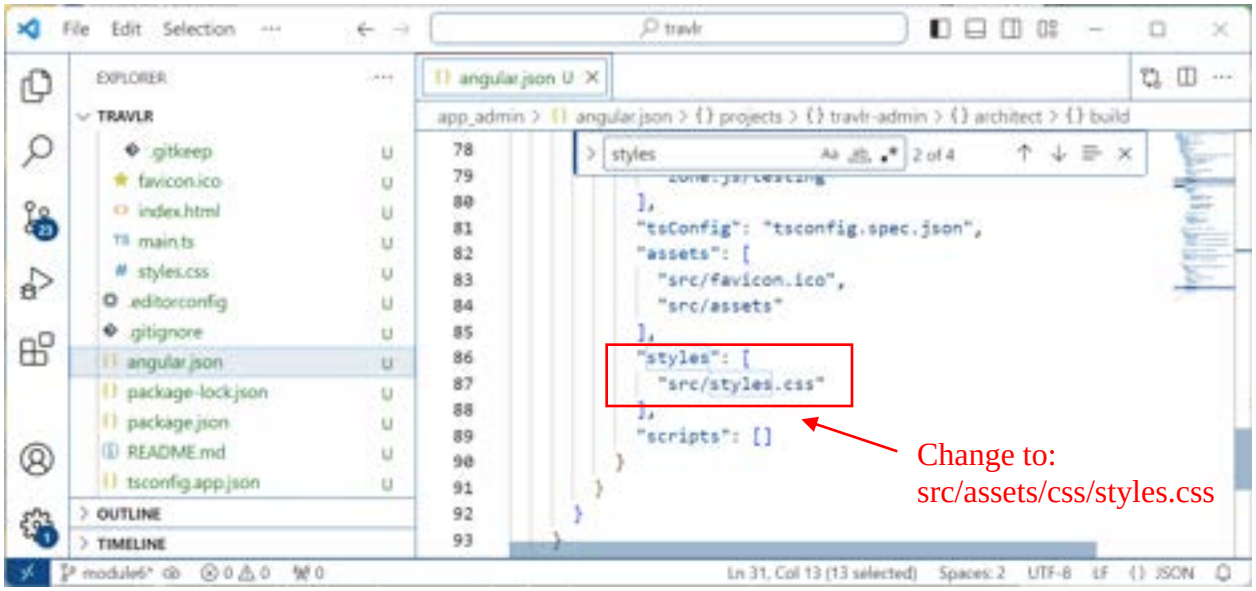
4. Once we have created the ***css*** folder, we will drag the **style.css** file from ***app_admin/src*** into the ***css*** folder. This will help to provide organizational consistency – but it will also require some additional editing of the default application.



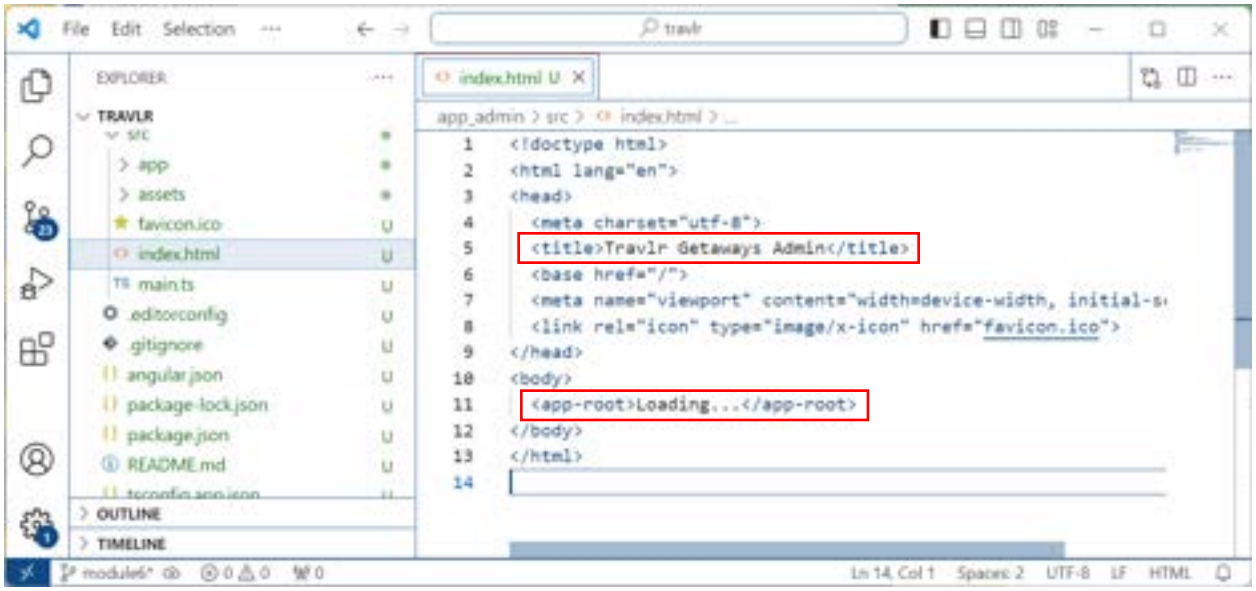
5. Now that we have relocated the stylesheet, we need to inform the Angular platform where to find it. To do this, we will be editing the file **app_admin/angular.json** and replacing the two references to **src/styles.css** with **src/assets/css/styles.css**.



and

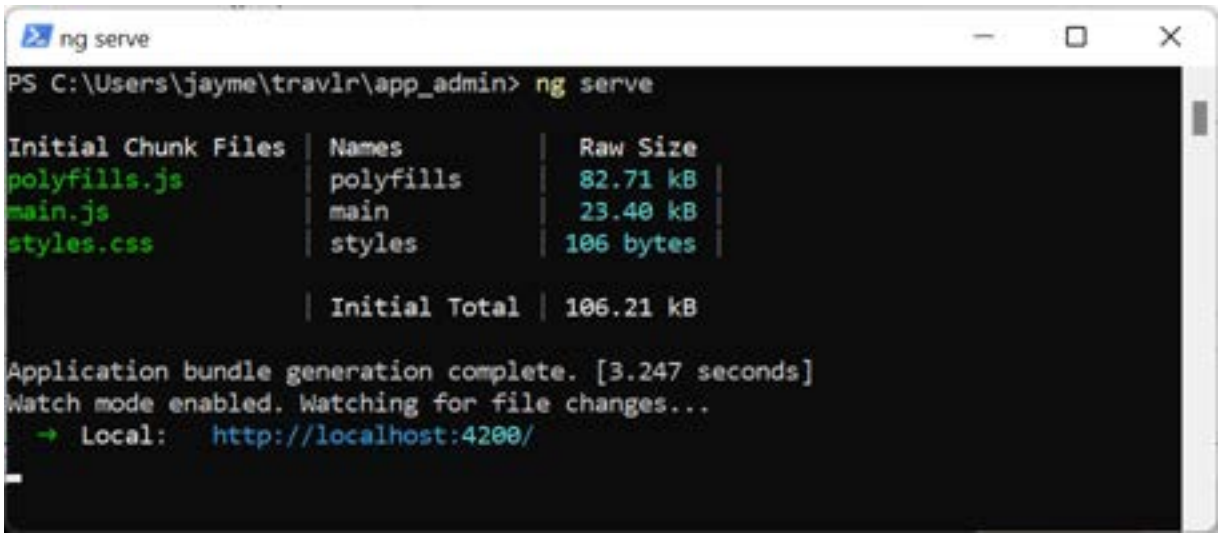


6. We are also going to want to replace the title and add a loading message. This will require an edit of the ***app_admin/src/index.html*** file.

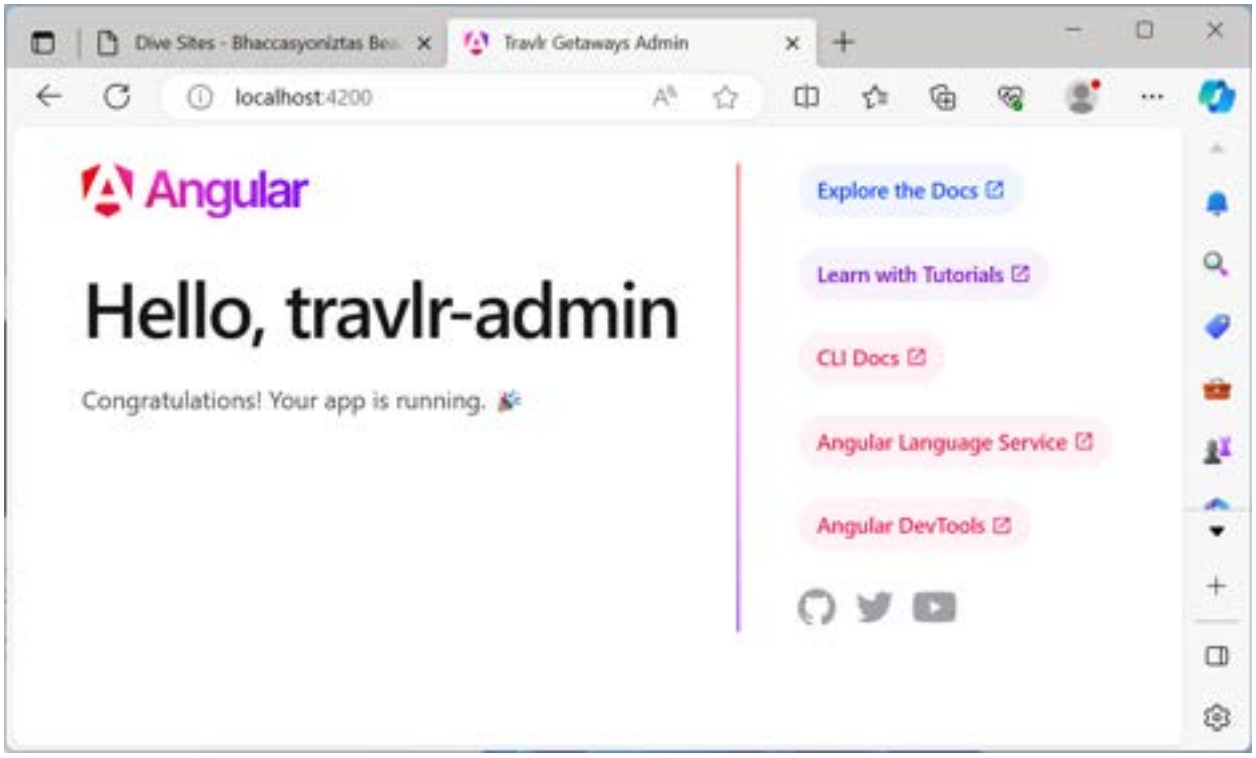


7. Now that we have completed the preliminary setup, you are going to want to open another PowerShell Window and start the administrative server. So change into the ***app_admin*** folder and start the server with:

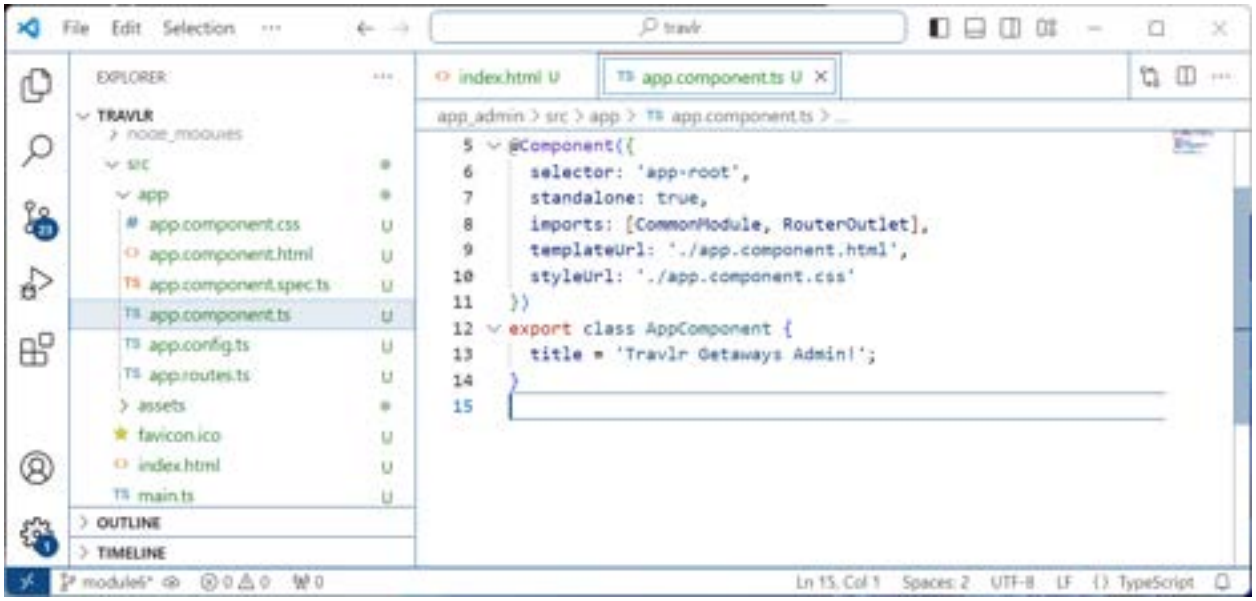
```
ng serve
```



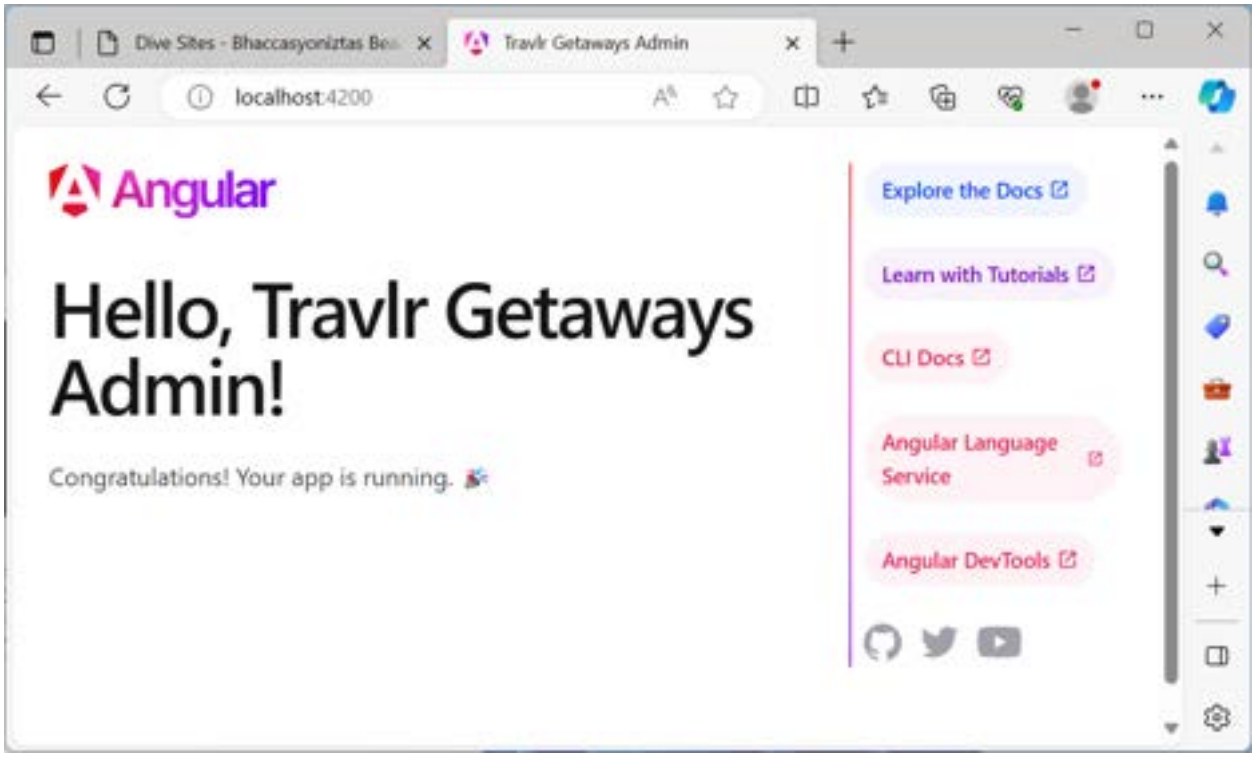
8. Now the application is running and we can attach to it. Open up a new tab in your browser and connect to the app at: http://localhost:4200 .



9. You will see the title 'Hello, travlr-admin' is rather clunky. We can adjust that in the /src/app/app.component.ts to change this string and save it. You should immediately see the browser refresh and the text should render as specified.



When we save this window, the client window re-renders as:



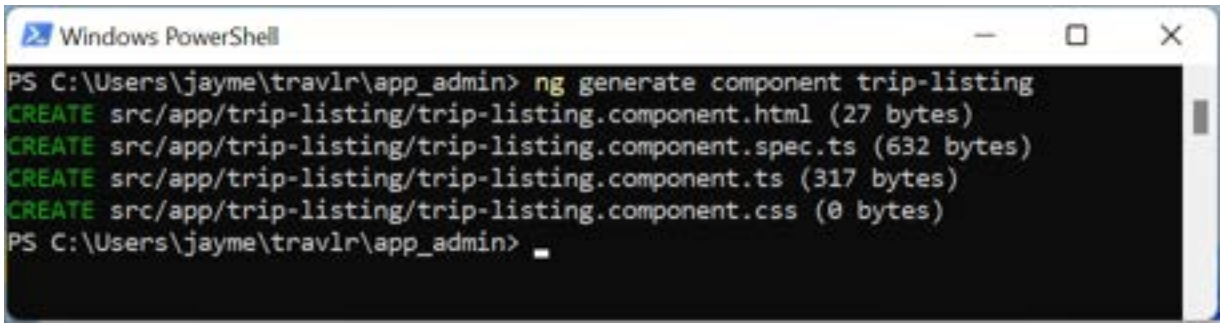
At this point, you have successfully created the basic Angular application that will support the administrative functionality for the Travlr Getaways application!

Create Trip Listing Component

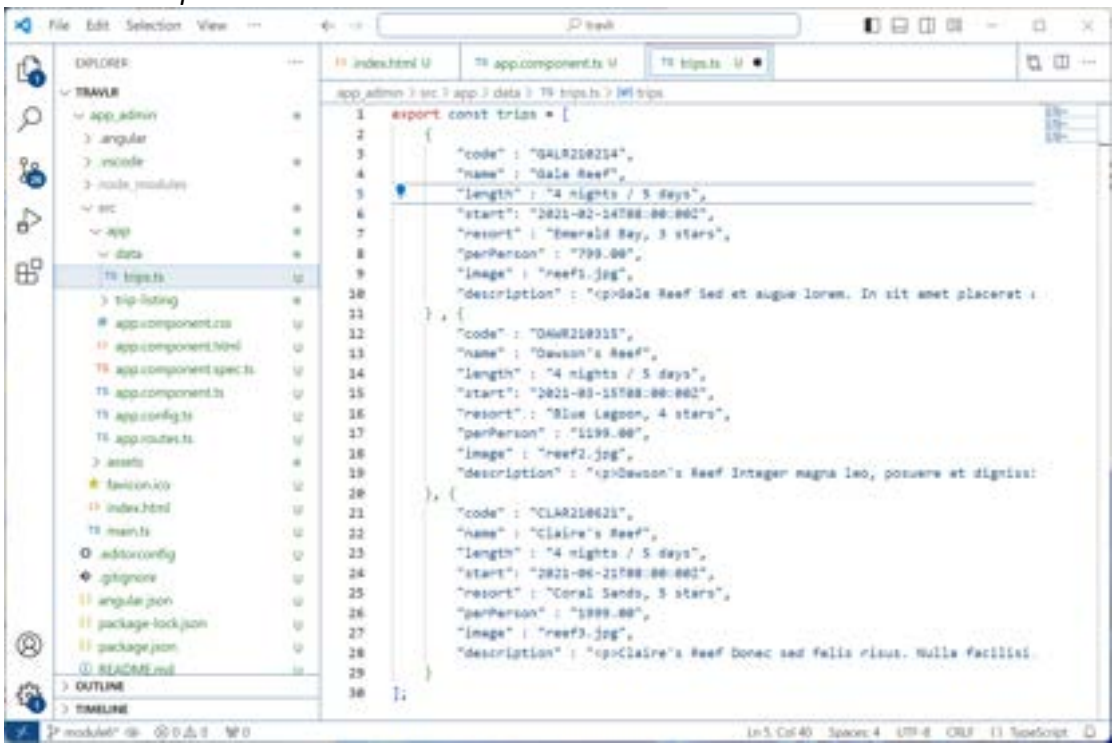
Now that we have the initial application constructed, we need to do something so that it will display useful data. To do this, we are going to need to create an Angular component. This component will be used to display the list of trips.

1. In your PowerShell window change to the ***app_admin*** folder in your application, and generate an Angular component called "trip-listing" to list the trips.

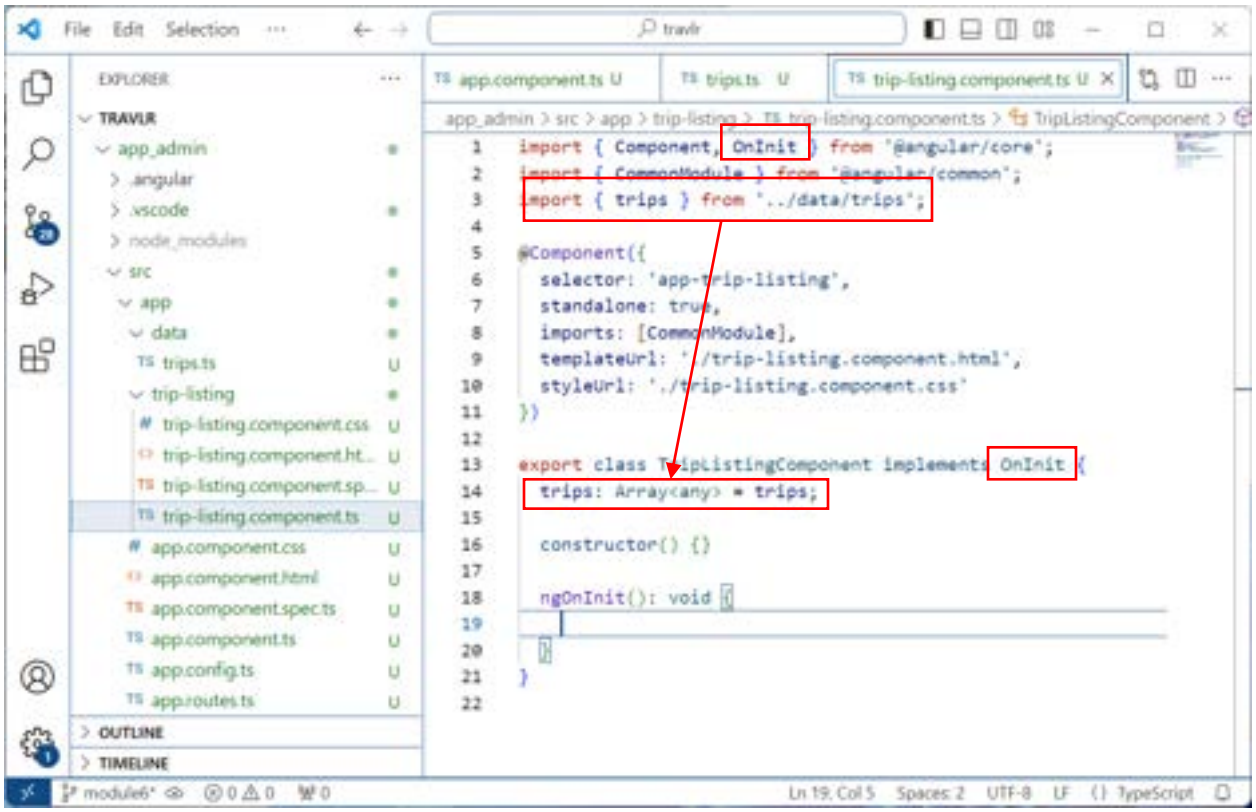
```
ng generate component trip-listing
```



2. Create a ***data*** folder beneath ***src/app*** and create a **trips.ts** file. This will contain TypeScript code that is based on the JSON data (collection of trip records) that we created earlier. Copy the contents of the ***trips.json*** file in the upper level ***/data*** folder and assign the data to a variable called *trips*.

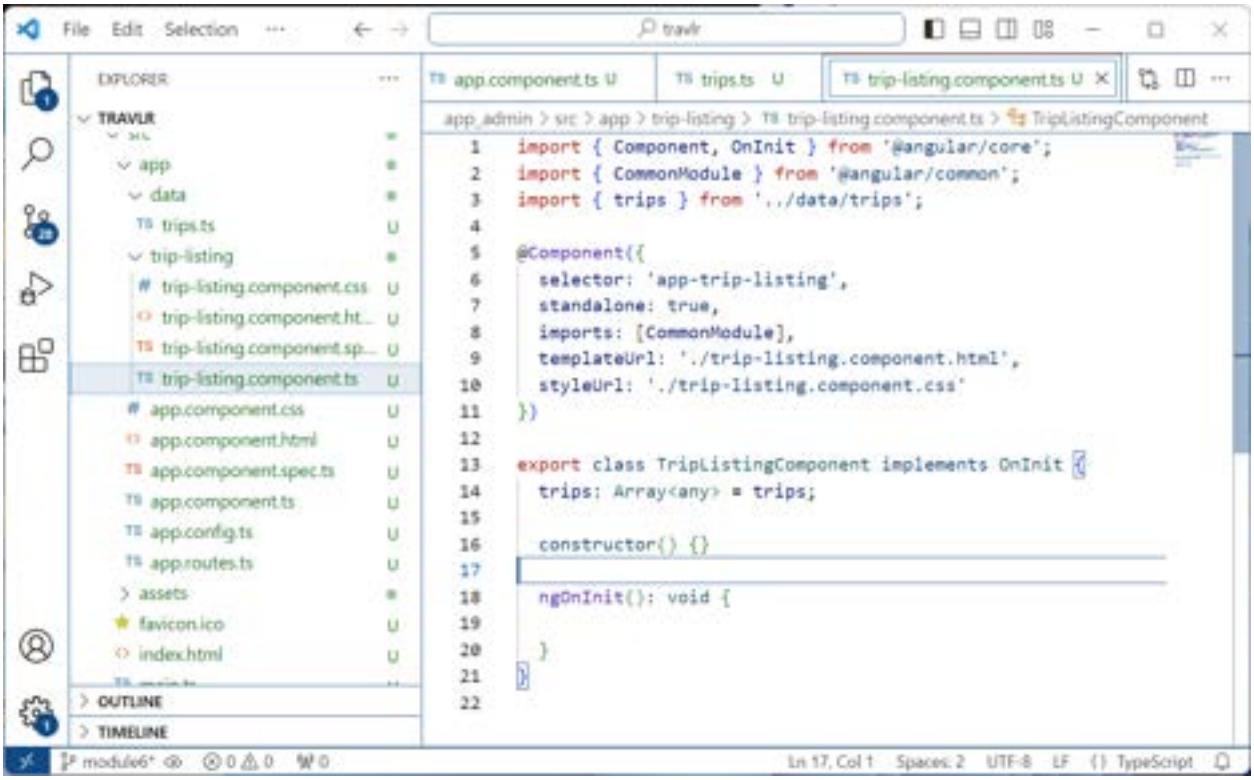


3. Edit the ***trip-listing.component.ts*** file to add an import of the new ***trips.ts*** file, and define an appropriate class variable within the *TripListingComponent* class to contain the data. We will also need to add *OnInit* to the imports for the class and setup the basic structure for our *TripListingComponent*.

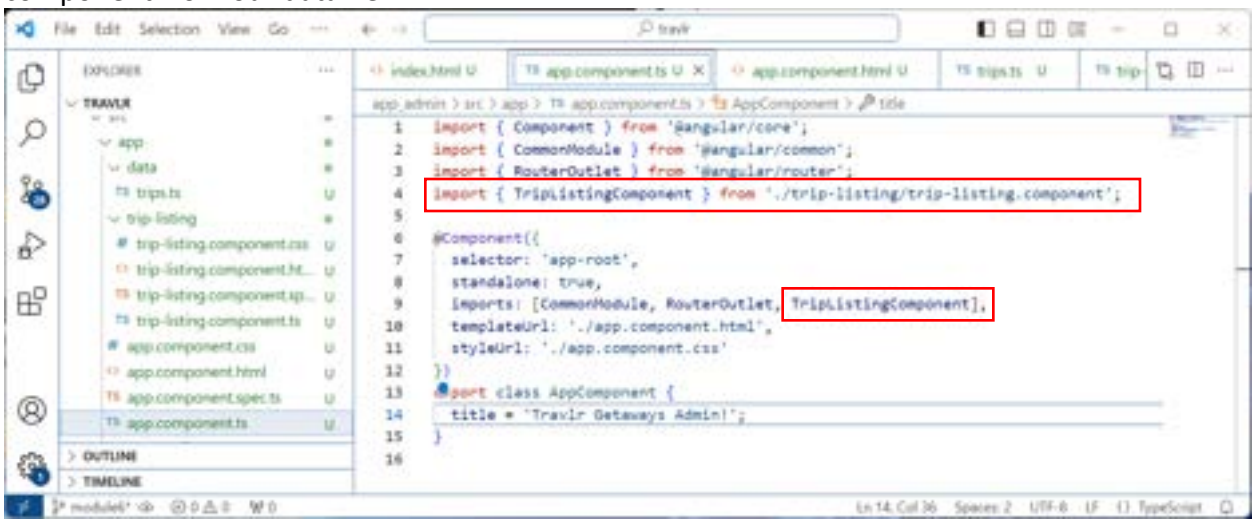


The next step will require editing the ***trip-listing.component.html*** file and replace the initial contents: trip-listing works! with <pre>{ { trips | json } }</pre>

By adding the trips array to the class file, the variable becomes accessible from within the HTML. Notice that Angular uses the double curly brace notation like Handlebars.

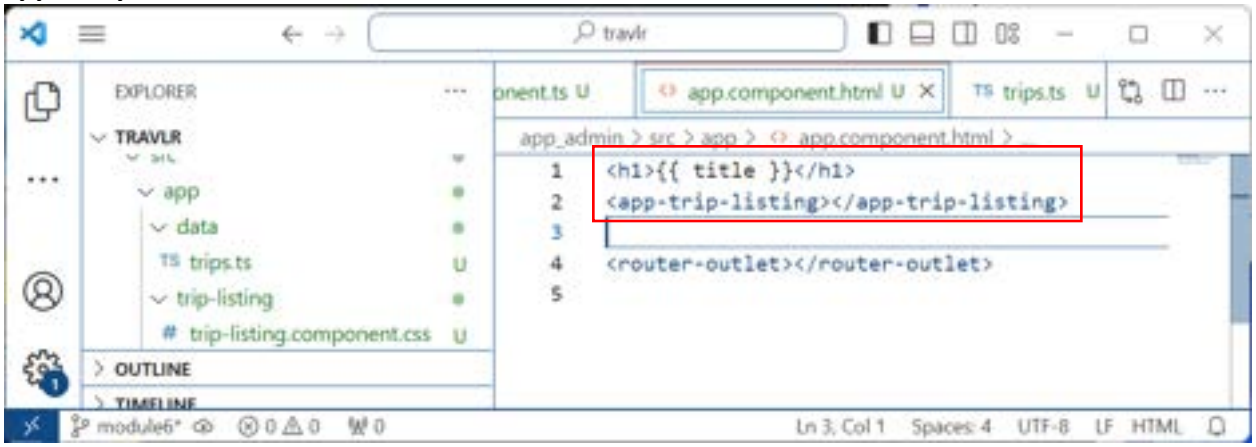


4. In the next step, we need to make one more edit to the ***app.component.ts*** file and pull in our *TripListingComponent*. This will allow us to access the data that we have pulled into that component from our datafile.

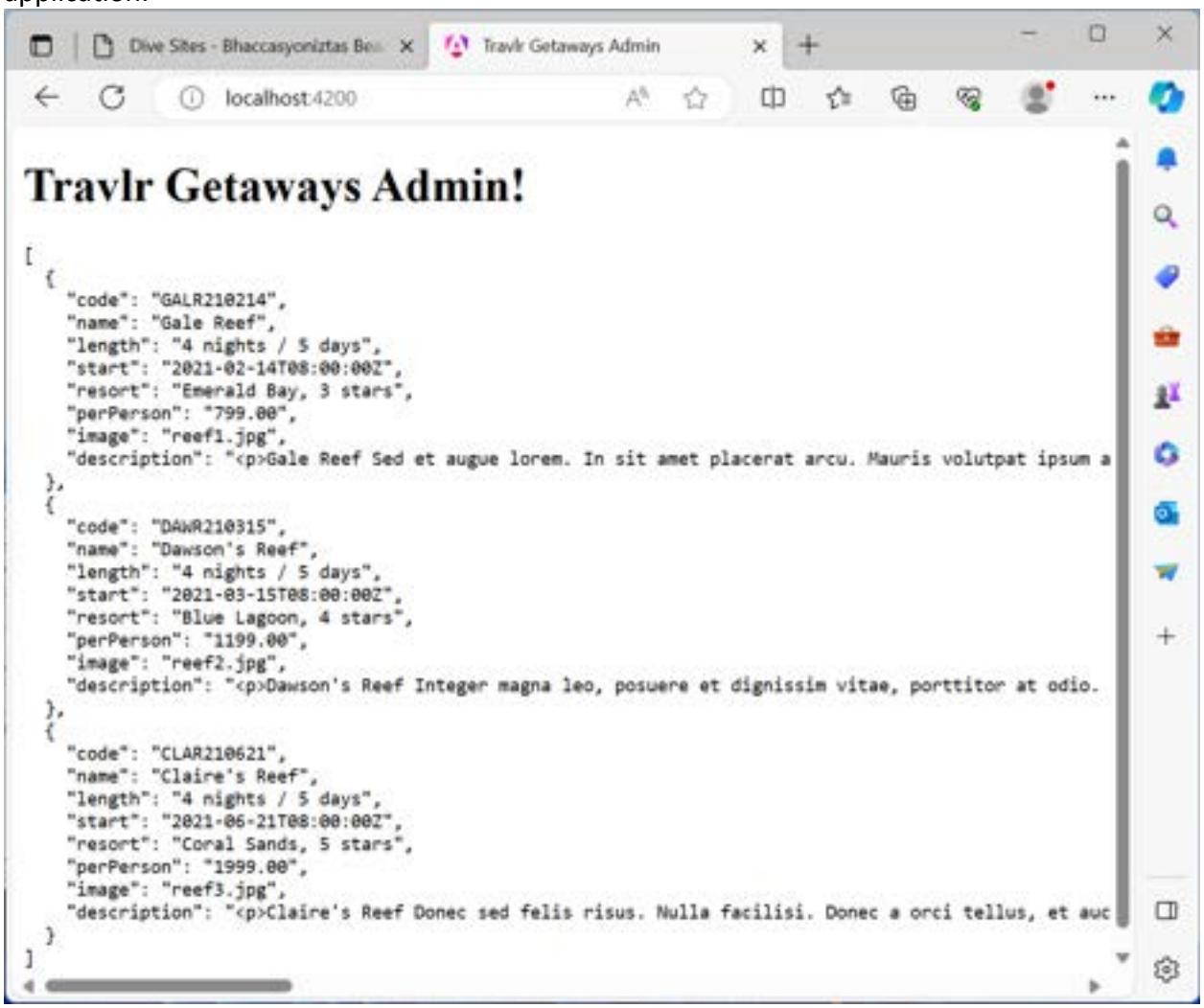


5. And with one more small edit, we can dynamically pull both the *title* variable from the *AppComponent* class and the *trips* array via the *app-trip-listing* selector from the *TripListingComponent* class and render them in our application. This edit will be made in the

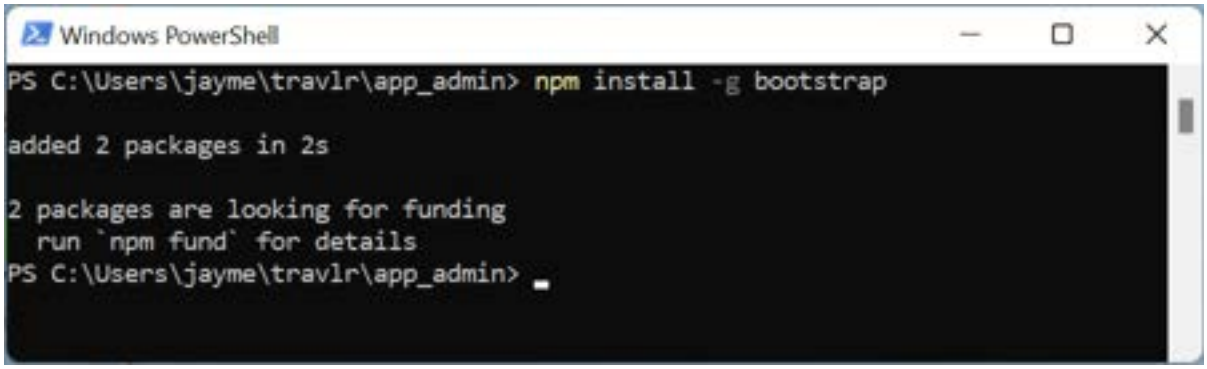
app.component.html file.



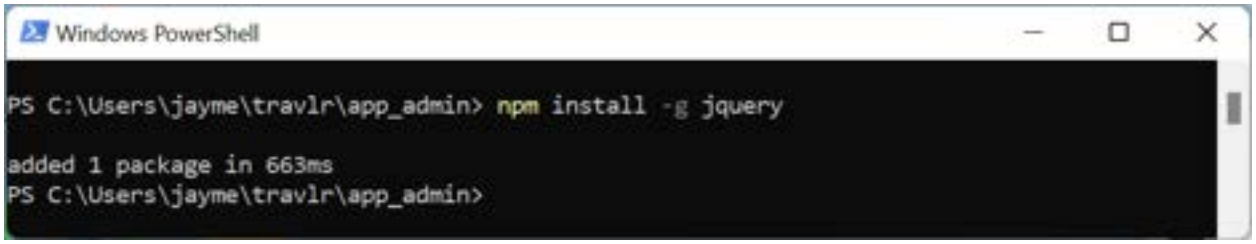
6. Once this file is saved, the application should automatically refresh and show the JSON in application.



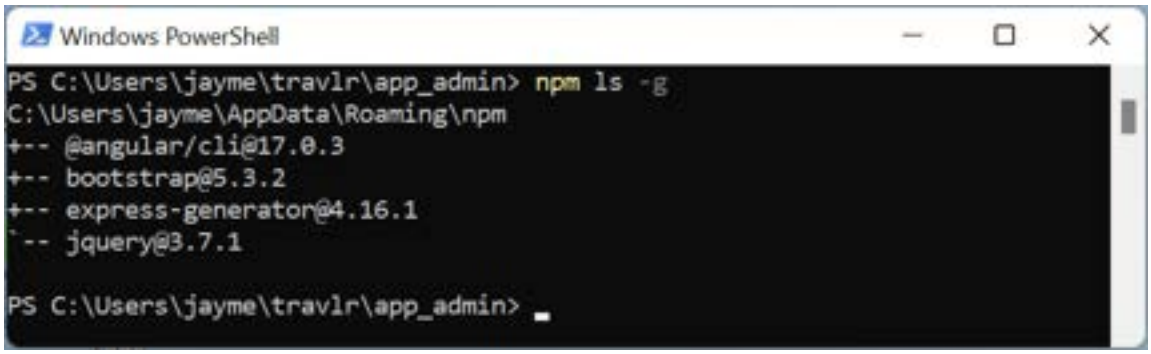
7. Now that we can see that we have access to the data, we need to determine how we can format the data so that it is usable for a front-end application. To handle this, we are going to involve the Bootstrap CSS framework. This requires an additional install in our Node.JS environment.



8. We are also going to need to install jQuery to enable a number of data manipulations later on. We are going to do this in our Node.JS environment. Once we install jQuery, we will use the 'node ls -g' command to show what we have installed in the global context in our Node.JS environment.



And then:



9. One of the things to bear in mind is that the Bootstrap CSS Framework is versioned. Consequently, we need to pay attention to the versions that we are using and including in our application. For the purposes of this Full-Stack guide, the following versions will be utilized:
    a. bootstrap – version 5.3.2
    b. popper – version 2.11.8
    c. jQuery – version 3.7.1

These necessary tags to import these packages into the code-base can be found in the following locations:

Bootstrap - https://getbootstrap.com/docs/5.3/getting-started/introduction/
Popper – on the above page with Bootstrap.
jQuery - https://releases.jquery.com

When you are working with these tools you may need to adjust version to take advantage of new features or to address material defects in existing code. Will we be adding code to the ***app_admin/src/index.html*** file.

We will be adding the bootstrap css package as a 'link' in the 'head' section, and the Javascript packages for popper, bootstrap, and jQuery packages as 'script' entries in the body. For this Full-Stack guide, the configurations will be:

Bootstrap (head section):

```
<link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bo
otstrap.min.css" rel="stylesheet" integrity="sha384-
T3c6CoIi6uLrA9TneNEoa7RxnatzjcDSCmG1MXxSR1GAsXEV/Dwwykc2MPK8M2
HN" crossorigin="anonymous">
```

Bootstrap (body section):

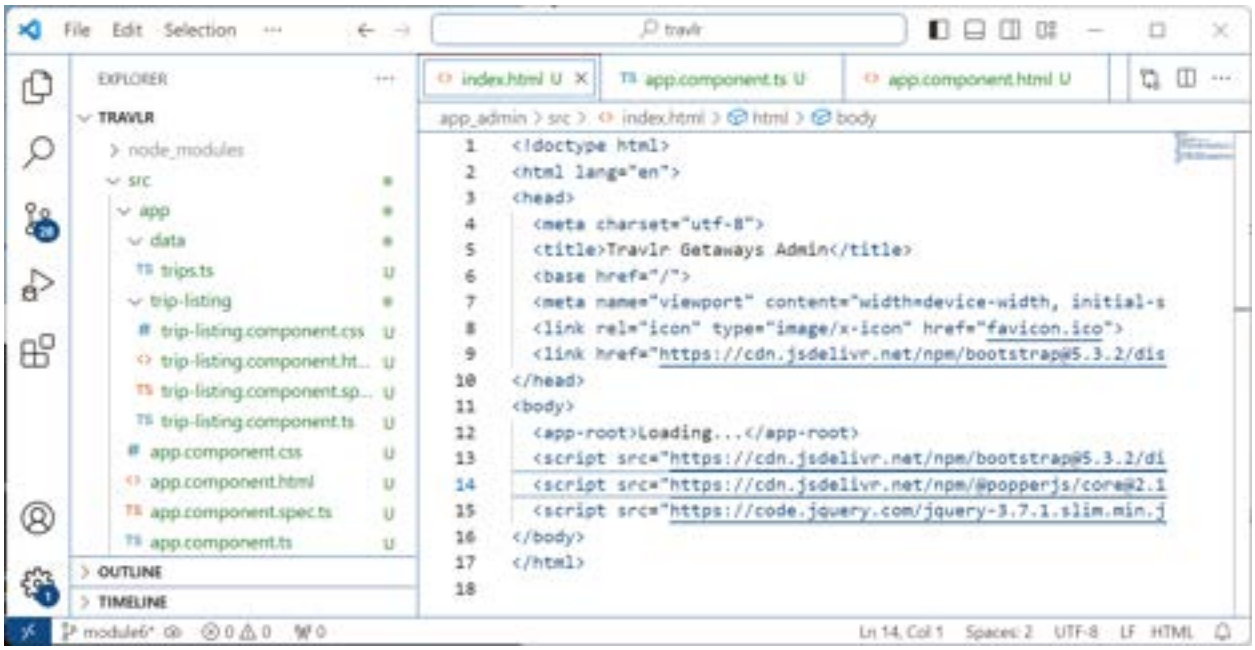
```
<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/boot
strap.bundle.min.js" integrity="sha384-
C6RzsynM9kWDrMNeT87bh95OGNyZPhcTNXj1NW7RuBCsyN/o0jlpcV8Qyq46cD
fL" crossorigin="anonymous"></script>
```

Popper (body section):

```
<script
src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11.8/dist/u
md/popper.min.js" integrity="sha384-
I7E8VVD/ismYTF4hNIPjVp/Zjvgyol6VFvRkX/vR+Vc4jQkC+hVqc2pM8ODewa
9r" crossorigin="anonymous"></script>
```

jQuery (body section):

```
<script src="https://code.jquery.com/jquery-3.7.1.slim.min.js"
integrity="sha256-
kmHvs0B+OpCW5GVHUNjv9rOmY0IvSIRcf7zGUDTDQM8="
crossorigin="anonymous"></script>
```



Note: This is required to make the Bootstrap CSS framework available to the entire application as well as to allow CORS requests to be made when retrieving this code.

10. Next, we need to copy the ***images*** folder from the Express public folder ***src/app/assets*** so the images are available to the Angular application.
    After copying the images, you will also need to modify the angular.json file so that it can find the images. Just above the 2 places that you updated the css code earlier in this file, there will be the following code:

```
"assets": [
    {
      "glob": "**/*",
      "input": "public"
    }
  ],
```

    Replace that with the following:

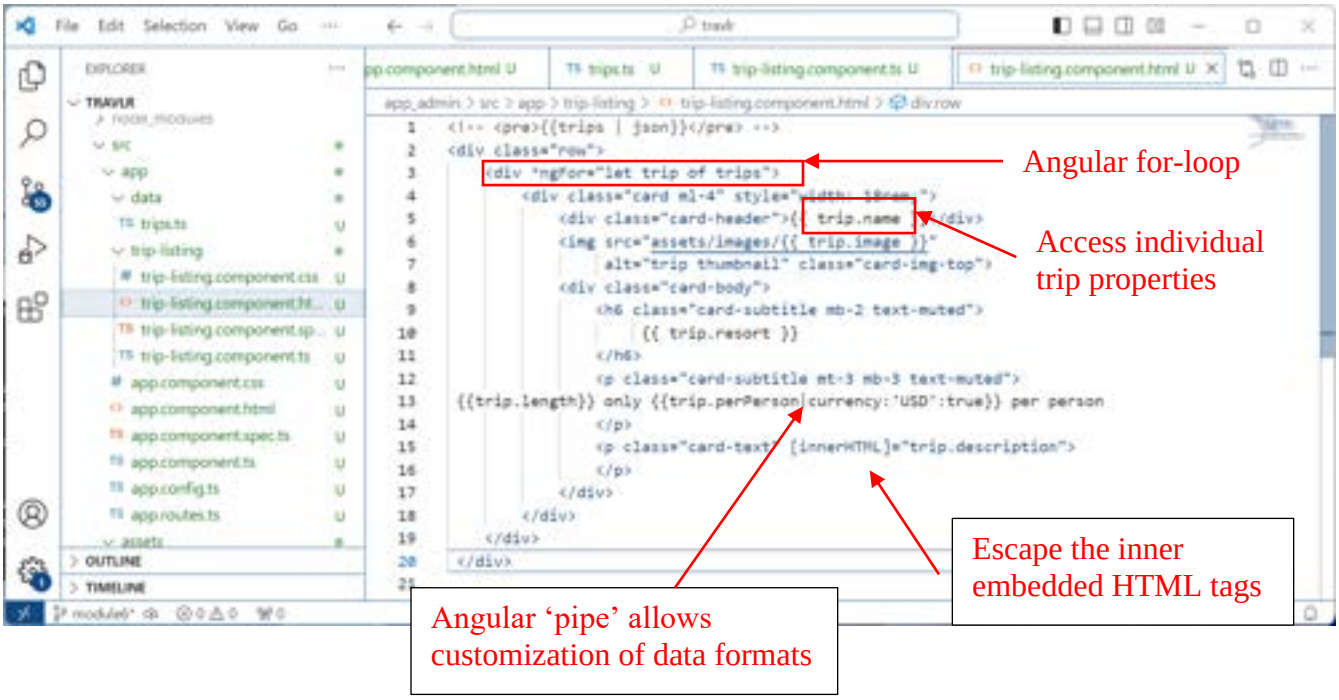
```
"assets": ["src/favicon.ico", "src/assets"],
```

    If you still have the ng serve command running, you will need to stop and restart it for this change to take effect.

11. The next set of edits belongs to the ***trip-listing.component.html*** file. This is where we replace the initial code that displayed the JSON data with a segment of HTML code that can be rendered for each of the data records provided by our ***trip-listing.component***.

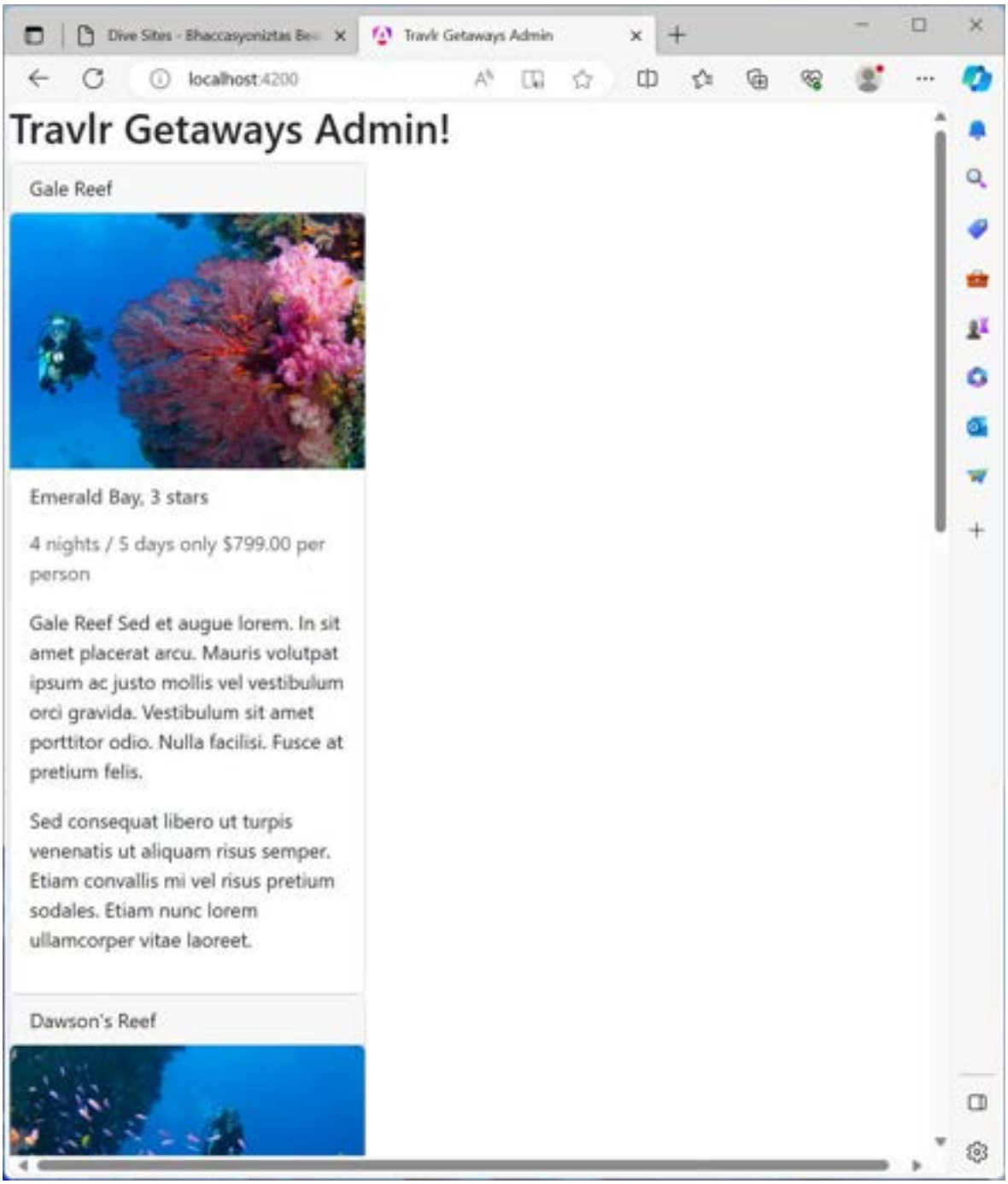
    The contents of the ***trip-listing.component.html*** file will be replaced with the following code:

```
<!-- <pre>{{trips | json}}</pre> -->
<div class="row">
    <div *ngFor="let trip of trips">
        <div class="card ml-4" style="width: 18rem;">
            <div class="card-header">{{ trip.name }}</div>
            <img src="assets/images/{{ trip.image }}"
                 alt="trip thumbnail" class="card-img-top">
            <div class="card-body">
                <h6 class="card-subtitle mb-2 text-muted">
                    {{ trip.resort }}
                </h6>
                <p class="card-subtitle mt-3 mb-3 text-muted">
 {{trip.length}} only {{trip.perPerson|currency:'USD':true}} per person
                </p>
                <p class="card-text" [innerHTML]="trip.description">
                </p>
            </div>
        </div>
    </div>
</div>
```

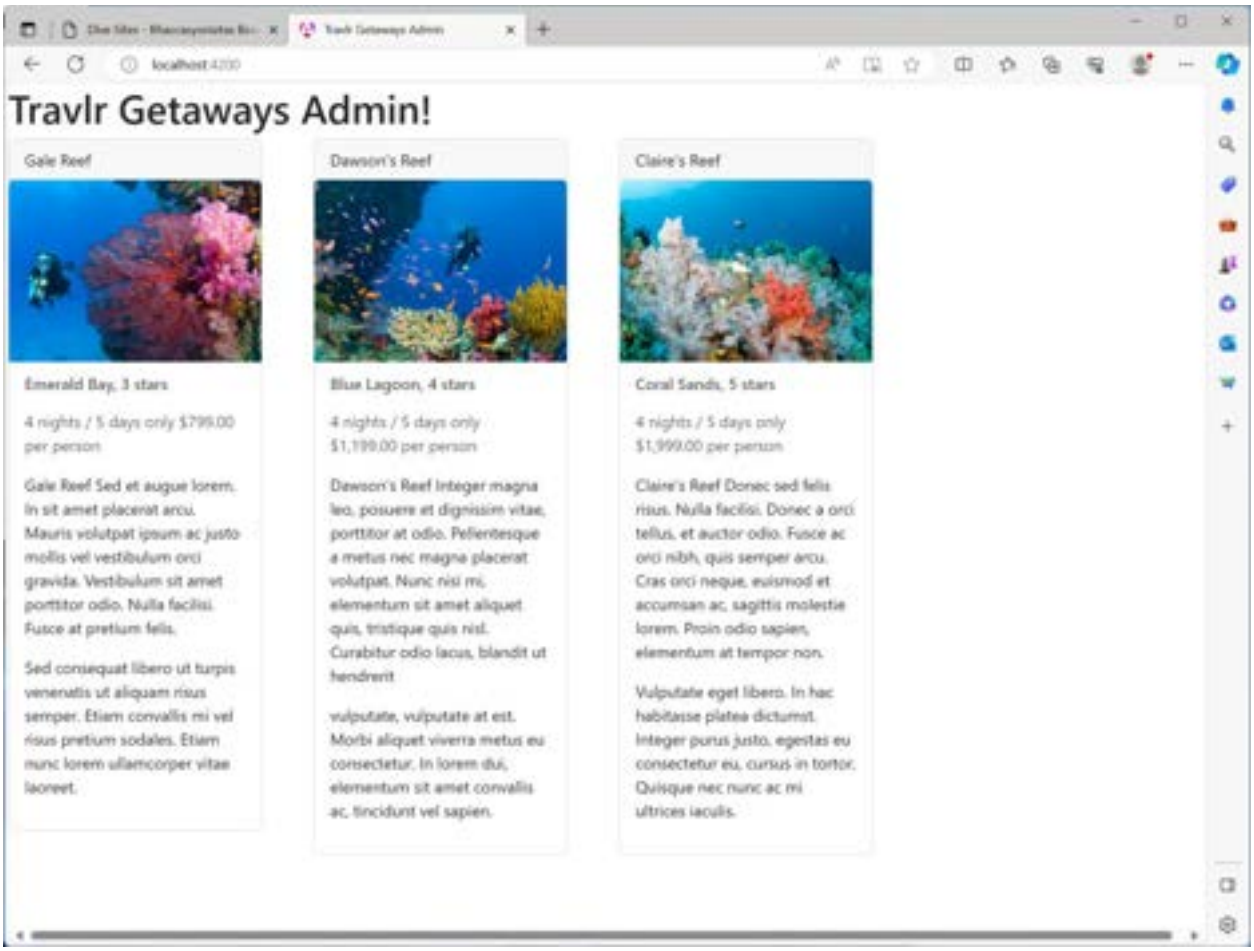


This code relies on the Bootstrap CSS framework to generate a pleasing view of the data. Please note: The cards will, by default, appear vertically in the window. To get them to appear horizontally, you need to make sure that you have enough horizontal space. Each row (in the

Bootstrap framework) consumes 12 columns which are proportional to the width of your browser window. When the file is saved will appear like this:

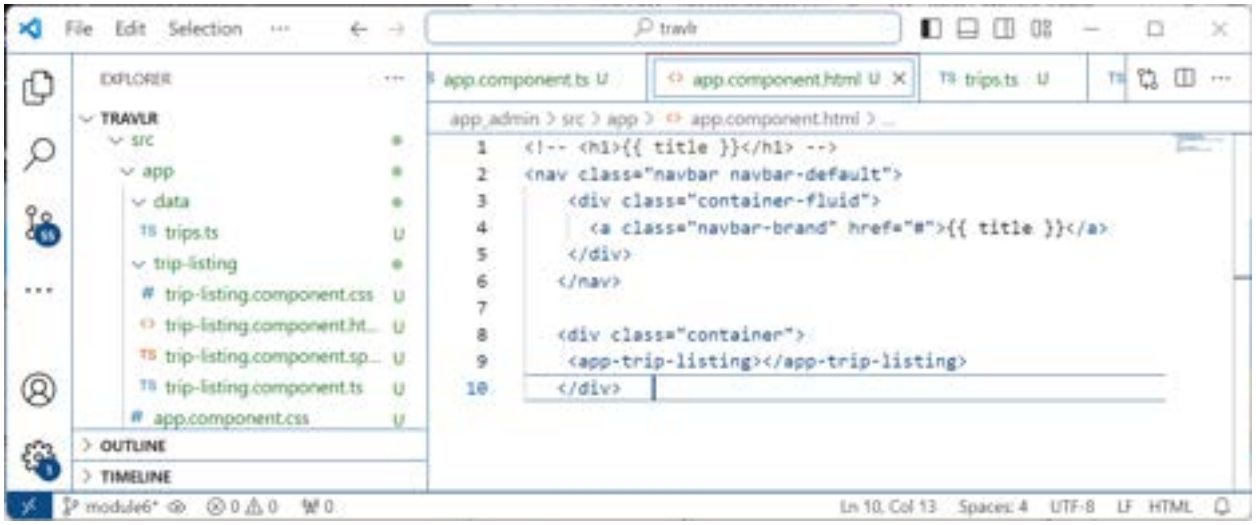


If you edit the '*ngFor' tag-set in the trip-listing.component.html file you can add a 'class' attribute that will set the number of columns that each card should consume. After 12 columns have been consumed the next cards will show beneath the first set of cards. As an example, editing the tag-set to read: <div *ngFor="let trip of trips" class="col-3"> - you should see the following result:

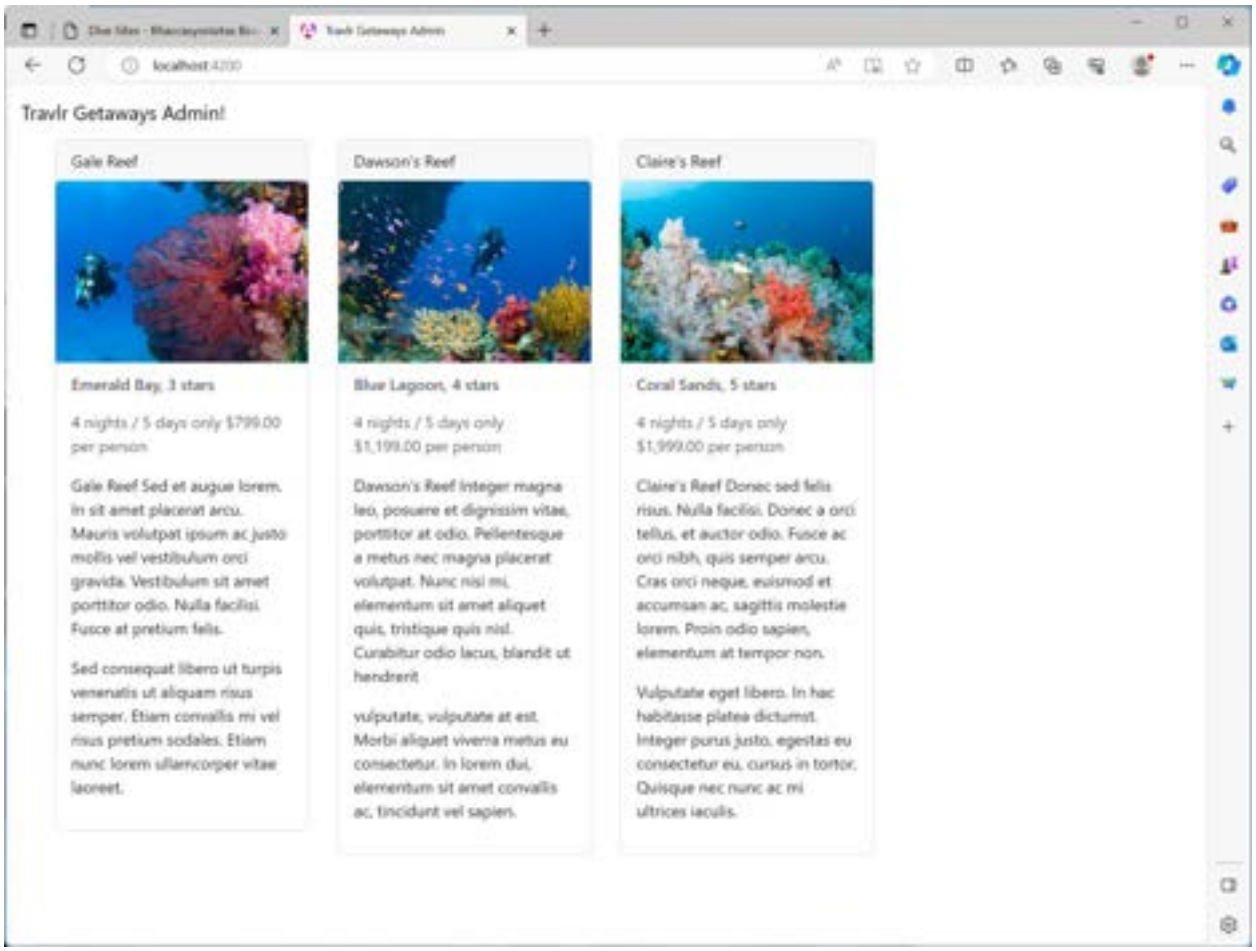


12. The ***app.component.html*** page can be cleaned up to render the listing with a navigation bar and some whitespace. To do this, we will replace the contents with:

```
<!-- <h1>{{ title }}</h1> -->
<nav class="navbar navbar-default">
 <div class="container-fluid">
   <a class="navbar-brand" href="#">{{ title }}</a>
 </div>
</nav>

<div class="container">
 <app-trip-listing></app-trip-listing>
</div>
```



And this will turn what had been a title to a hotlink to our admin page.



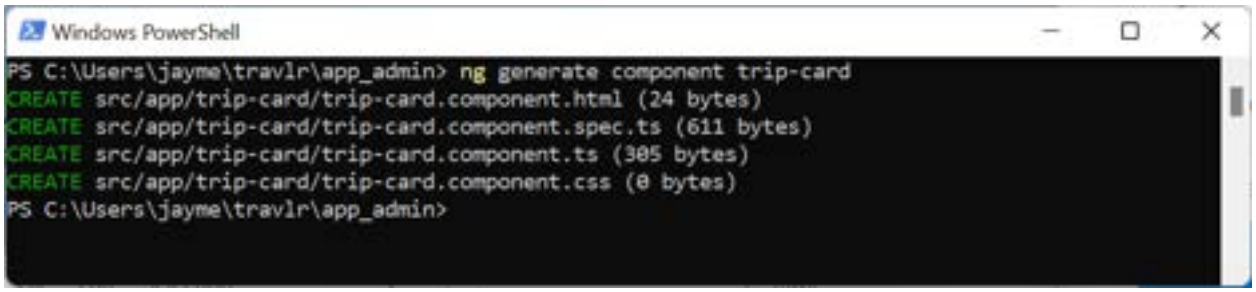
Refactor Trip Rendering Logic into an Angular Component

Now that we can see the trip listing in something other than an array of JSON objects things are really looking up for our application. However, there is still a fundamental problem: It can only show the trip listing one way, as cards. If we need to give the user the ability to toggle between a card view and a list view, it can quickly degenerate into a nightmare of coding to handle two different types of rendering for the same application with the same data.

However, if we pulled the card rendering logic out into a separate component, then the trip listing would only need to toggle the correct component (this can be accomplished with a selector-tag) to switch the layout. This technique of separating the logic so that one class only does one thing is referred to as Separation of Concerns or SoC – a powerful software engineering principle. This distinguishes itself by reducing single large 'mega-classes' to smaller easier to manage classes that only must account for a single responsibility.

1. We will begin the process of pulling the rendering into a separate component by generating a new component called trip-card. We will accomplish this from PowerShell within the ***app_admin*** directory, with the command:

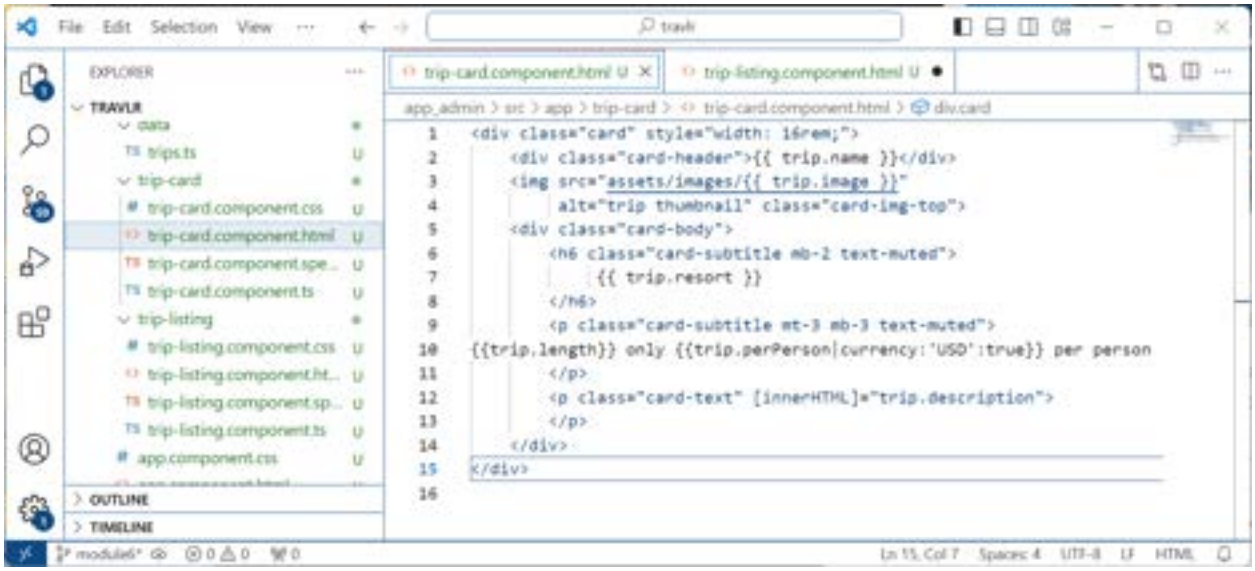
```
ng generate component trip-card
```



2. With the new component, we need to make some changes to a pair of files. First, we are going to pull the rendering code from the ***trip-listing.component.html*** file and placing it in the ***trip-card.component.html*** file. For this purpose, the trip-card.component.html file has the following contents:

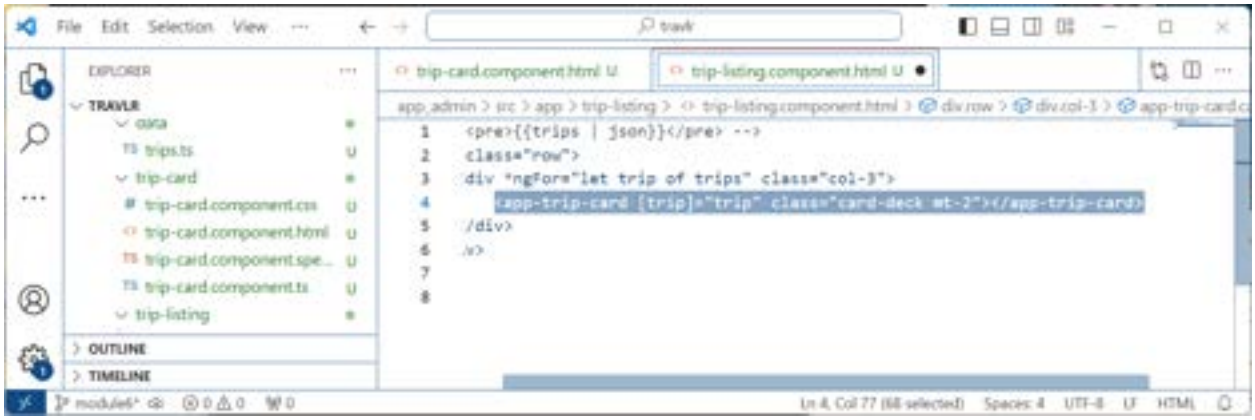
```
<div class="card" style="width: 16rem;">
    <div class="card-header">{{ trip.name }}</div>
    <img src="assets/images/{{ trip.image }}"
         alt="trip thumbnail" class="card-img-top">
    <div class="card-body">
        <h6 class="card-subtitle mb-2 text-muted">
            {{ trip.resort }}
        </h6>
        <p class="card-subtitle mt-3 mb-3 text-muted">
{{trip.length}} only {{trip.perPerson|currency:'USD':true}} per person
```

```
            </p>
            <p class="card-text" [innerHTML]="trip.description">
            </p>
        </div>
    </div>
```

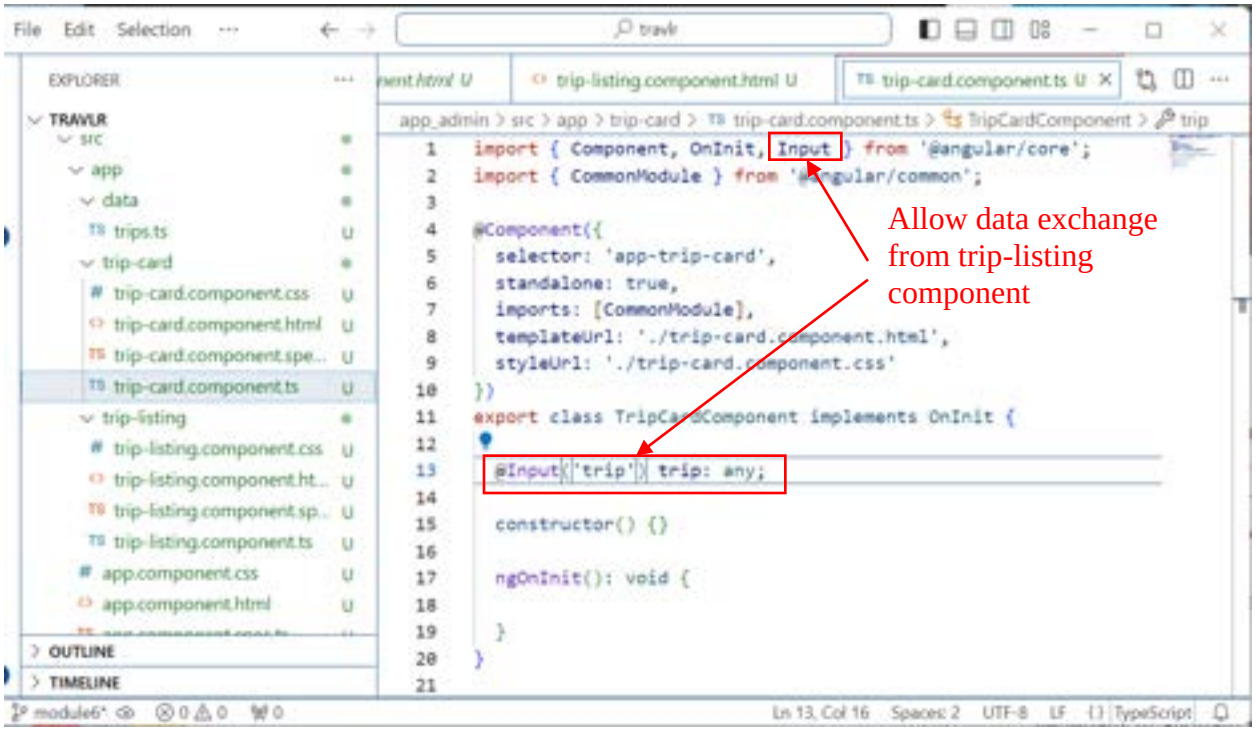


This section is replaced in the ***trip-listing.component.html*** file with:

```
<app-trip-card [trip]="trip" class="card-deck mt-2"></app-trip-card>
```



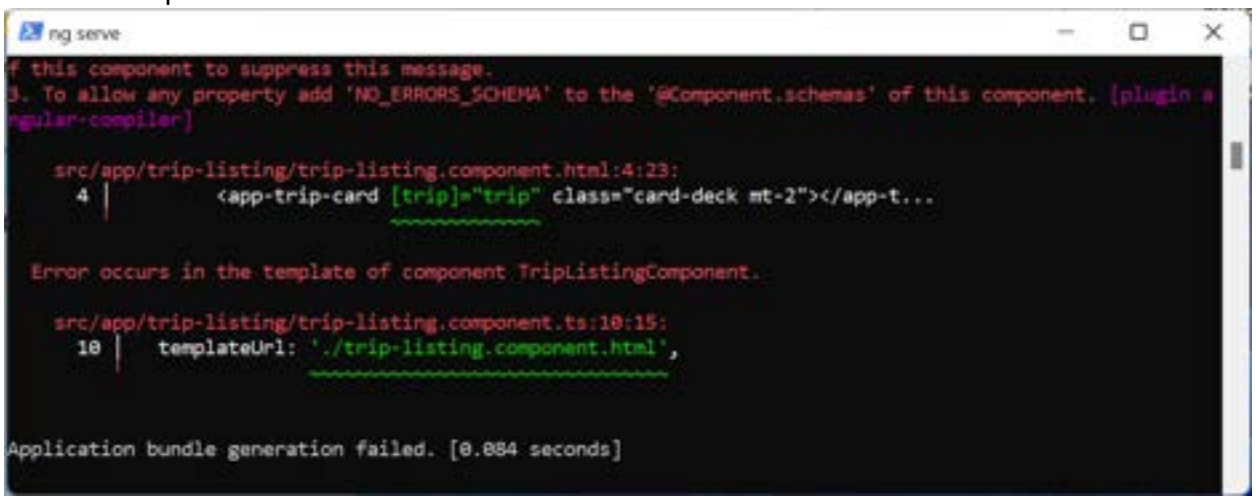
3. In order for this to get pulled into our application, we now have to add some code to the ***trip-card.component.ts*** file to add the *Input* directive in order that the *trip-listing* component can pass the *trip* data so that it can render an instance of a trip.



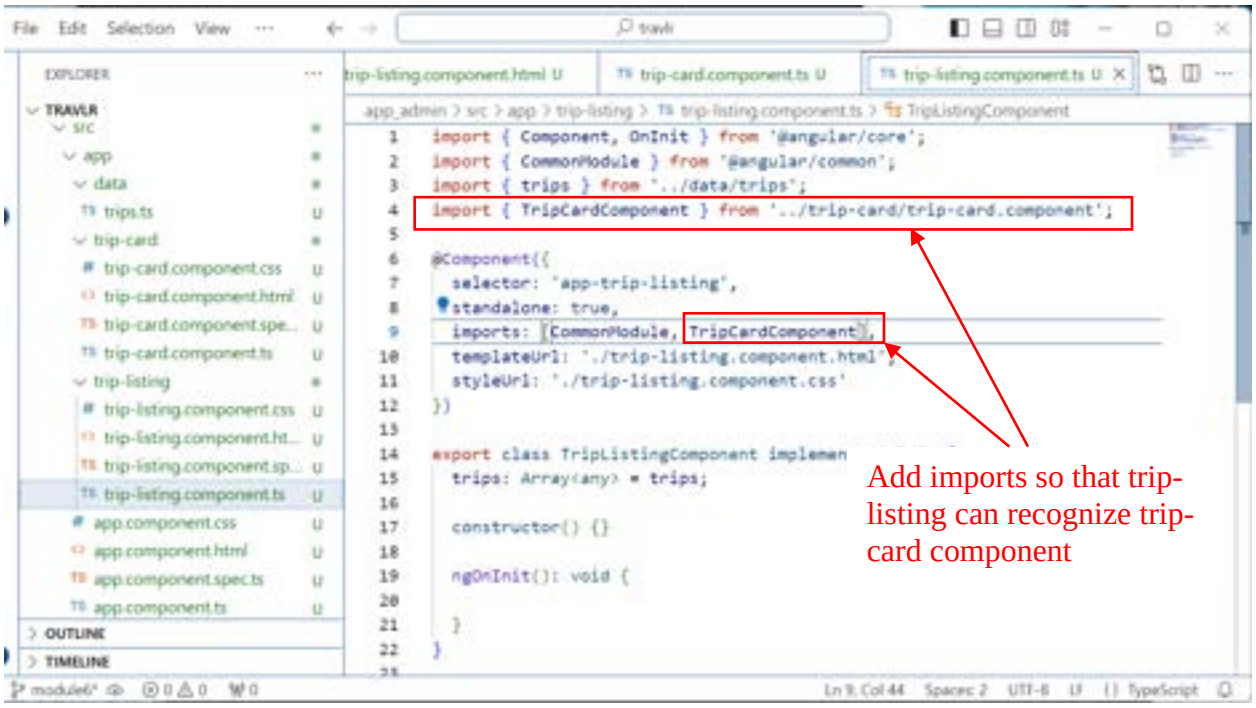
Your ***trip-card.component.ts*** file should contain this code:

```
import { Component, OnInit, Input } from '@angular/core';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-trip-card',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './trip-card.component.html',
  styleUrl: './trip-card.component.css'
})
export class TripCardComponent implements OnInit {

  @Input('trip') trip: any;

  constructor() {}

  ngOnInit(): void {

  }
}
```

4. If you have been paying attention to your console window, you will notice that there is an error at this point:



This is because we are calling a new component from our ***trip-listing*** component and ***trip-listing*** doesn't know what it is or what it can do. To fix this, we need to make one more change in the ***trip-listing.component.ts*** file.

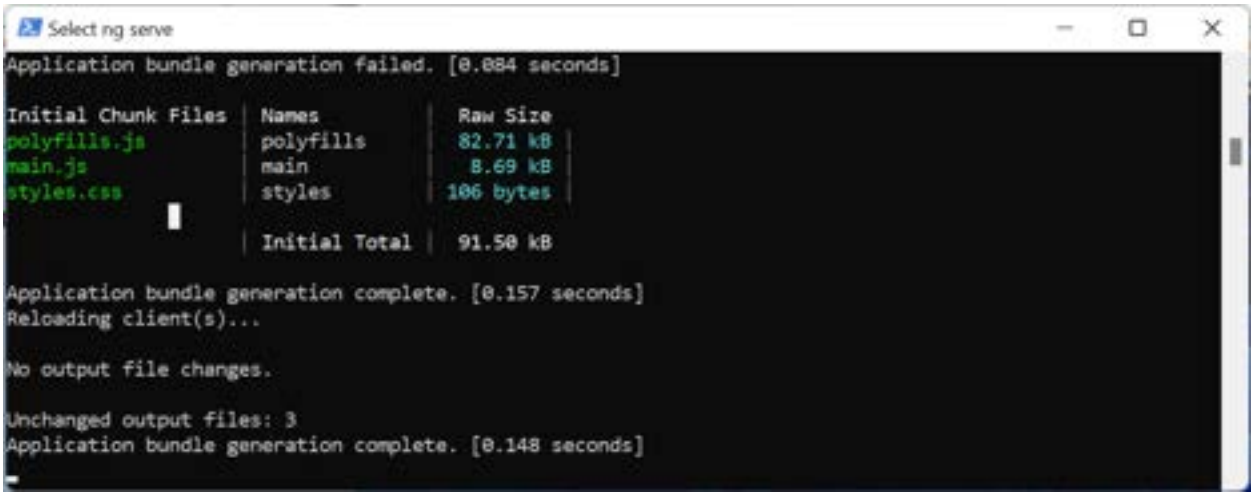


Your updated ***trip-listing.component.ts*** file should look like this:

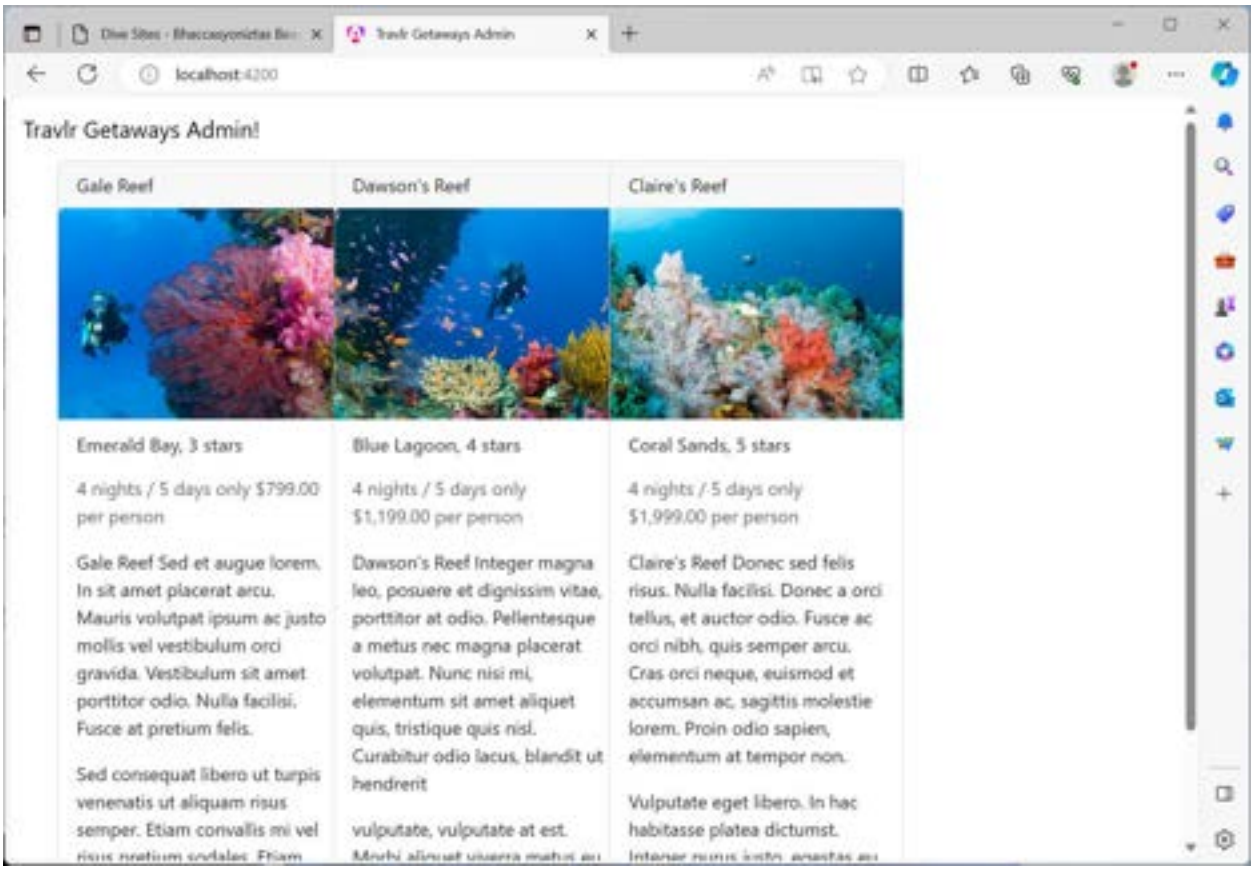
```
import { Component, OnInit } from '@angular/core';
import { CommonModule } from '@angular/common';
import { trips } from '../data/trips';
```

```
import { TripCardComponent } from '../trip-card/trip-
card.component';

@Component({
  selector: 'app-trip-listing',
  standalone: true,
  imports: [CommonModule, TripCardComponent],
  templateUrl: './trip-listing.component.html',
  styleUrl: './trip-listing.component.css'
})

export class TripListingComponent implements OnInit {
  trips: Array<any> = trips;

  constructor() {}

  ngOnInit(): void {

  }
}
```

Once these changes are made and saved you will see output like the following in your 'ng serve' PowerShell window:



And your output window for the application should look like this:



With these quick steps, we have created a component that handles the rendering of a single trip and refactored our existing code to simplify the application. Additionally, the component can be used anywhere in the application.

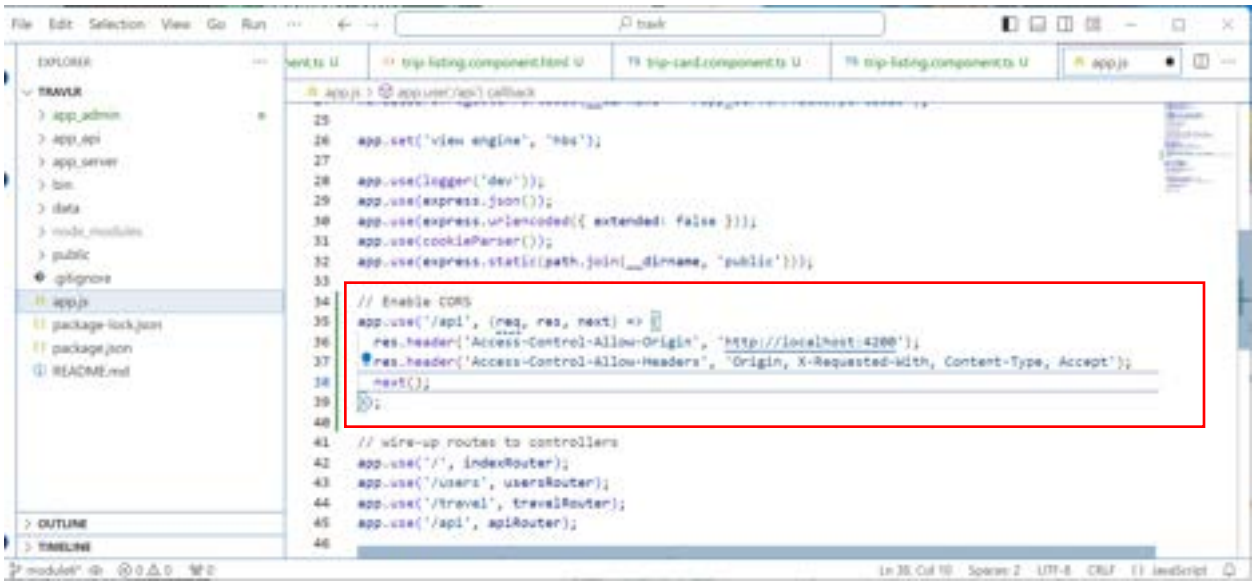
Create Trip Data Service

Now that we have seen what it takes to create the necessary components to render our trips, we need to start thinking about where the data is going to come from. In a single-page application (SPA), it is increasingly common to obtain information from another part of the application, particularly with distributed (or multi-tier) architectures. In a SPA, this generally means calling a REST endpoint to obtain data. Angular supports Separation of Concerns (SoC) by defining a "service" to contain functions or objects that are used by components to perform actions.

In Module 5 we created an "/api" REST endpoint on our backend application using Express. This was designed to handle data requests, and we were successfully able to test that with our application. Now we will create an Angular service to handle access to the backend endpoint for trip information.

1. The first step in allowing the Angular admin site to make calls against the Express backend API is to adjust the Express application to allow for the external calls. This change is made in

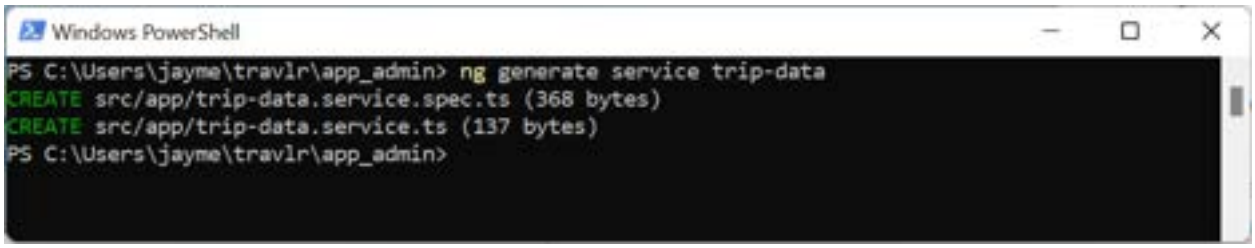
the ***app.js*** file at the root of our application tree and it enables what is generally referred to as CORS (Cross Origin Resource Sharing).



You can refer to the section "Allowing CORS requests in Express" in Chapter 9 of the textbook for additional information.

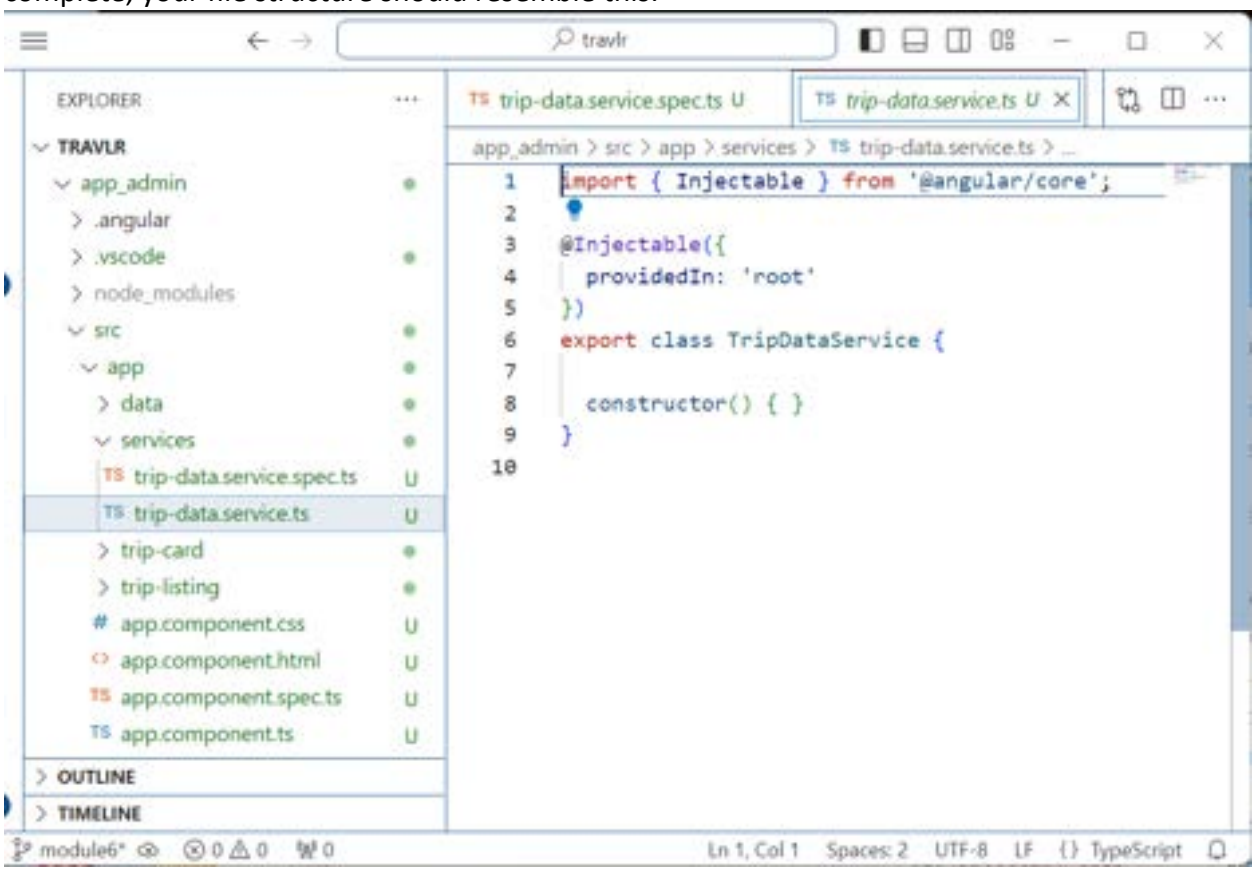
2. Now we will need to create a new Angular service called trip-data to allow us to configure the appropriate data paths. The command we need to use to accomplish this is:

```
ng generate service trip-data
```

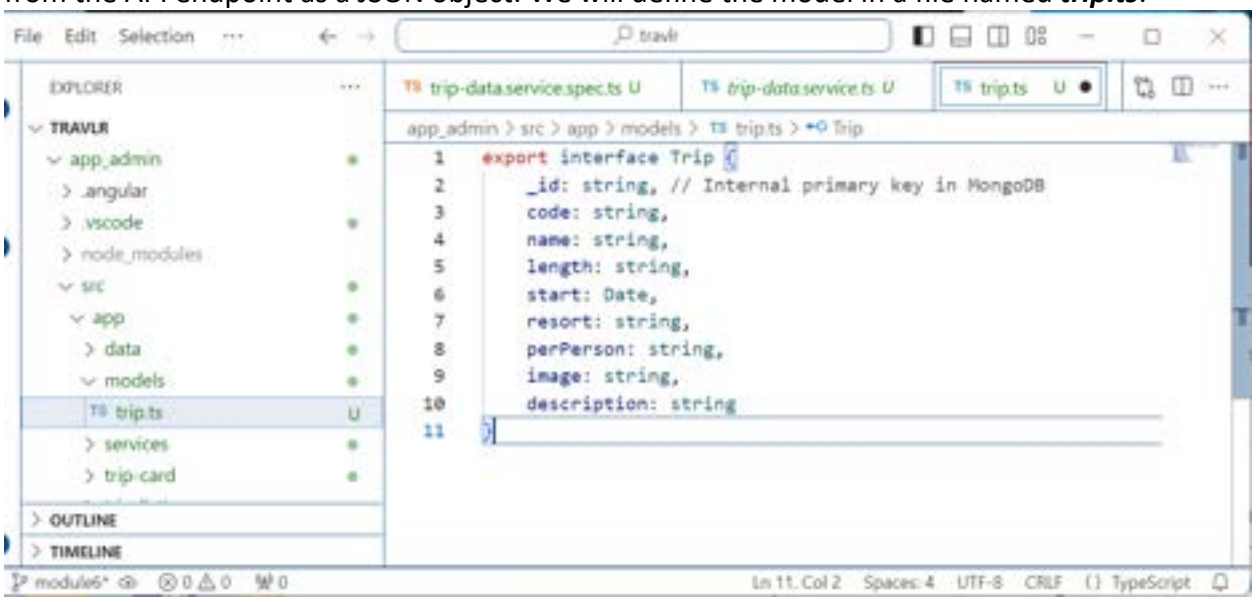


3. At this point we need to do some house-keeping in order to prepare the application structure for additional services later on. We will want to create a new folder under ***app_admin/src/app*** called '***services'*** and move the ***trip-data.service.**** files from the ***app_admin/src/app*** folder into the new ***app_admin/src/app/services*** folder. When

complete, your file structure should resemble this:



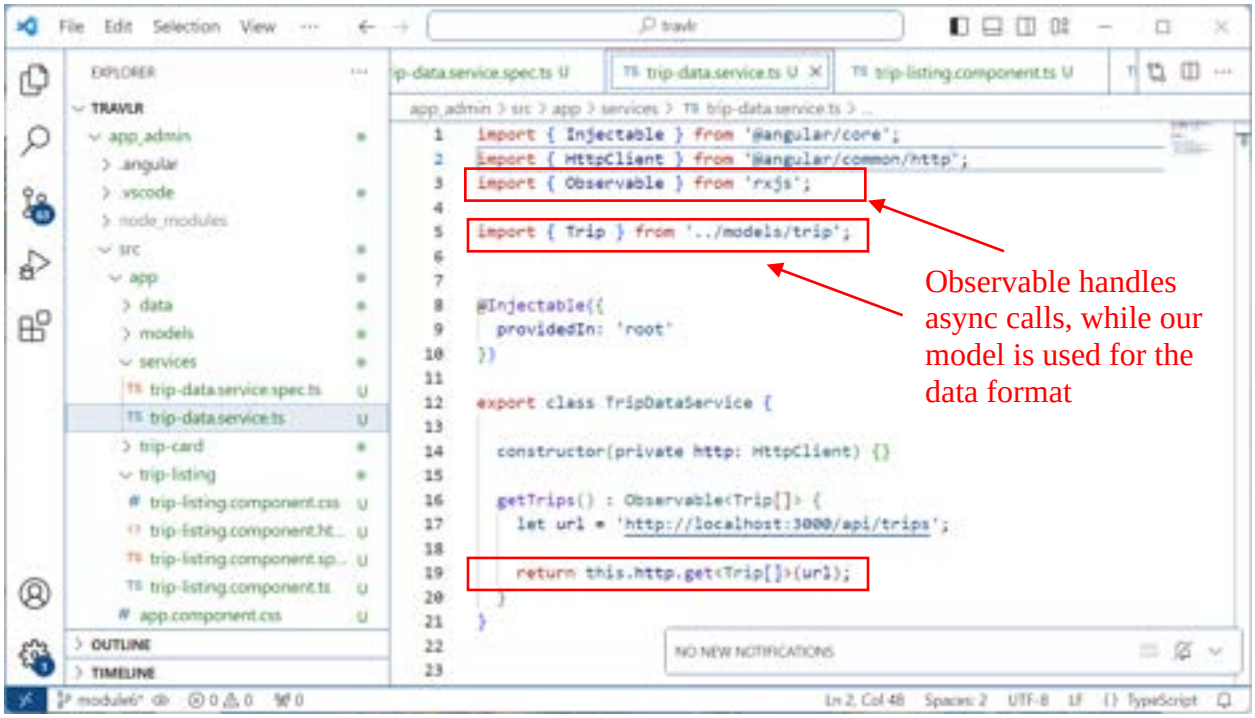
4. In order for our new basic service to function properly, we need some way to communicate what type of data it is going to handle. For this we need to create a ***models*** folder. We are going to put this parallel to the ***data*** and ***service*** directories under ***app_admin/src/app***. This is where we will create an interface to define the data for a single Trip that will be received from the API endpoint as a JSON object. We will define the model in a file named ***trip.ts***.



Instances of this interface will be used to transfer HTML form data to the component for rendering as well as between components and the REST endpoint. Angular will automatically marshal the data back and forth between JSON and JavaScript objects.

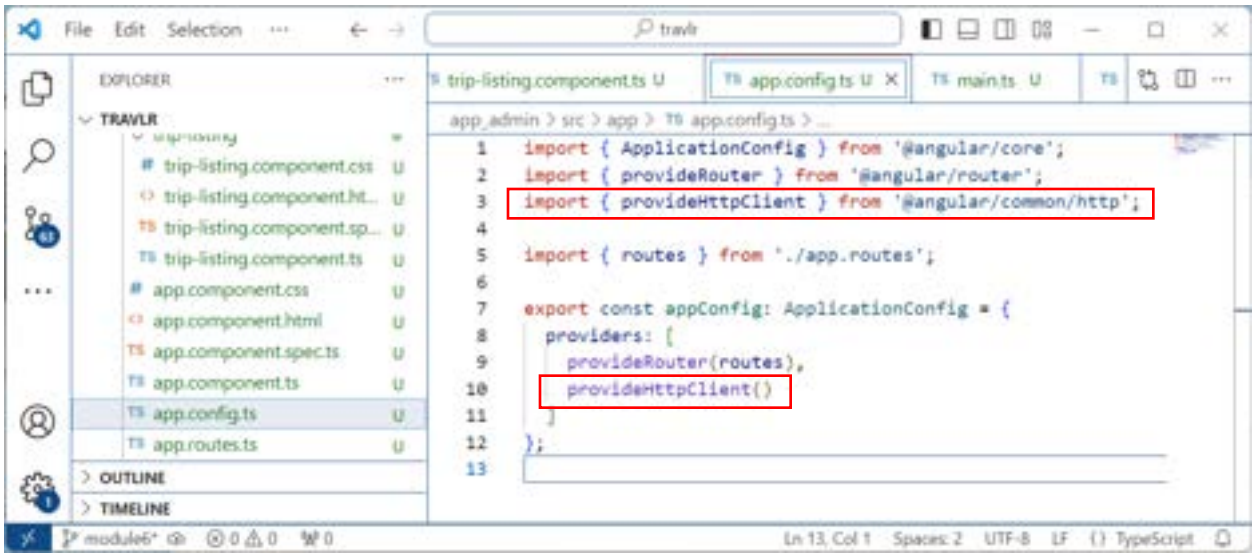
5. Now that we have the interface defined, we need to introduce it to the service. In order to accomplish this, we need to edit the ***trip-data.service.ts*** file and add some code. This is going to be very straight-forward because we aren't going to introduce error handling on the connection at this time, but we are going to introduce another module for import because the HTTP connection in the service is *asynchronous*.

   You should recall that we had to treat the internal API calls a bit differently in the Express application because they were issued asynchronously in Node JS. The same item applies here with Angular, but the implementation is slightly different. The module that we are going to introduce is '*Observable*' from the *rxjs* package.



   Because we are returning an Observable object, our component can attach to that object and get notification when the async call has been completed and the associated promise fulfilled.

6. Before we move forward with making the change to the ***trip-listing.component.ts*** file we need to make one small change to our application so that our new data service will be functional. We have to enable the HTTP services at the application level. To do this, we will add two lines to the ***app.config.ts*** file in our ***app_admin/src/app*** folder.



The two lines we added to the file allow the Angular environment to communicate via HTTP. Without these lines, our new service would not be able to function.

7. Now we need to make some changes to the ***trip-listing.components.ts*** file so that we can take advantage of our new service, and pull our data from the database instead of our local data file. The changes include:
    a. Importing our Trip model.

```
import { Trip } from '../models/trip';
```

    b. Importing our TripDataService

```
import { TripDataService } from '../services/trip-data.service';
```

    c. Registering TripDataService as a provider

```
  providers: [TripDataService]
```

    d. Creating a constructor to initialize the TripDataService

```
  constructor(private tripDataService: TripDataService) {
    console.log('trip-listing constructor');
  }
```

    e. Creating a method that will call the getTrips() method in TripDataService

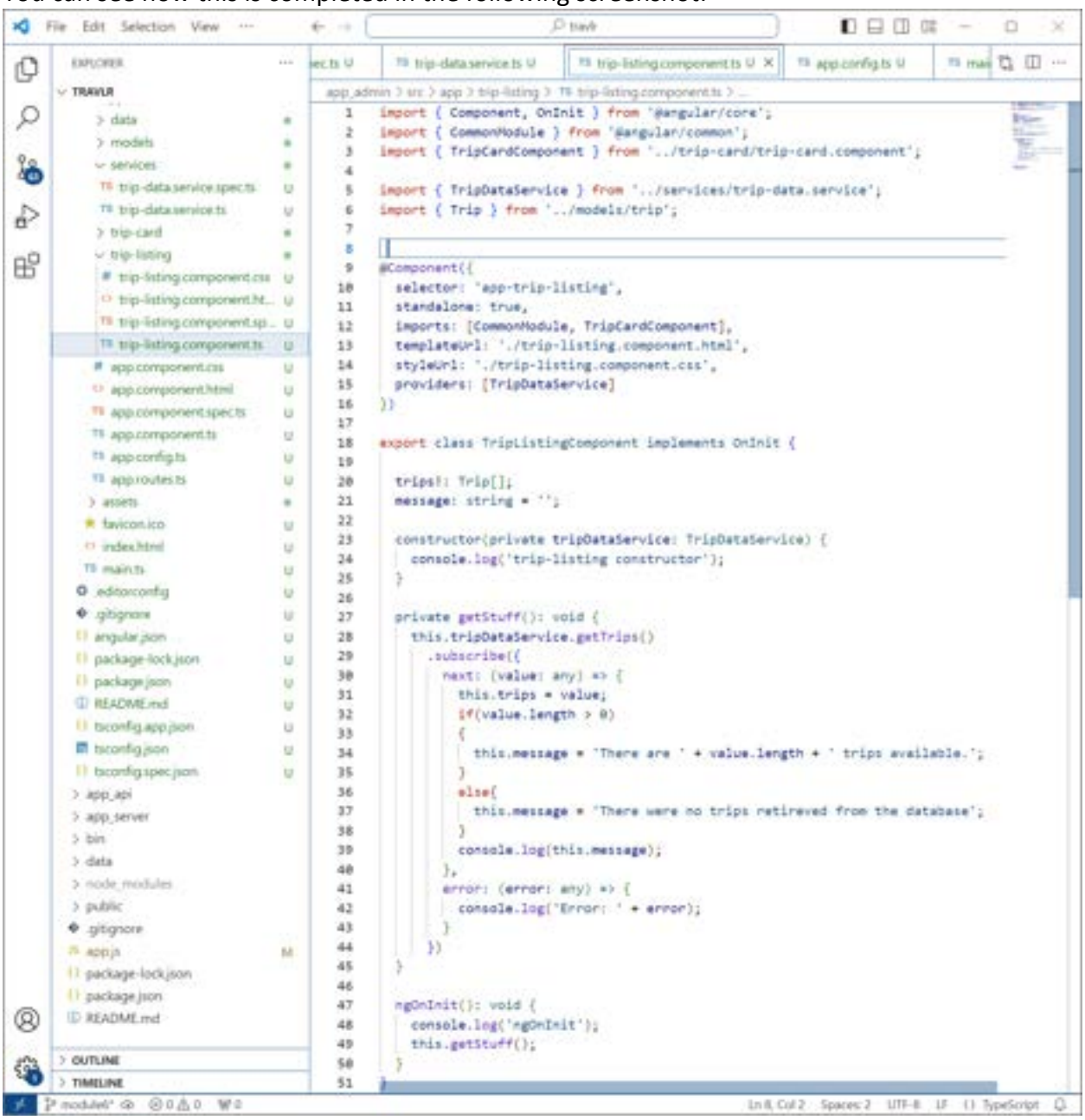
```
  private getStuff(): void {
    this.tripDataService.getTrips()
      .subscribe({
```

```
        next: (value: any) => {
          this.trips = value;
          if(value.length > 0)
          {
            this.message = 'There are ' + value.length + ' trips
available.';
          }
          else{
            this.message = 'There were no trips retireved from the
database';
          }
          console.log(this.message);
        },
        error: (error: any) => {
          console.log('Error: ' + error);
        }
      })
  }
```

f. And creating an ngOnInit method that will call our private method when the component is initialized.

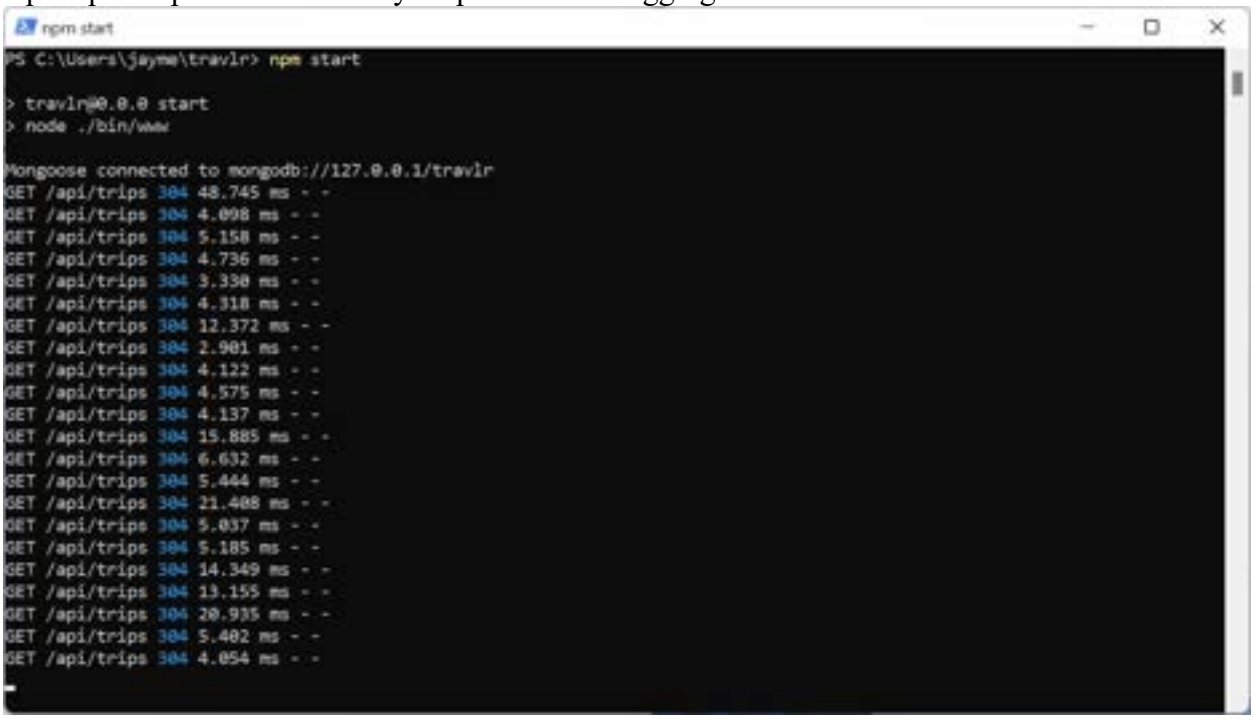
```
  ngOnInit(): void {
    console.log('ngOnInit');
    this.getStuff();
  }
```

You can see how this is completed in the following screenshot:

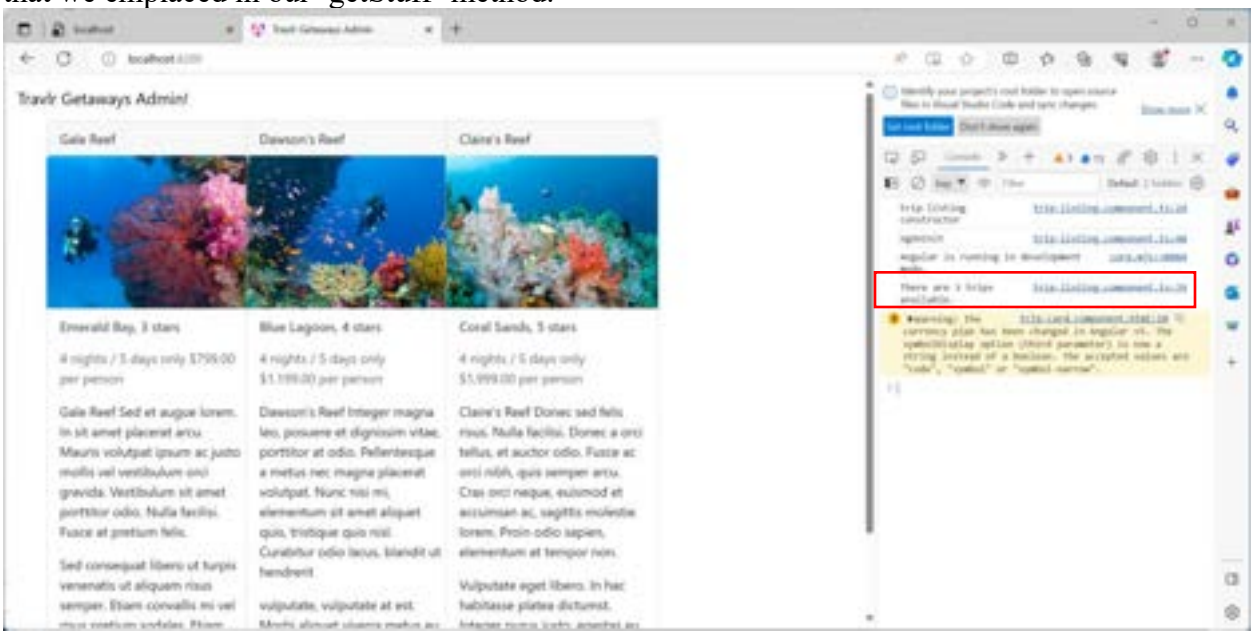


If you haven't already done so, make sure that you have your Express app running in a PowerShell Window. From this console, you can see each time an API call is made to the

/api/trips endpoint – this is very helpful with debugging.

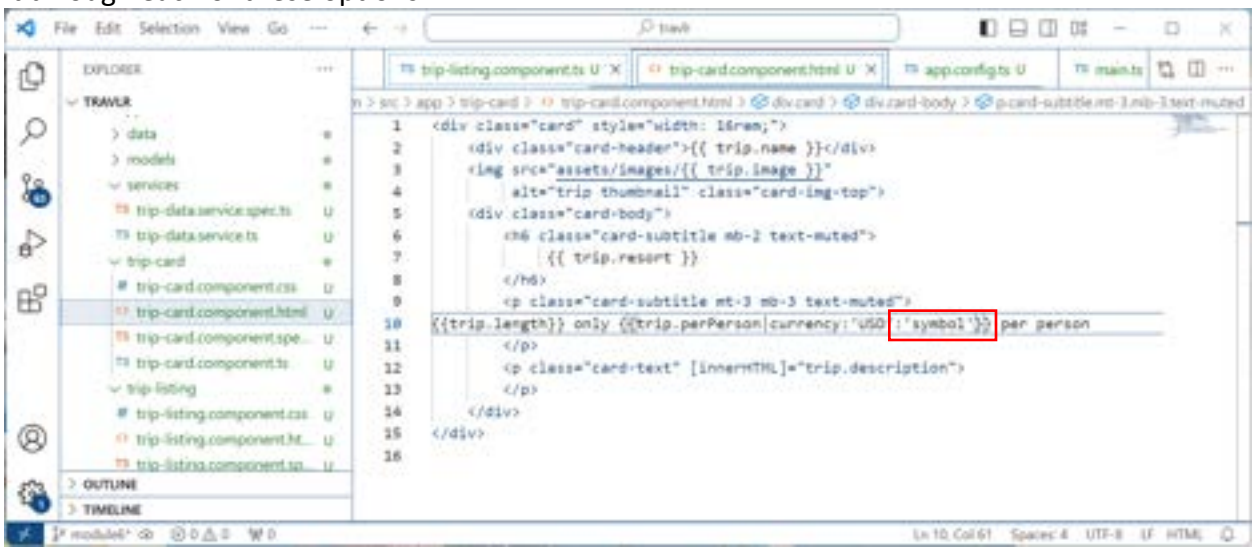


Additionally, if you haven't already, you should open the developer tools panel for your web-browser. In the Microsoft-Edge Browser, you can achieve this with CTRL-Shift-I. This is a very, very useful method to assist with debugging Angular applications – as you can watch the local console and output *console.log* messages to provide debugging information for your development process. Here you can see the results of the message that we emplaced in our 'getStuff' method.

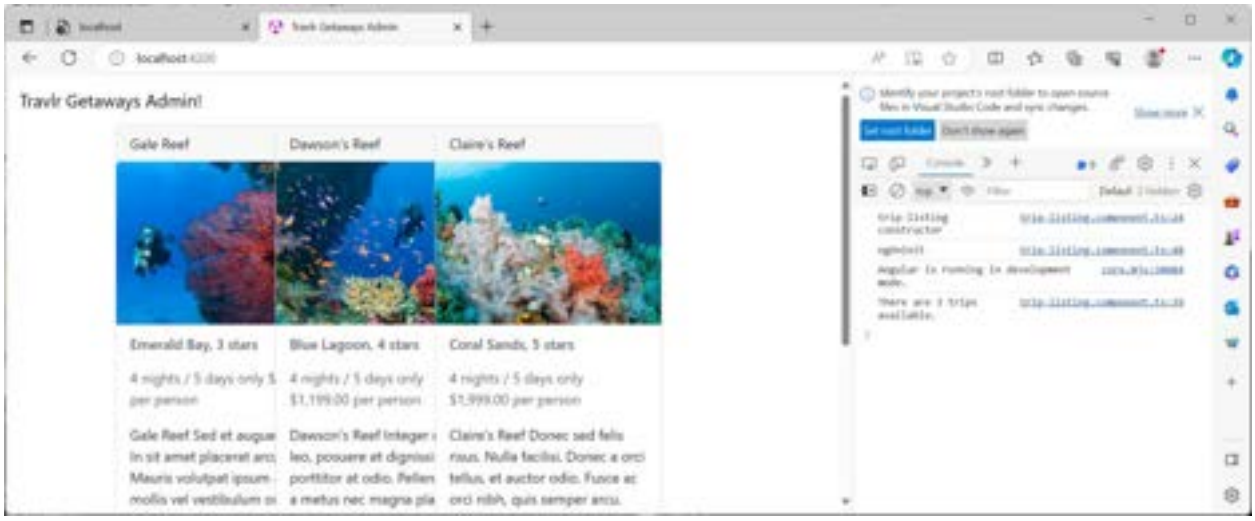


You can also see warnings in this window. In this case, we find that there is an issue with the currency pipe that we are using in the ***trip-card.component.html*** file. It is not uncommon for newer versions of Angular to deprecate (make obsolete) items from previous versions. When we find these items, we need to fix them.

8. Fixing the error that we have found when inspecting the results of our service is very straightforward. The offending issue is with the pipe in the ***trip-card.component.html*** file which formats the data as currency. In the previous instance, the third argument to the pipe was a Boolean value that indicated that the conversions should be done. In this case it becomes one of three flags: 'code', 'symbol', or 'symbol-narrow'. For our case, we will make it 'symbol', but you should feel free to experiment and see what happens when you change it through each of these options.



And now, you will note, that while the display is the same, we have no errors in the developer's console:

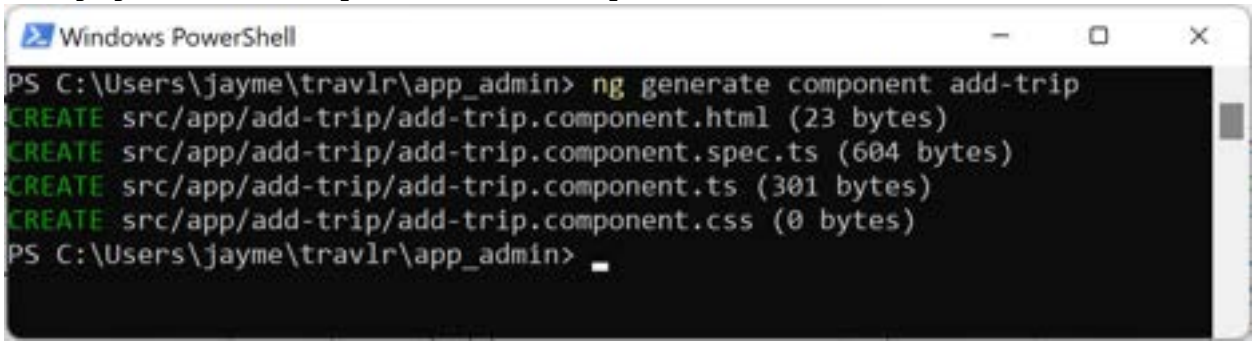


Add Trips

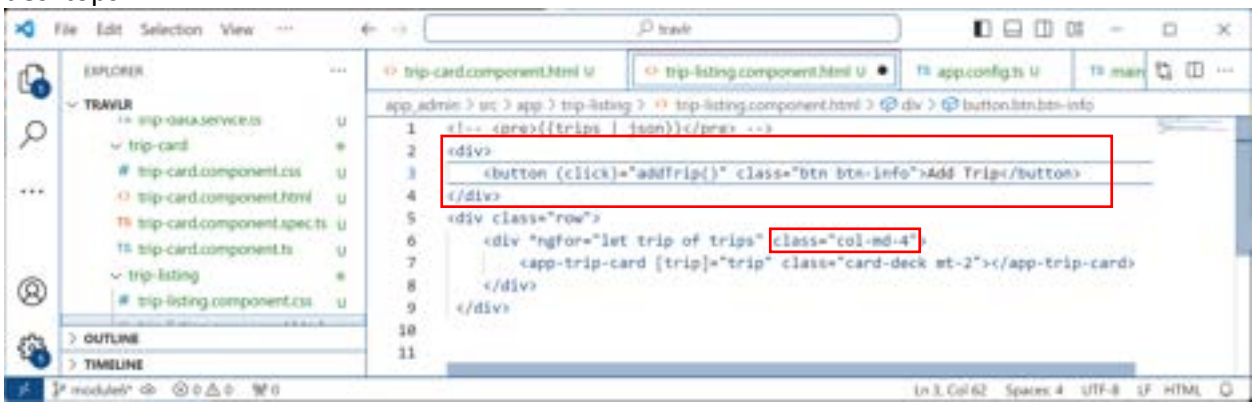
To this point, we have created an application that will display our trips, but if we wanted to make any changes to a trip definition or add a new trip, we would need to go and manually edit our Mongo Database. That really isn't practical in a production application, so we are going to add some code that will allow us to add and edit trip information. This process is going to require us to add new components, forms, and routes, as well as updating the logic on the backend to add additional API capability for our /api endpoint to support these new capabilities.

1. We will begin by creating a new Angular Component called ***add-trip*** that will support the addition of a new trip to our application.

```
ng generate component add-trip
```

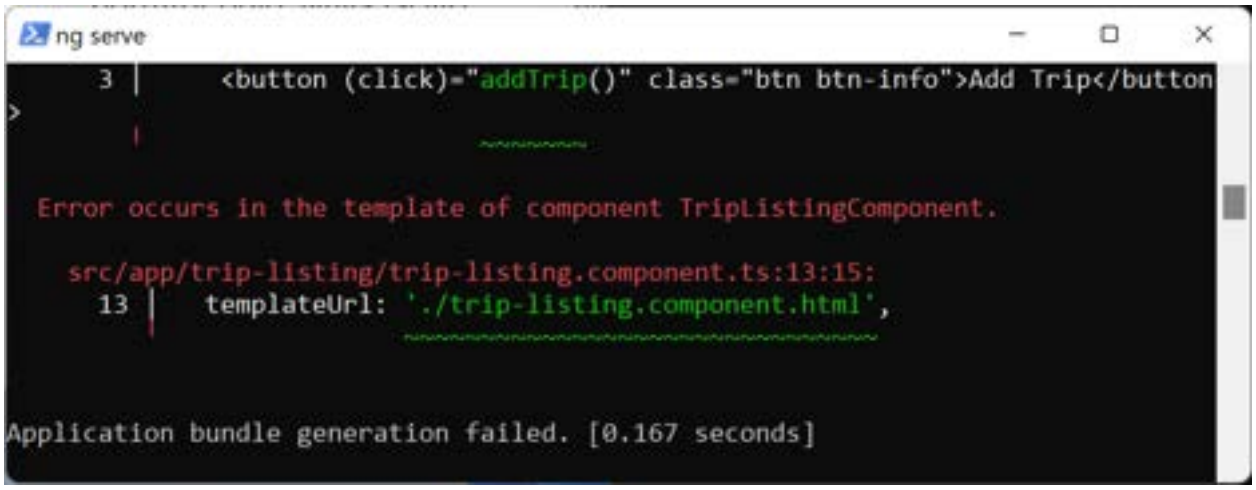


2. The next thing we are going to do is to make a change to our ***trip-listing.component.html*** file to add a button that we will utilize to select the add-trip functionality for our application. At the same time, we are going to make a small change to the column definition for our application to make the display more pleasing and make sure that the cards will flow properly for both changing screen sizes and utilization on mobile devices as well as desktops.



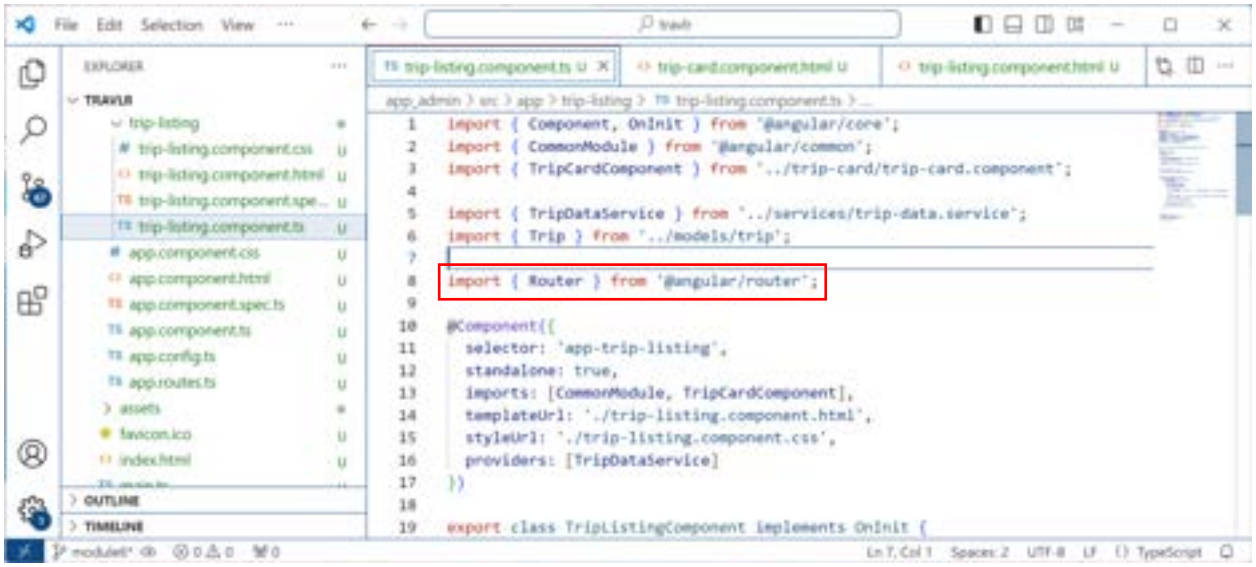
You will notice that the button is set to execute an addTrip() method when pressed. We will be wiring this up in the coming steps. The change to the class definition for our trip selector specifies to Bootstrap that each card will consume 4 of 12 columns on a medium or larger screen (768 pixels or greater) and will wrap (stacking the cards vertically) on anything smaller.

3. However, we cannot currently see the impact of our update, because as we save this file we are informed that the addTrip() property does not exist for our TripListingComponent.



So the next step becomes modifying the ***trip-listing.component.ts*** file to add our new capability.

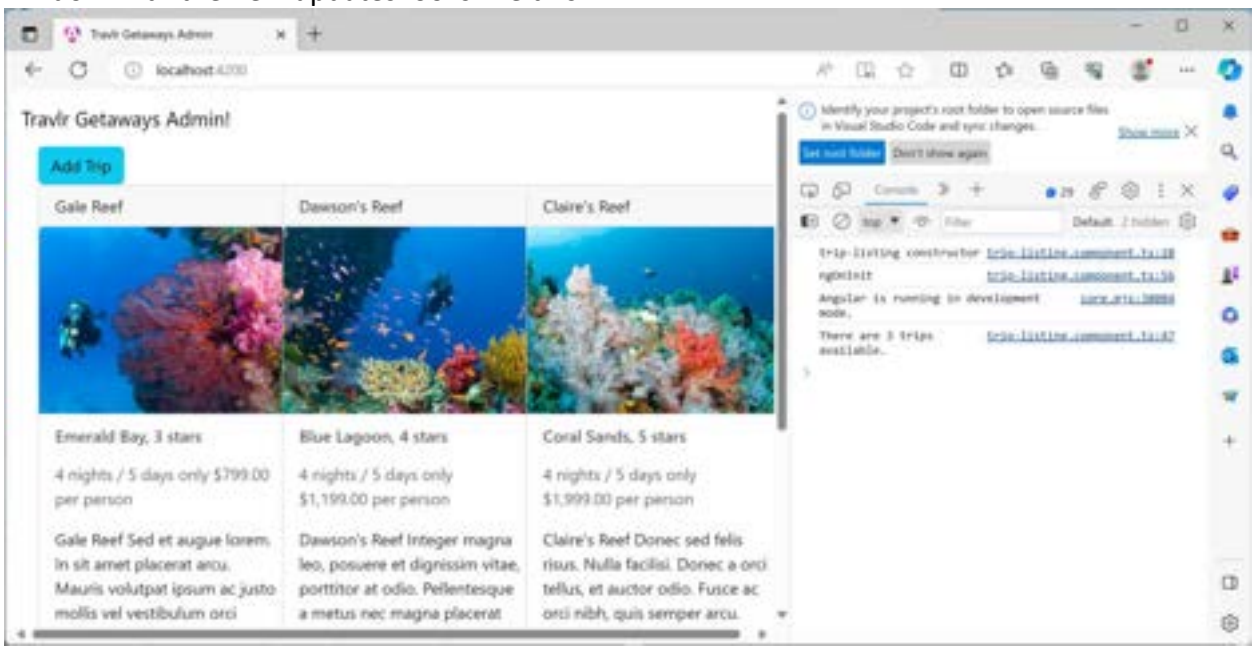
4. We are going to make three changes in this file, we need to add an import statement to bring in the routing capability, we need to update the constructor to initialize the routing capability, and we need to add an addTrip() method that will support our new button. This is our import statement:



This pulls in the Angular routing capability.

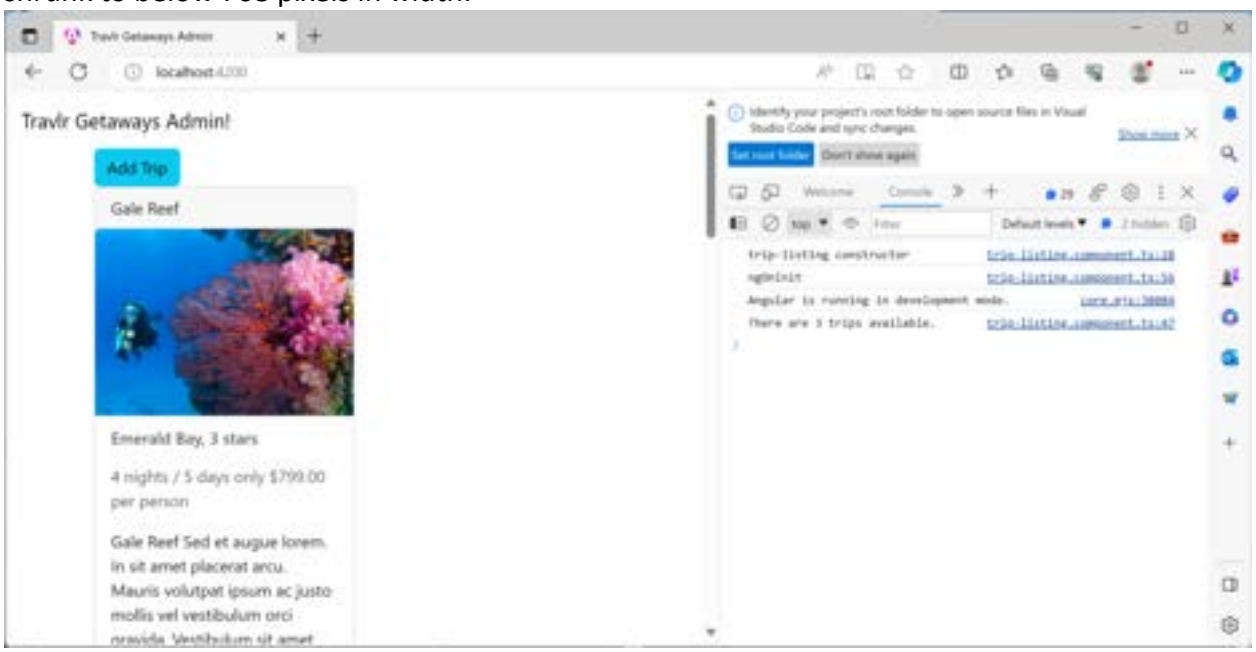
The final two changes are shown here:

Please note that we need to make the addTrip() method public because it is going to need to be accessed externally from the perspective of the trip-listing component. At this point, our window with the new updates looks like this:

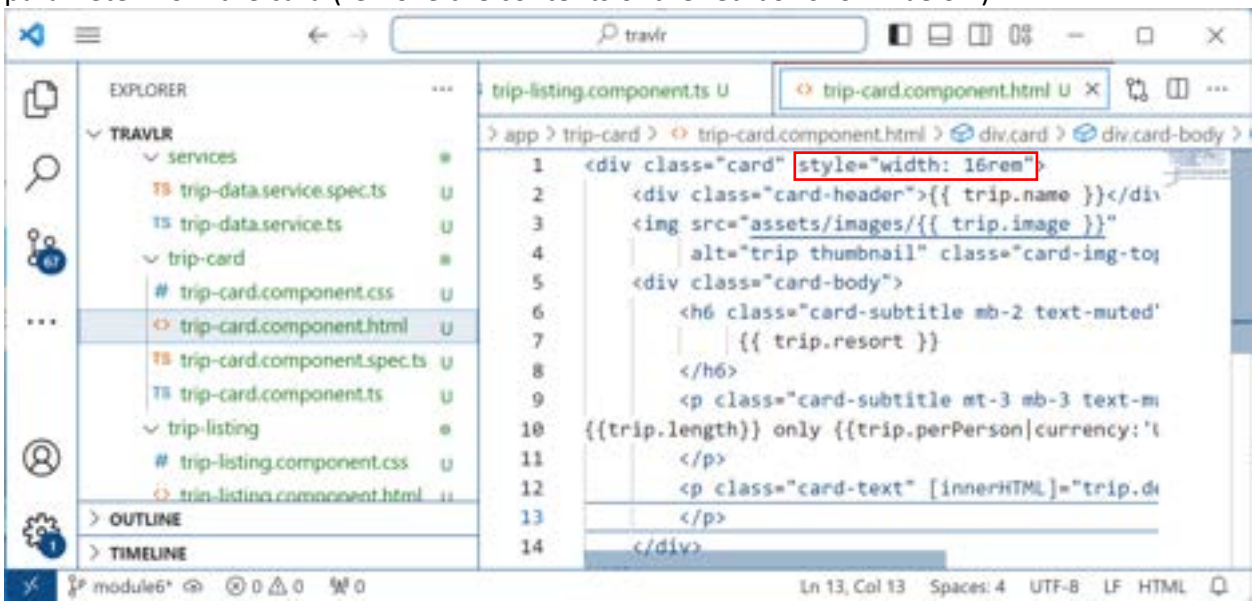


This is visually pleasing, but if you manipulate the screen size and shrink it, you will notice that the on a smaller screen with vertically stacked cards, the cards have the same width and the scale doesn't look quite right. The following is what we see with the viewport

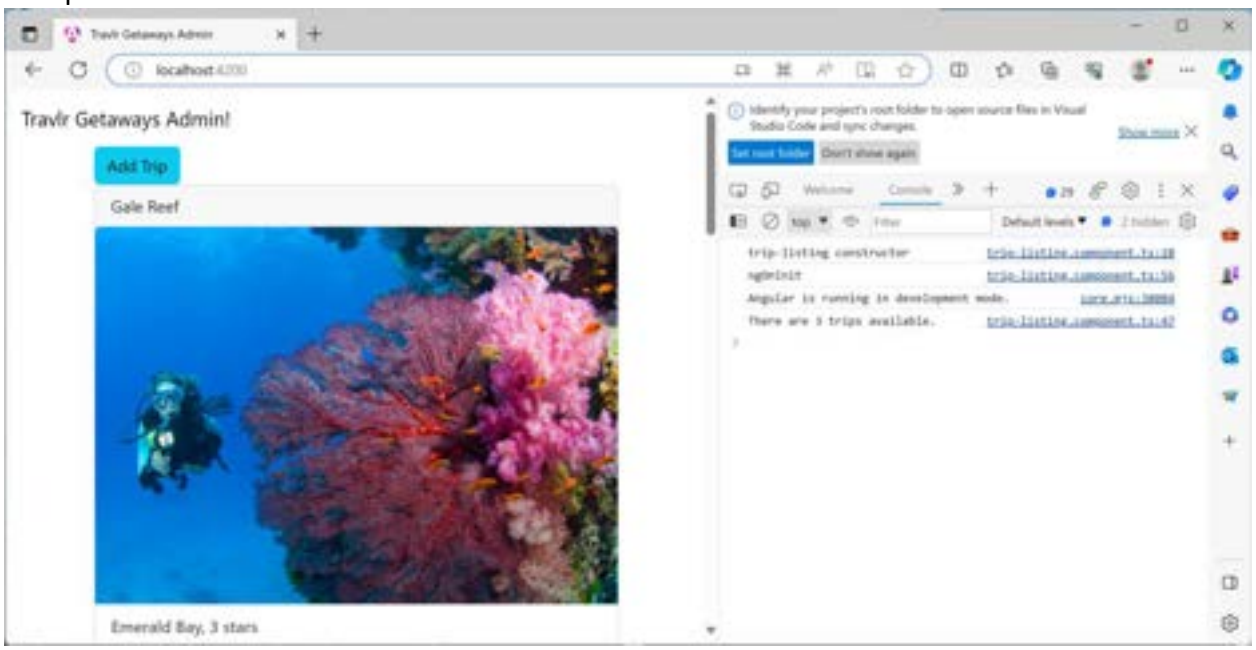
shrunk to below 768 pixels in width:



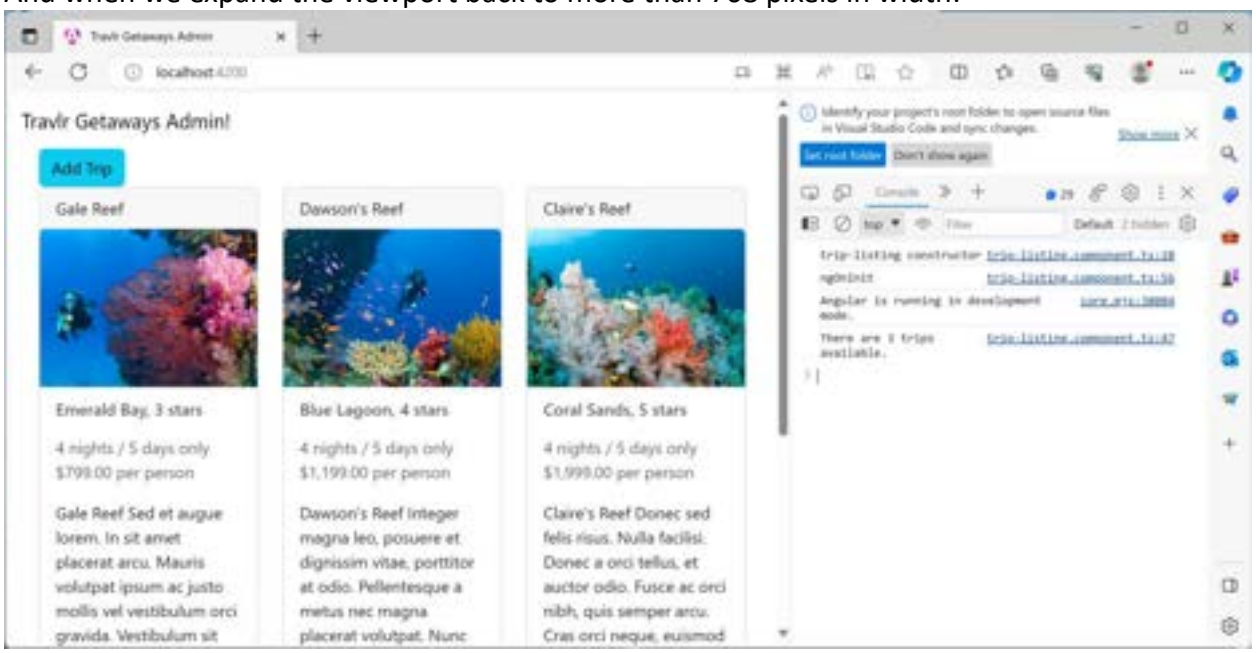
We can make one more small modification to improve things and make them scale dynamically. So let's edit the ***trip-card.component.html*** file and remove the width parameter from the card (remove the contents of the red box shown below):



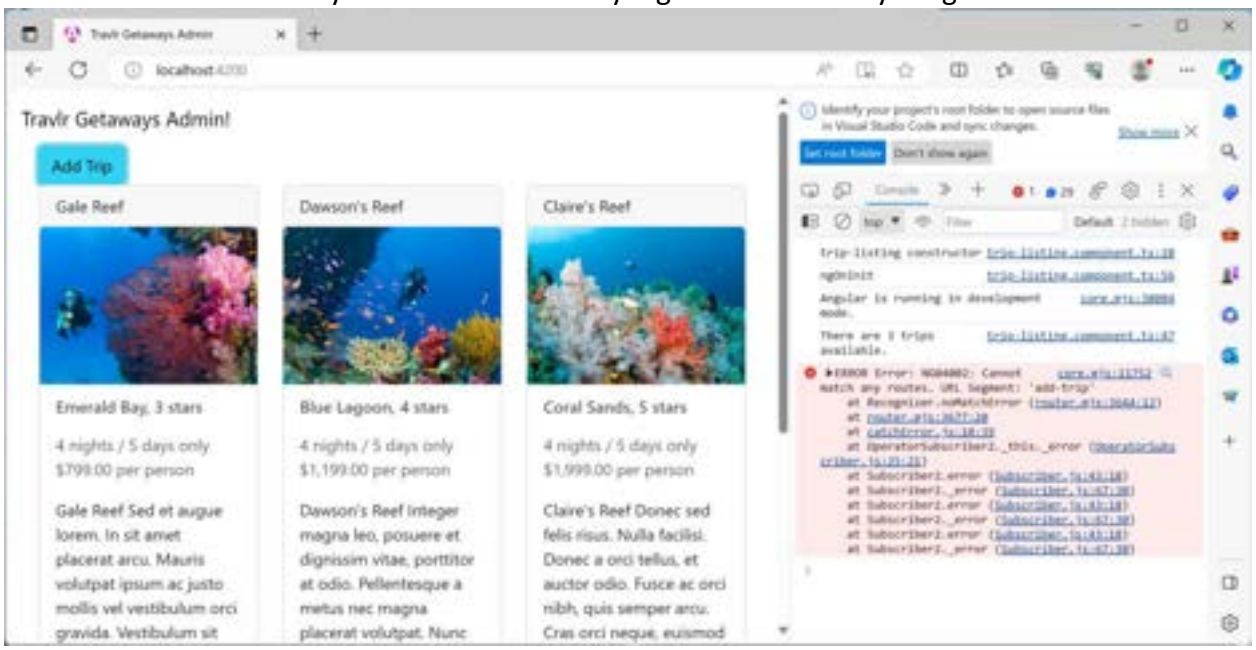
With this change, you can see that the card will now automatically scale with the size of the viewport:



And when we expand the viewport back to more than 768 pixels in width:

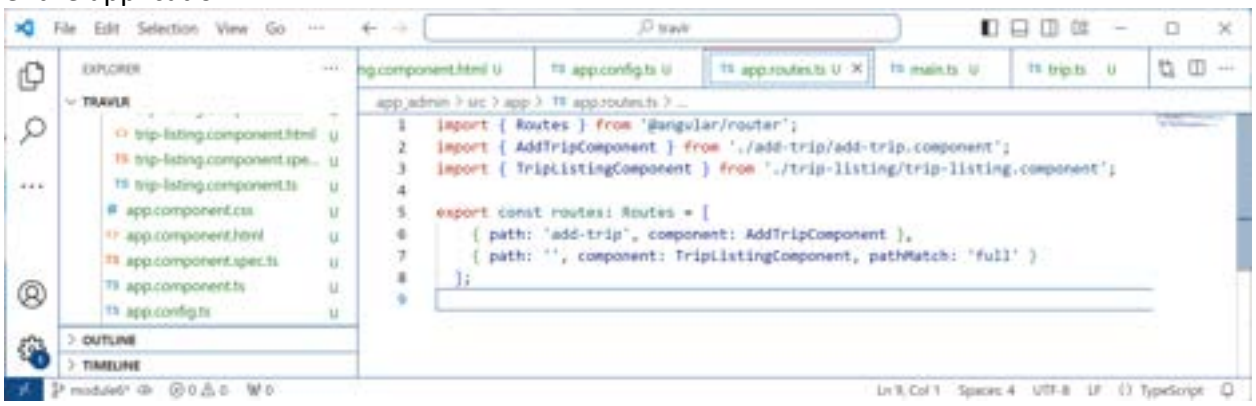


5. If we were to test our application now by pressing the *add-trip* button we would generate an error because we do not yet have the necessary logic for it to do anything:



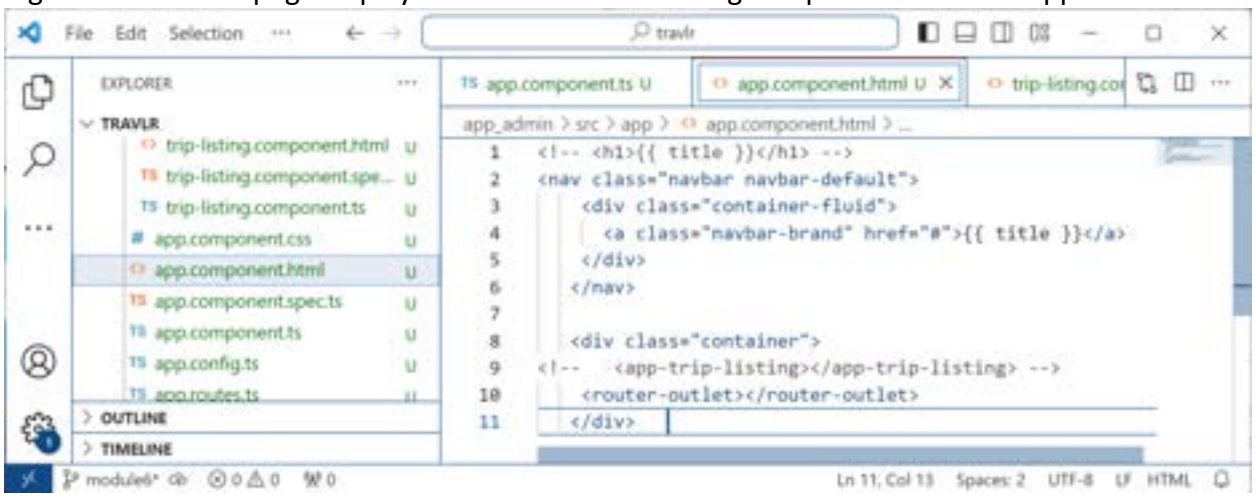
Which leads us into the next step for the application which is to wire-up the routing capability for our application. The built-in routing capability in Angular is very powerful, and will allow us to update our application to become more dynamic with its display capabilities.

6. It is common for Single-Page Applications (SPA) and web-applications in general to have many URLs for their various pages/features. This is where Angular's routing service allows us to simplify how these are addressed in our SPA. We are following a best-practice by isolating the routing code – we will inject this into our ***app.routes.ts*** file to enable routing across our entire application.

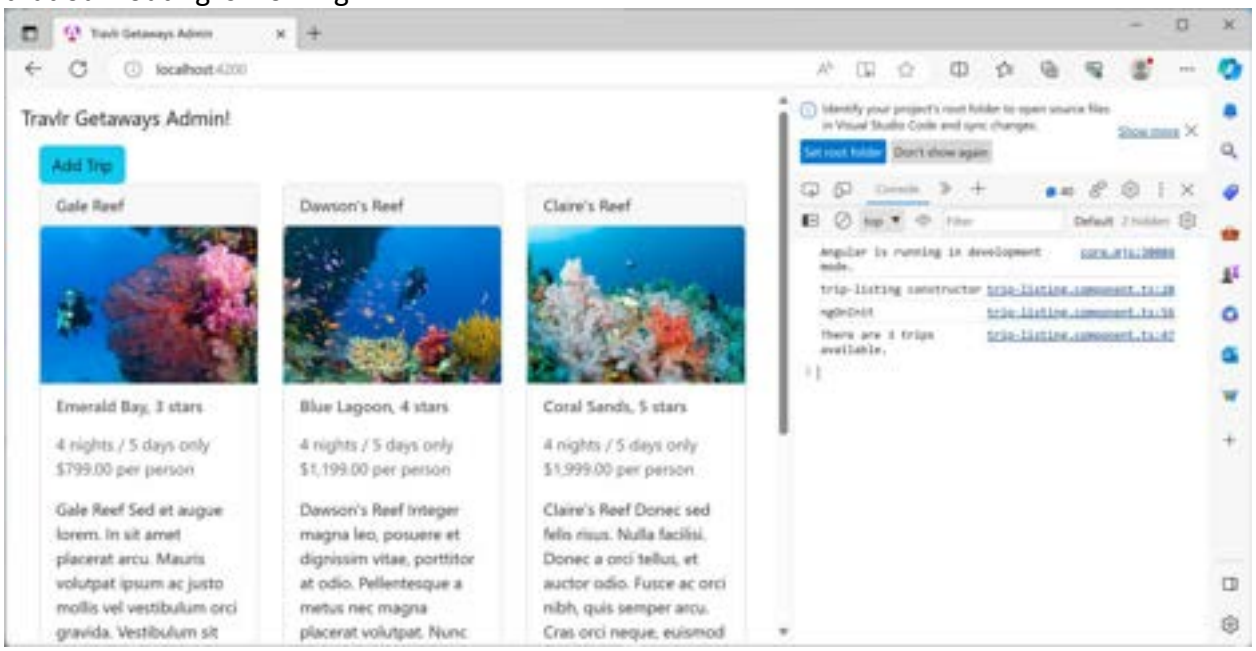


As you can see we created two paths in our definition – this is so that we can activate each of these components separately with our application. Please Note: You must also import each component for which you are providing a route.

7. We are going to make a small change to our ***app-component.html*** file to enable the routing logic to handle the page displays rather than hard-coding components into the application.



Here we have commented out our original code for the ***trip-listing*** component and replaced it with Angular's *router-outlet* tag. This allows our routing configuration to control when to display the components in our application. Examining the resulting rendered page shows us that our routing is working:



8. Now that we have routing configured properly, we need to work on our add-trip component. So we will be editing the ***add-trip.component.ts*** file. The contents for this file should look like this:

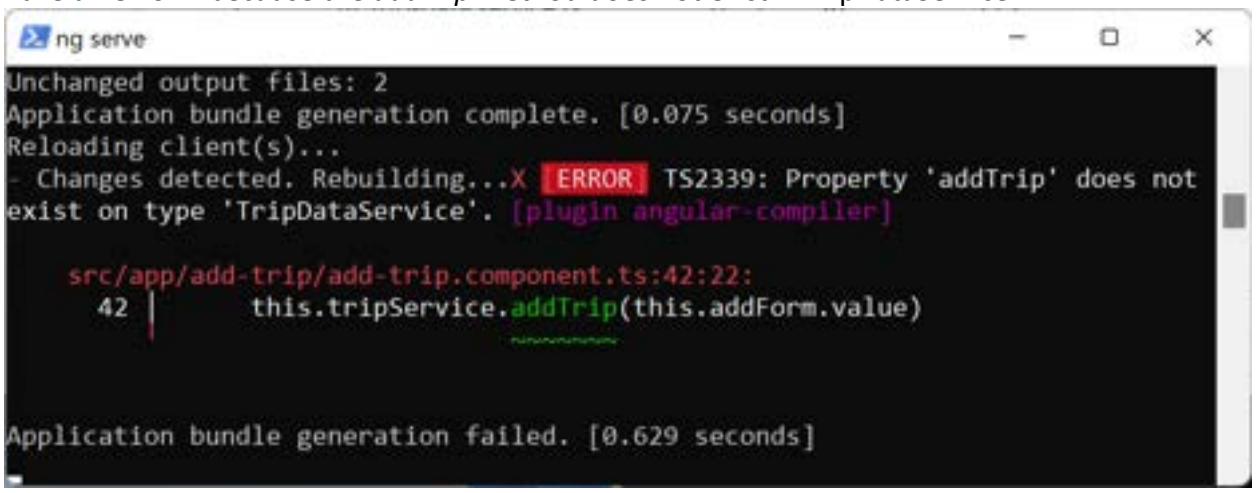
```
import { Component, OnInit } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormBuilder, FormGroup, Validators } from "@angular/forms";
```

```
import { Router } from "@angular/router";
import { TripDataService } from '../services/trip-data.service';

@Component({
  selector: 'app-add-trip',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './add-trip.component.html',
  styleUrl: './add-trip.component.css'
})

export class AddTripComponent implements OnInit {
 addForm!: FormGroup;
 submitted = false;

 constructor(
   private formBuilder: FormBuilder,
   private router: Router,
   private tripService: TripDataService
 ) { }

 ngOnInit() {
   this.addForm = this.formBuilder.group({
     _id: [],
     code: ['', Validators.required],
     name: ['', Validators.required],
     length: ['', Validators.required],
     start: ['', Validators.required],
     resort: ['', Validators.required],
     perPerson: ['', Validators.required],
     image: ['', Validators.required],
     description: ['', Validators.required],
   })
 }

 public onSubmit() {
   this.submitted = true;
   if(this.addForm.valid){
     this.tripService.addTrip(this.addForm.value)
     .subscribe( {
      next: (data: any) => {
       console.log(data);
       this.router.navigate(['']);
     },
     error: (error: any) => {
```

```
        console.log('Error: ' + error);
      }));
    }
  }
  // get the form short name to access the form fields
  get f() { return this.addForm.controls; }
}
```

This code initializes routing, and sets up both our form, and the appropriate data structure (the formBuilder.group) that lays out how we will access data that the user will enter into our form. The @Angular/forms module is what handles the data-binding between the form, and our variables.

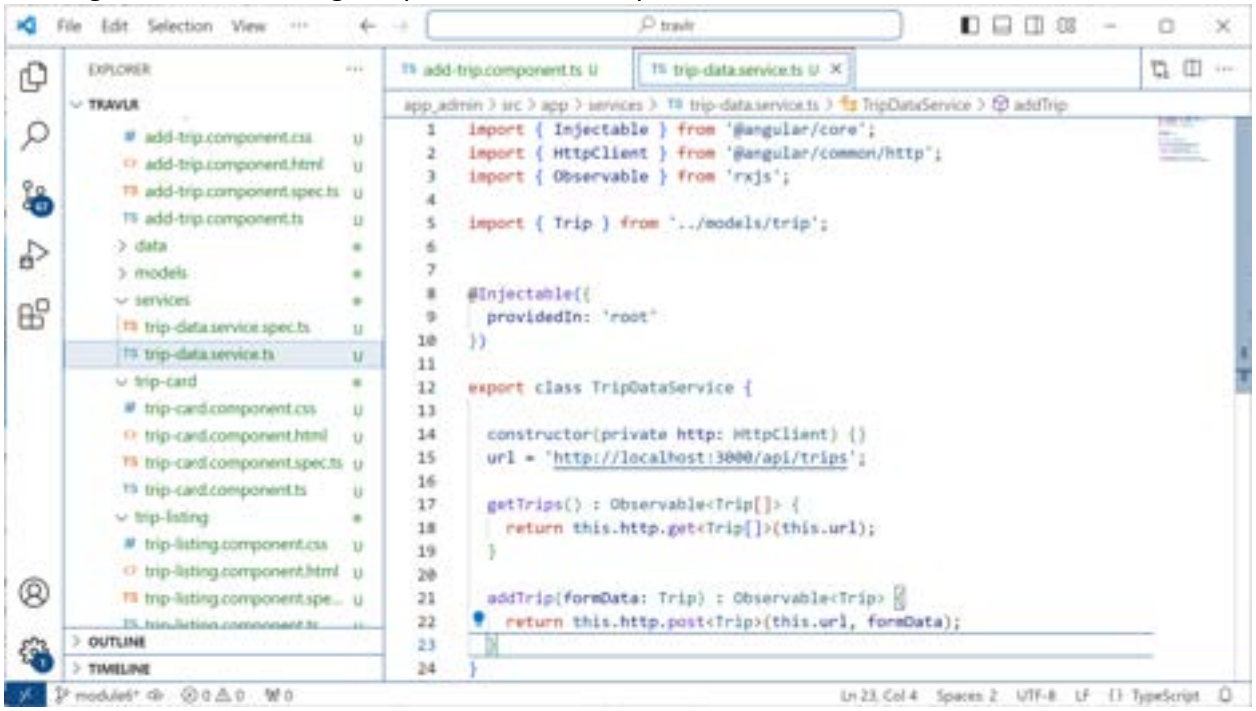
The onSubmit() method that we create is declared as public so that it can be called on the button-press, and it utilizes the TripDataService that to pass the data back to the Express application. You can see the similarity in how we handle the subscription to the TripDataService method – it is the same methodology that we use in the ***trip-listing*** component when we get the data from the service. You should note that at this time, we have an error – because the *addTrip* method does not exist in TripDataService:



Which will be what we address next.

9. We need to edit the ***services/trip-data.service.ts*** file to add a metehod for addTrip. We will also take this opportunity to refactor the url variable out of the getTrips method, because it

is being used in both the getTrips, and the addTrip methods.



This will satisfy the binding for the ***add-trip*** component but it will not get us anywhere because we still need to add the HTML code to generate the form, and we need to build the back-end API method to accept the posted form data.

10. Now that we have the add-trip component logic built, we need to create the form that will allow the user to enter the data we are trying to collect. This belongs in the ***add-trip.component.html*** file and is fairly straight-forward. Please Note: The control names for each of the form fields must match exactly with the individual field names from the formGroup defined in your ***add-trip.component.ts*** file. This file should contain the following:

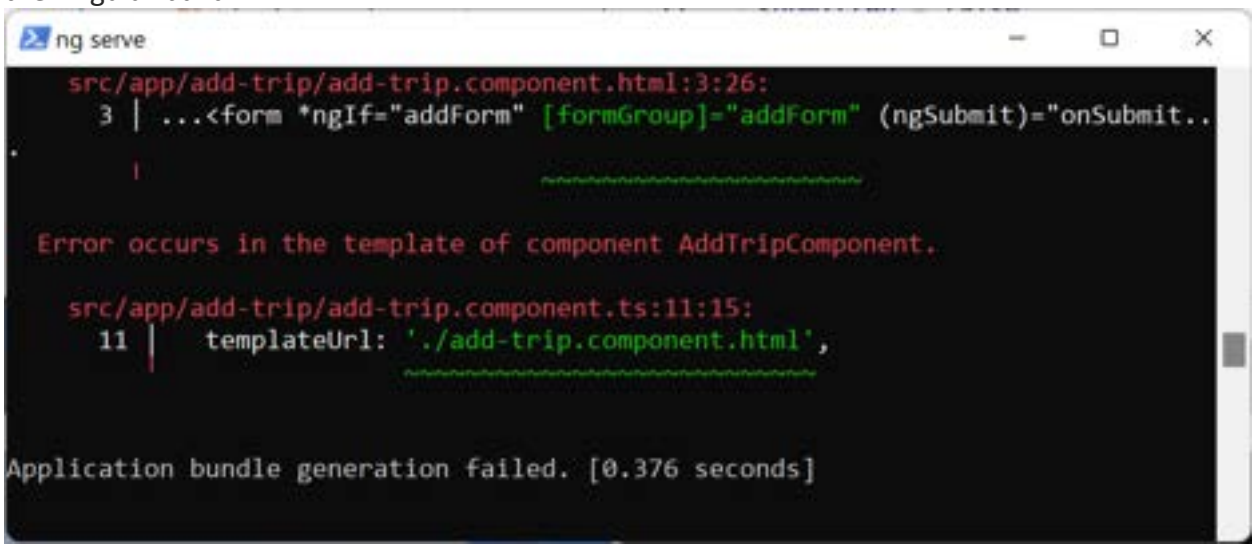
```
<div class="col-md-4">
    <h2 class="text-center">Add Trip</h2>
    <form *ngIf="addForm" [formGroup]="addForm"
(ngSubmit)="onSubmit()">

      <div class="form-group">
        <label>Code:</label>
        <input type="text" formControlName="code"
          placeholder="Code" class="form-control"
          [ngClass]="{ 'is-invalid': submitted && f['code'].errors
}">
        <div *ngIf="submitted && f['code'].errors">
          <div *ngIf="f['code'].errors?.['required']">
             Trip Code is required
```

```
        </div>
      </div>
    </div>

    <div class="form-group">
      <label>Name:</label>
      <input type="text" formControlName="name"
        placeholder="Name" class="form-control"
        [ngClass]="{ 'is-invalid': submitted && f['name'].errors
}">
      <div *ngIf="submitted && f['name'].errors">
        <div *ngIf="f['name'].errors?.['required']">
           Name is required
         </div>
      </div>
    </div>

    <div class="form-group">
      <label>Length:</label>
      <input type="text" formControlName="length"
        placeholder="Name" class="form-control"
        [ngClass]="{ 'is-invalid': submitted && f['length'].errors
}">
      <div *ngIf="submitted && f['length'].errors">
        <div *ngIf="f['length'].errors?.['required']">
           Length is required
         </div>
      </div>
    </div>

    <div class="form-group">
      <label>Start:</label>
      <input type="date" formControlName="start"
        placeholder="Start" class="form-control"
        [ngClass]="{ 'is-invalid': submitted && f['start'].errors
}">
      <div *ngIf="submitted && f['start'].errors">
        <div *ngIf="f['start'].errors?.['required']">
           Date is required
         </div>
      </div>
    </div>

    <div class="form-group">
      <label>Resort:</label>
```

```
        <input type="text" formControlName="resort"
          placeholder="Resort" class="form-control"
          [ngClass]="{ 'is-invalid': submitted && f['resort'].errors
}">
        <div *ngIf="submitted && f['resort'].errors">
          <div *ngIf="f['resort'].errors?.['required']">
             Resort is required
           </div>
        </div>
      </div>

      <div class="form-group">
        <label>Per Person:</label>
        <input type="text" formControlName="perPerson"
          placeholder="Perperson" class="form-control"
          [ngClass]="{ 'is-invalid': submitted &&
f['perPerson'].errors }">
        <div *ngIf="submitted && f['perPerson'].errors">
          <div *ngIf="f['perPerson'].errors?.['required']">
             Per Person is required
           </div>
        </div>
      </div>

      <div class="form-group">
        <label>Image Name:</label>
        <input type="text" formControlName="image"
          placeholder="Image" class="form-control"
          [ngClass]="{ 'is-invalid': submitted && f['image'].errors
}">
        <div *ngIf="submitted && f['image'].errors">
          <div *ngIf="f['image'].errors?.['required']">
             Image Name is required
           </div>
        </div>
      </div>

      <div class="form-group">
        <label>Description:</label>
        <input type="text" formControlName="description"
          placeholder="Description" class="form-control"
          [ngClass]="{ 'is-invalid': submitted &&
f['description'].errors }">
        <div *ngIf="submitted && f['description'].errors">
          <div *ngIf="f['description'].errors?.['required']">
```

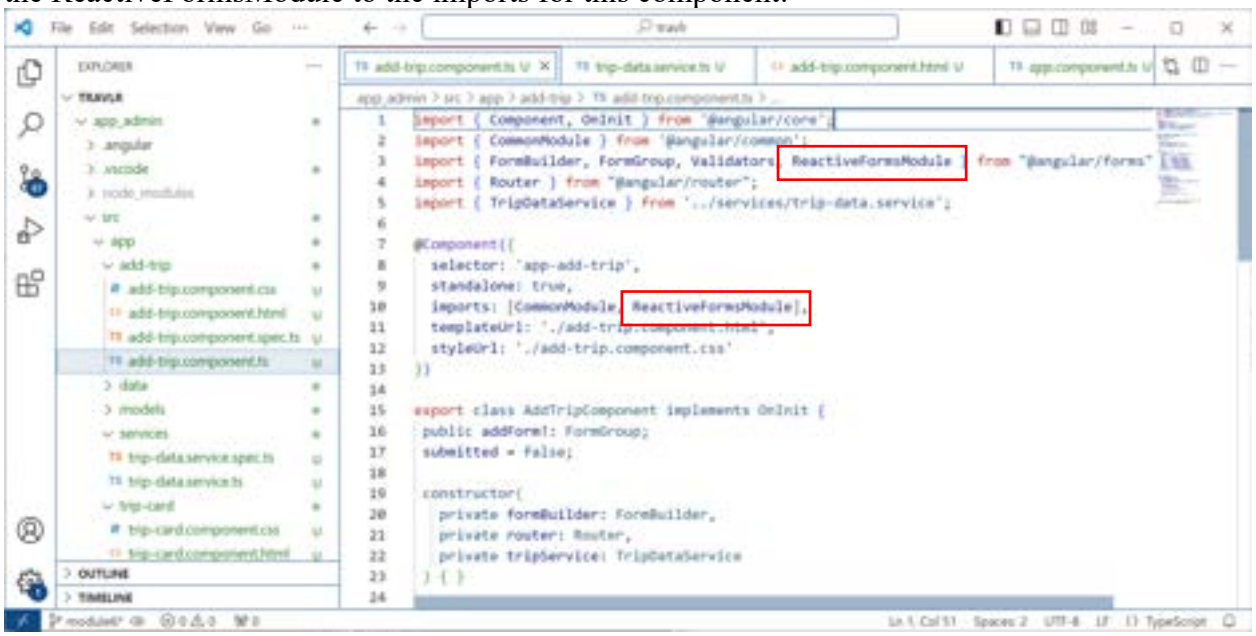
```
            Description is required
          </div>
        </div>
      </div>

      <button type="submit" class="btn btn-info">Save</button>
    </form>
  </div>
```

When you save this file, we find one more item that we must address – there is an error in the Angular build:

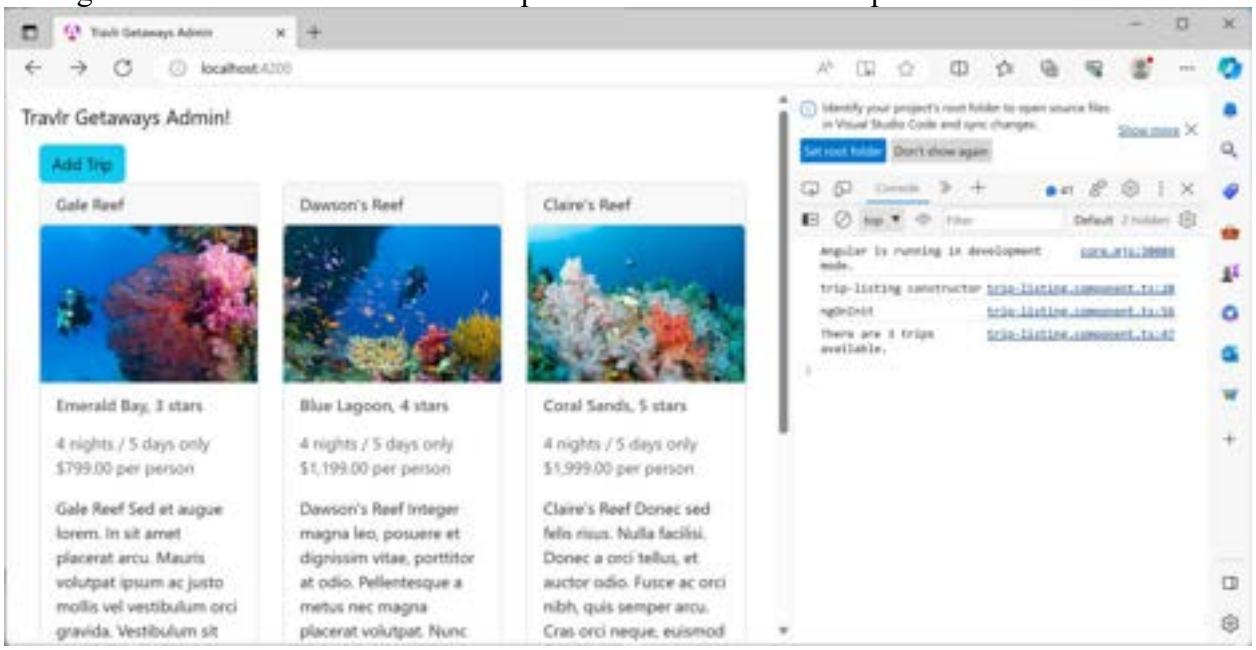


When we double-check our add-trip.component.ts file, we find that we have the appropriate definitions for addFrom, and we imported FormGroup, but it still isn't working. We need to make two more changes. We need to make addForm a public variable, and we need to add

the ReactiveFormsModule to the imports for this component:



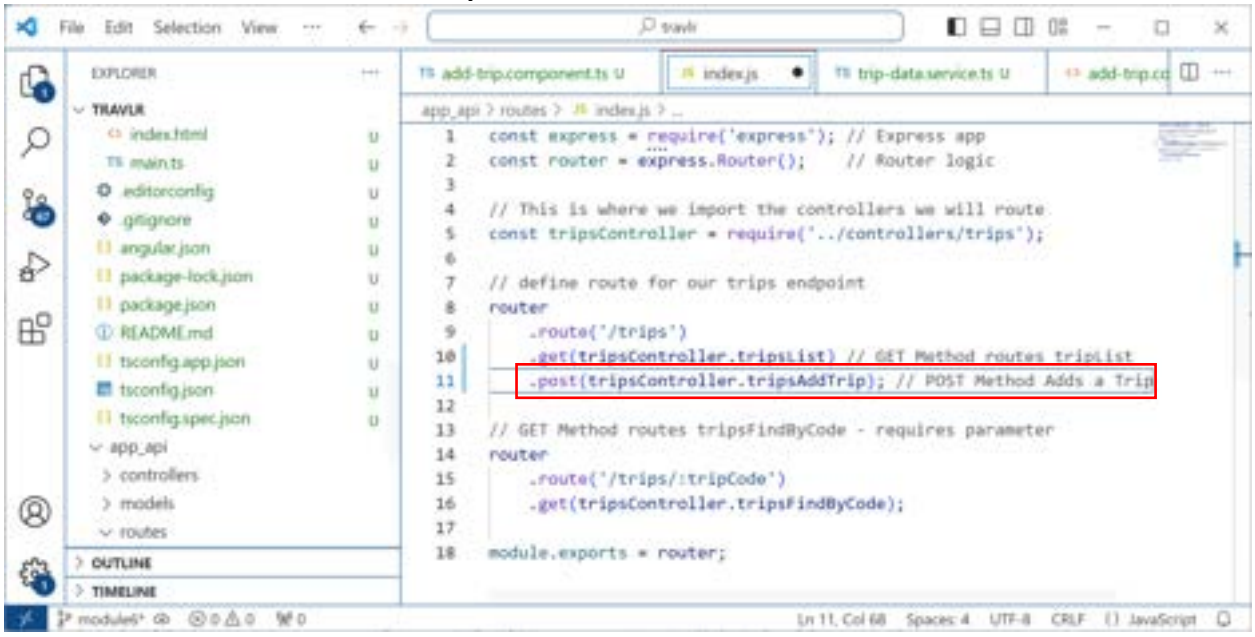
Saving the file will result in a clean compile and no errors in the inspection window:



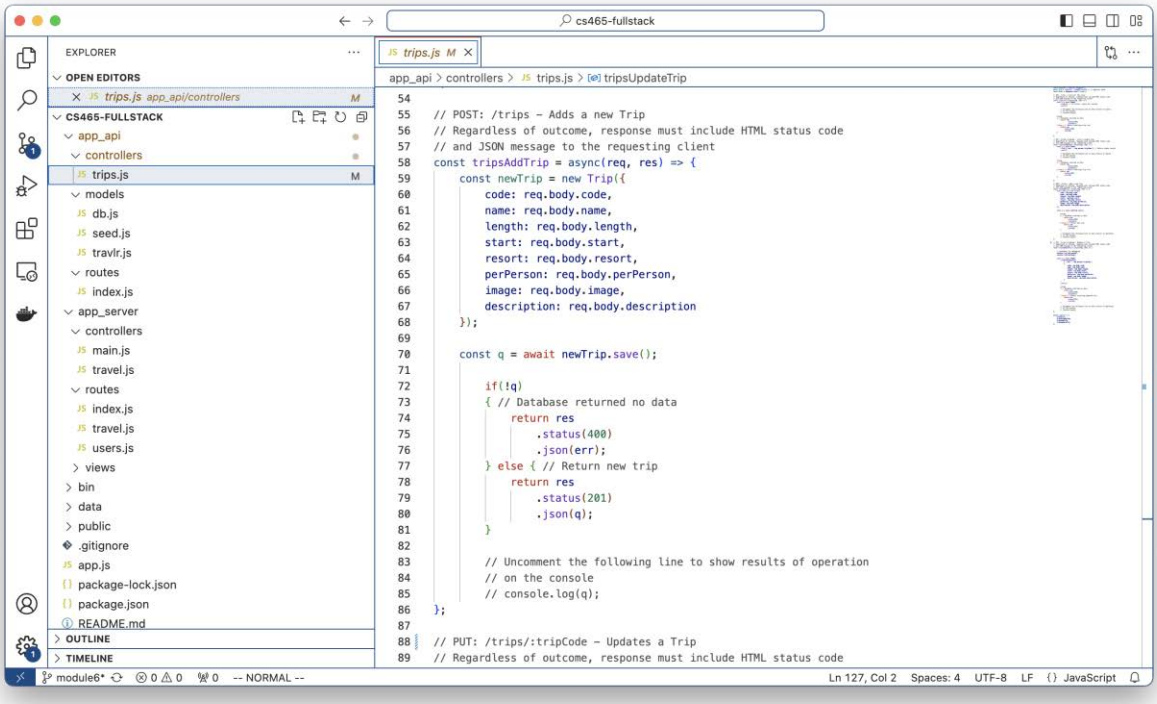
However, selecting the Add-Trip button and completing the form will generate a 404 error because the backend does not know how to treat the post command.

11. Now we have to circle back around to the backend. We will edit the ***app_api/routes/index.js*** file to add a *post* option to our endpoint and designate a new

method we need to create in our ***tripsController***.

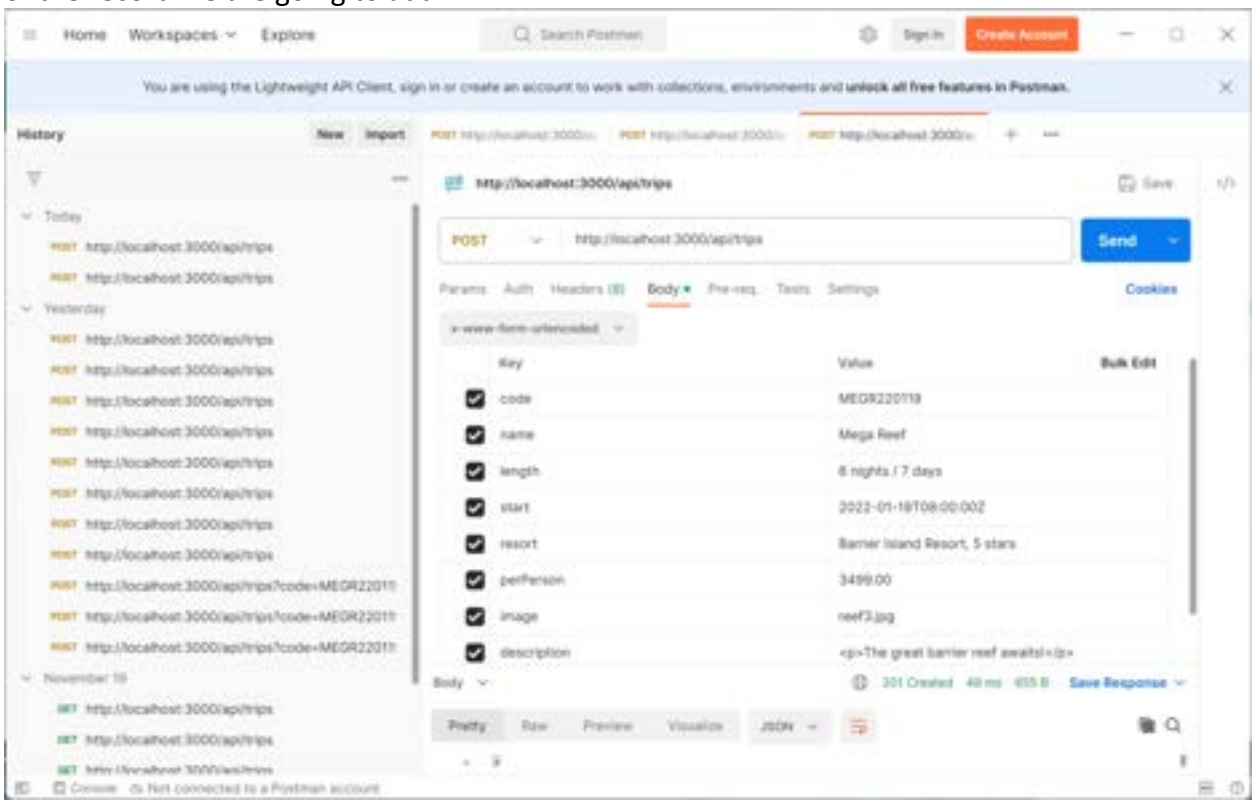


12. Now we need to modify the tripsController to handle the data that we are passing back within the request body.

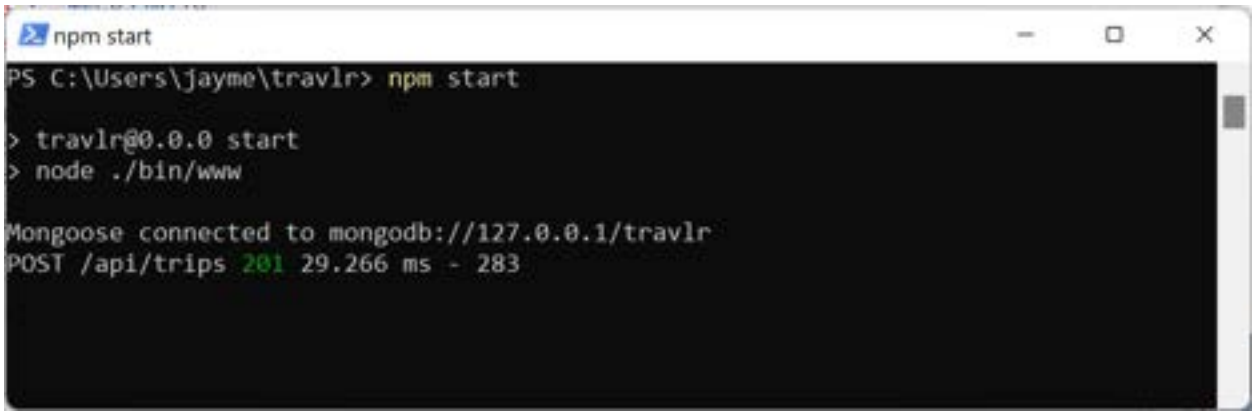


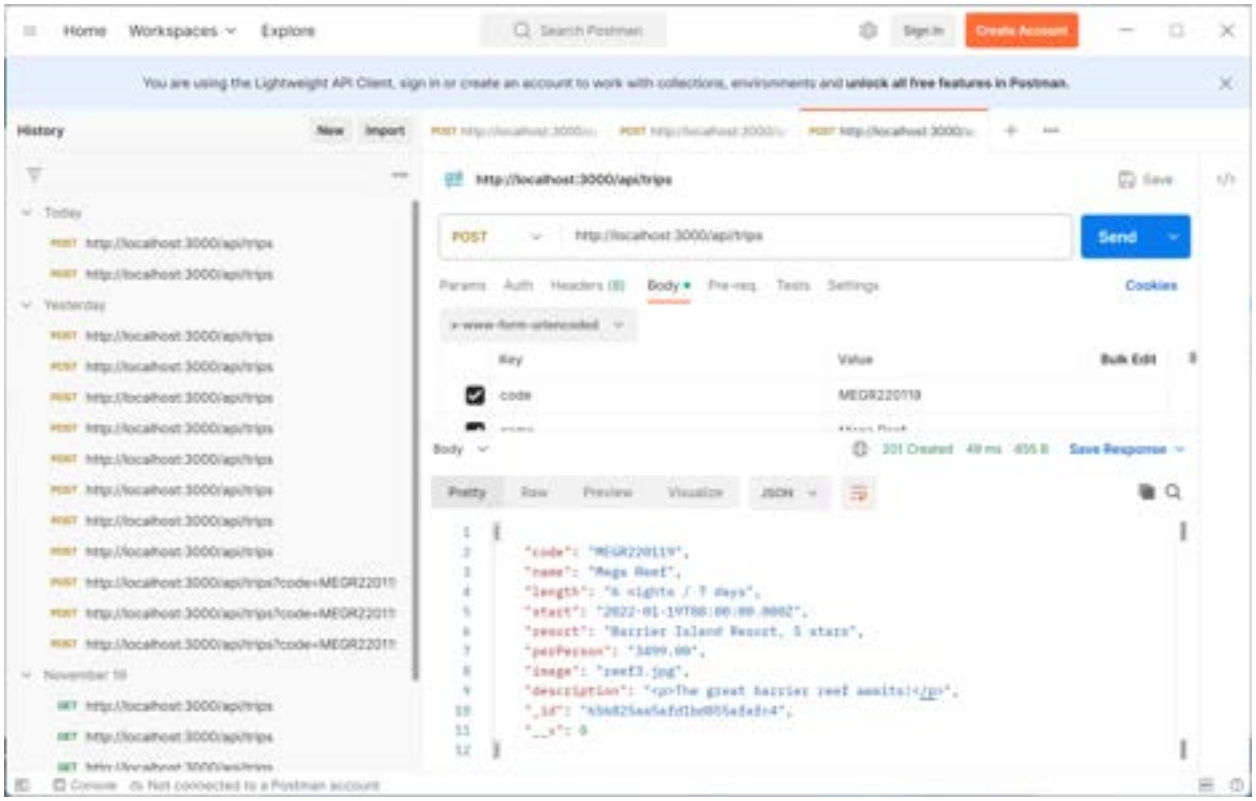
Make sure you add your new method to the modules.exports list!

13. We will start by testing our endpoint with Postman. Set the query type to post, and the api endpoint to http://localhost:3000/api/trips. Select the 'Body' tab, and the select 'x-www-form-urlencoded' as the data type. Add the appropriate key-value pairs for each of the fields of the record we are going to add.

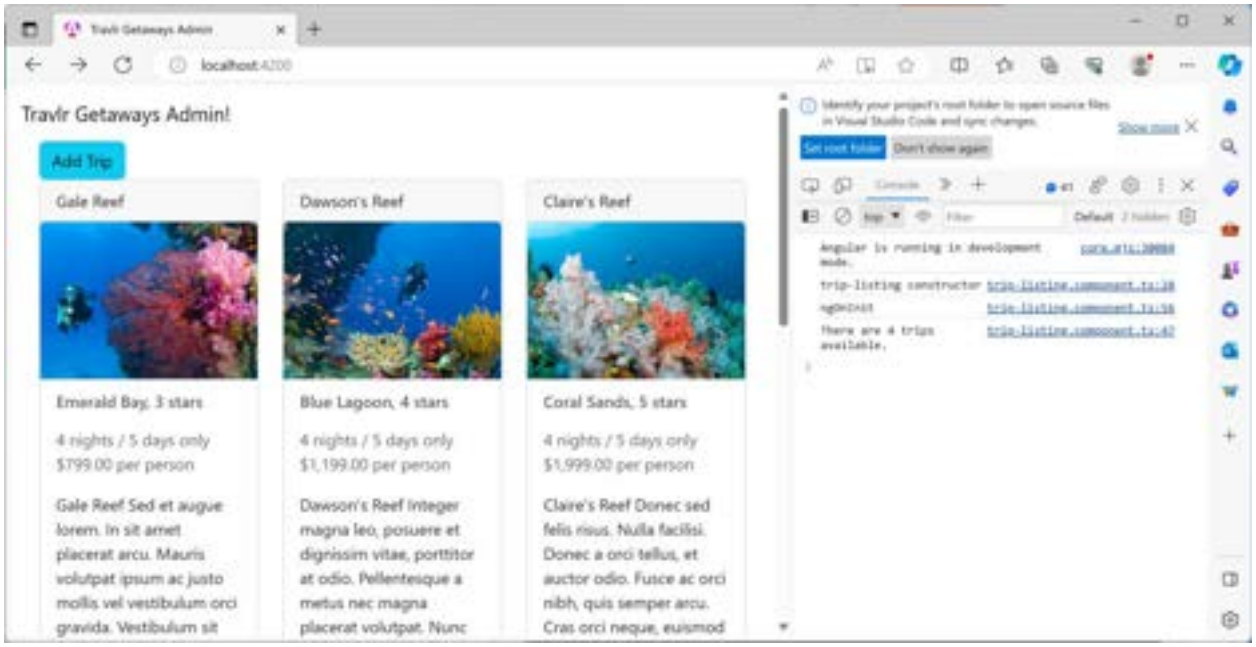


Once you have verified that you have added each field (and that they are spelled correctly) you can press the 'send' button to bounce your query against the backend API endpoint. If everything is successful, the call will return a JSON object with the record that you just entered into the MongoDB, and you will see the result of this in the PowerShell Window where you are running your Express server:





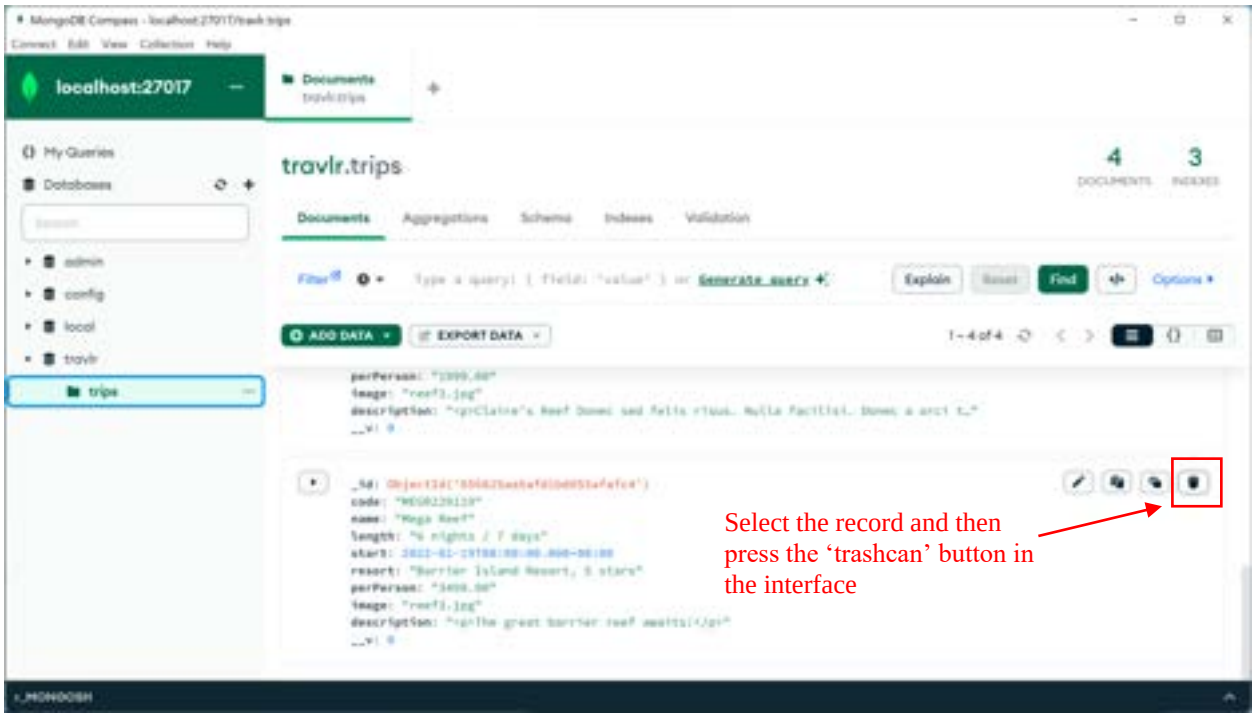
You should also be able to look at your front-end and see that there is now a fourth record available:

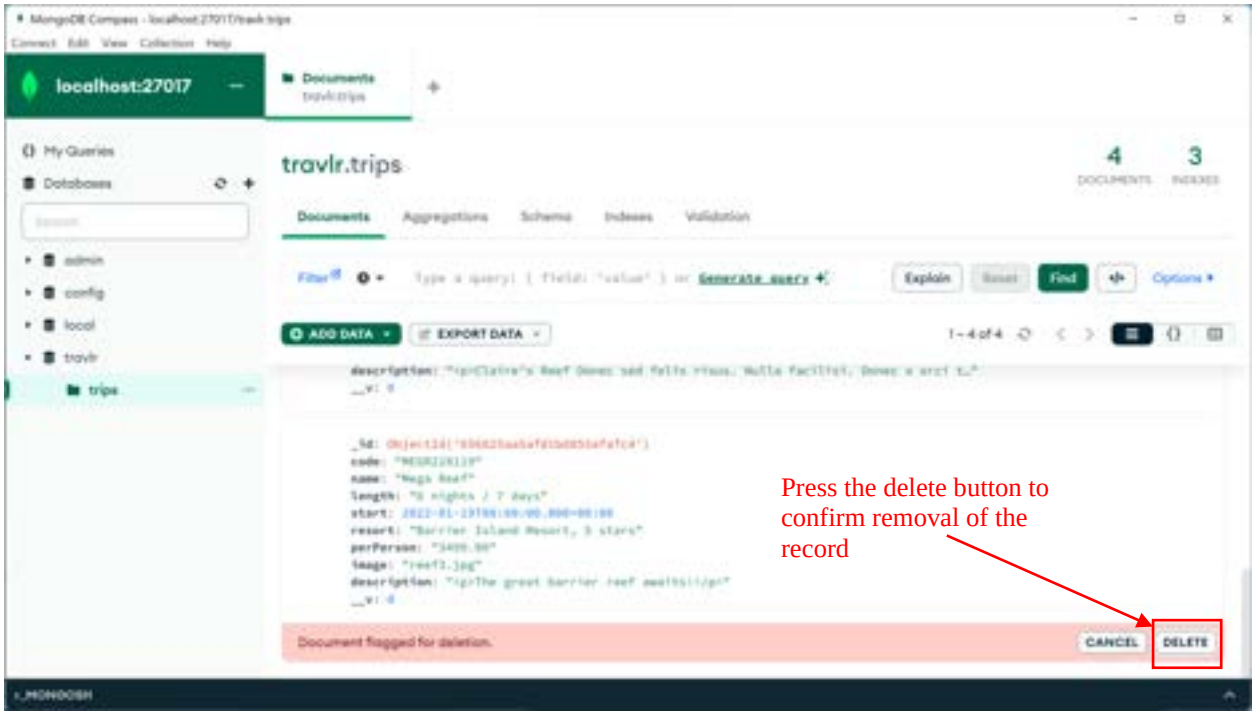


14. Now that we have shown that our test works directly with the back-end, we need to test to make certain that it also works when the request is submitted from our Angular JS front-end

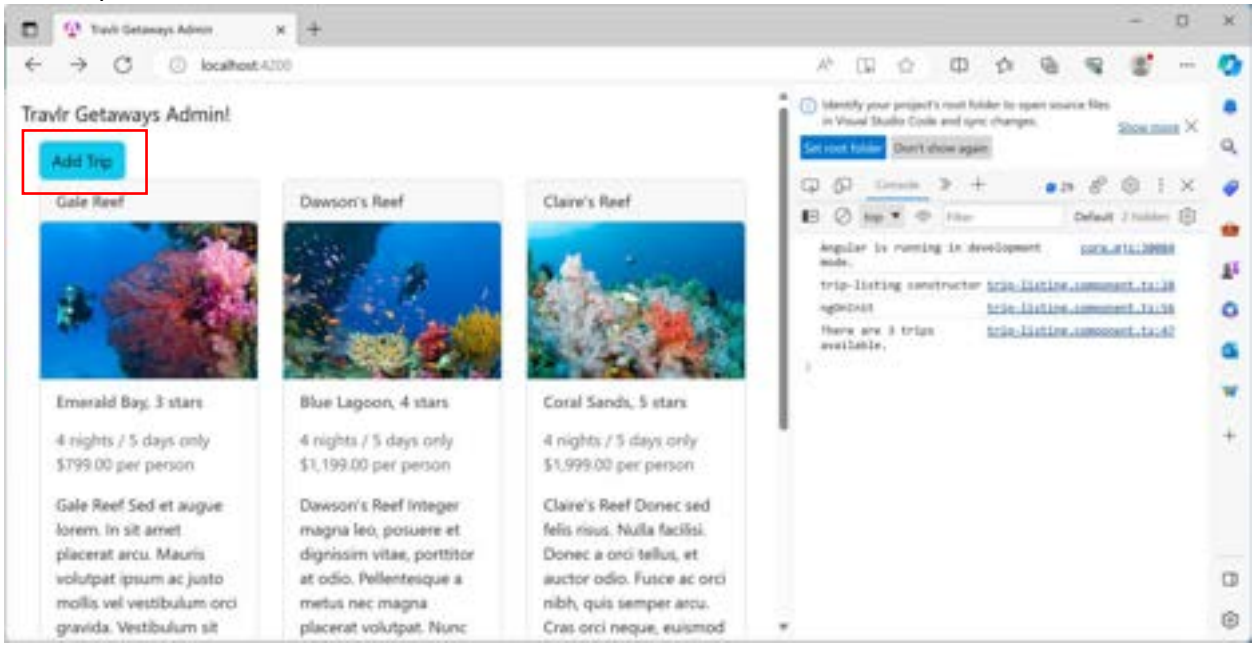
application. We have two choices here. We can remove the record we just added from the Mongo Database, or we can add a duplicate record. I am going to go through the process of removing the record from Mongo so that I can definitely determine the success of my test with the Angular application.

In MongoDB Compass (or DBeaver), find the record that you just added via Postman, and remove it from the database:

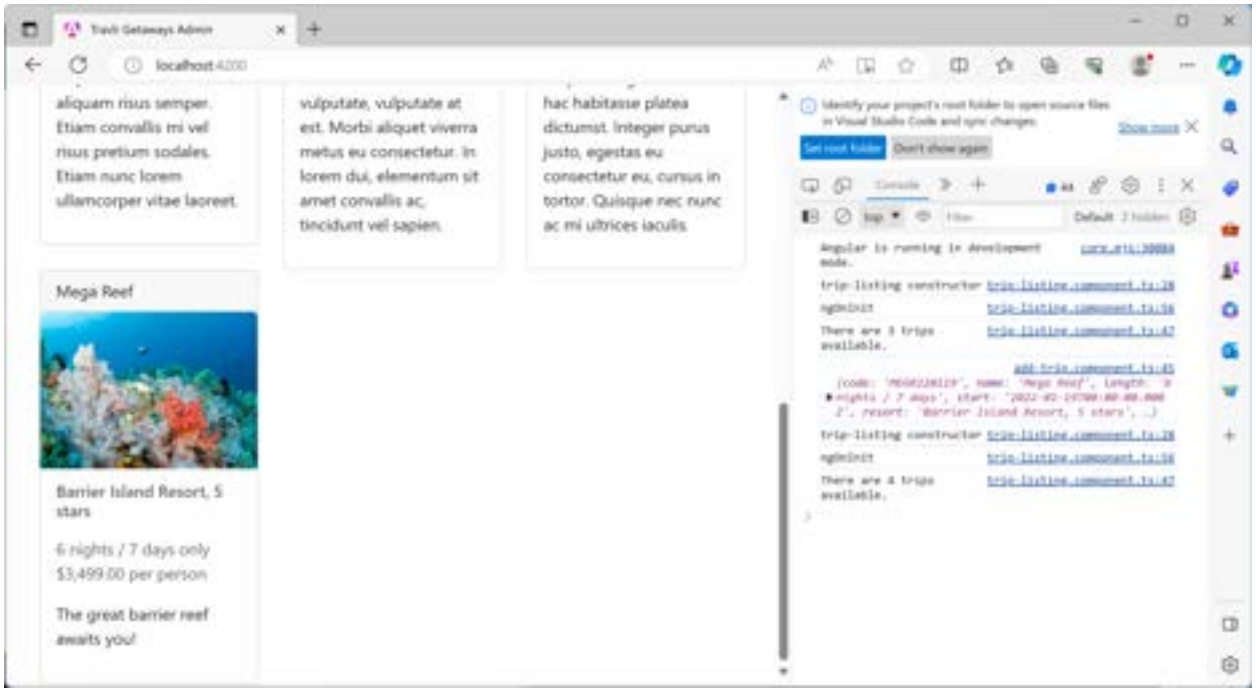




15. Now that we are back to having only three records in the database, let's go back to our Angular application and test our data entry form. From our Angular application, press the Add-Trip Button:



Fill in the form and press the 'Save' button. This should submit your record to the Mongo DB through the backend Express API. You should see results that match what you saw when you used Postman to test the API.



You can clearly see in the inspector that there are now 4 trips available instead of three, and if you scroll down you can see that trip displayed in your application.

Edit Trips

Now that we have shown that we could successfully add the necessary code and controls to add a trip to our application, it should be very straightforward to add the code to Edit (update) an existing trip. This should help pull together all the concepts that we have learned so far with an exercise that should round-out the initial capabilities for the application.

1. The action of updating a record in our database utilizes a new HTTP verb – PUT. GET was used to perform the read operation from the database; POST was used to add a record to the database; now we will use PUT to update a record in the database. We will start this process by creating a new method in our ***app_api/controllers/trips.js*** file and it will be similar to the method we just created for adding a new trip.

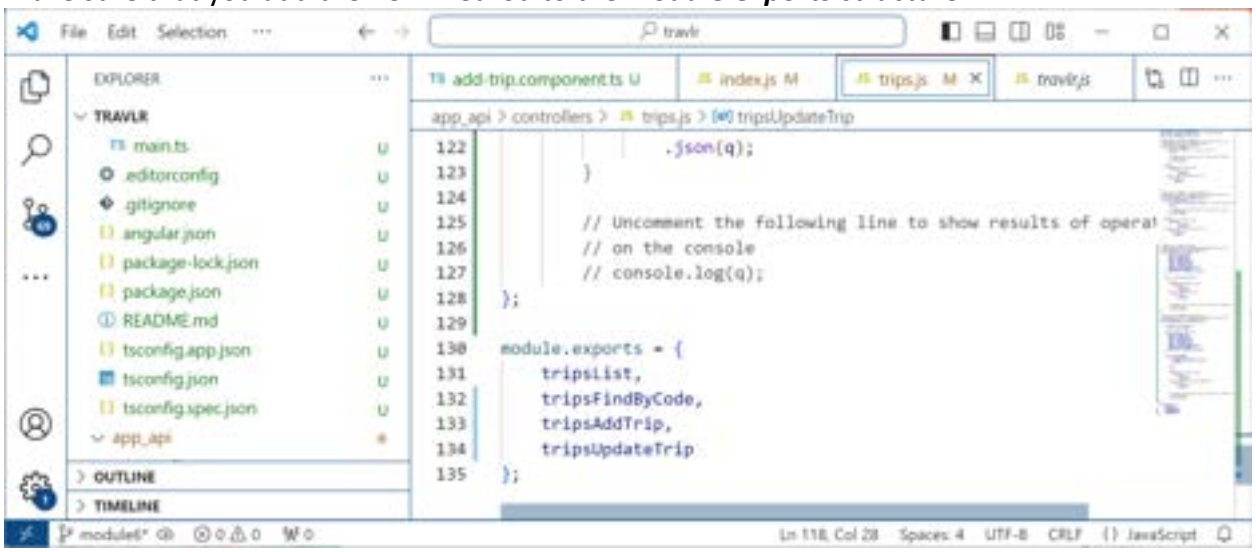
   We will add the following code to the file:

```
// PUT: /trips/:tripCode - Adds a new Trip
// Regardless of outcome, response must include HTML status
code
// and JSON message to the requesting client
const tripsUpdateTrip = async(req, res) => {

    // Uncomment for debugging
    console.log(req.params);
    console.log(req.body);

    const q = await Model
        .findOneAndUpdate(
            { 'code' : req.params.tripCode },
            {
                code: req.body.code,
                name: req.body.name,
                length: req.body.length,
                start: req.body.start,
                resort: req.body.resort,
                perPerson: req.body.perPerson,
                image: req.body.image,
                description: req.body.description
            }
        )
        .exec();

        if(!q)
        { // Database returned no data
            return res
                .status(400)
                .json(err);
```

```
        } else { // Return resulting updated trip
            return res
                .status(201)
                .json(q);
        }

        // Uncomment the following line to show results of
operation
        // on the console
        // console.log(q);
};
```

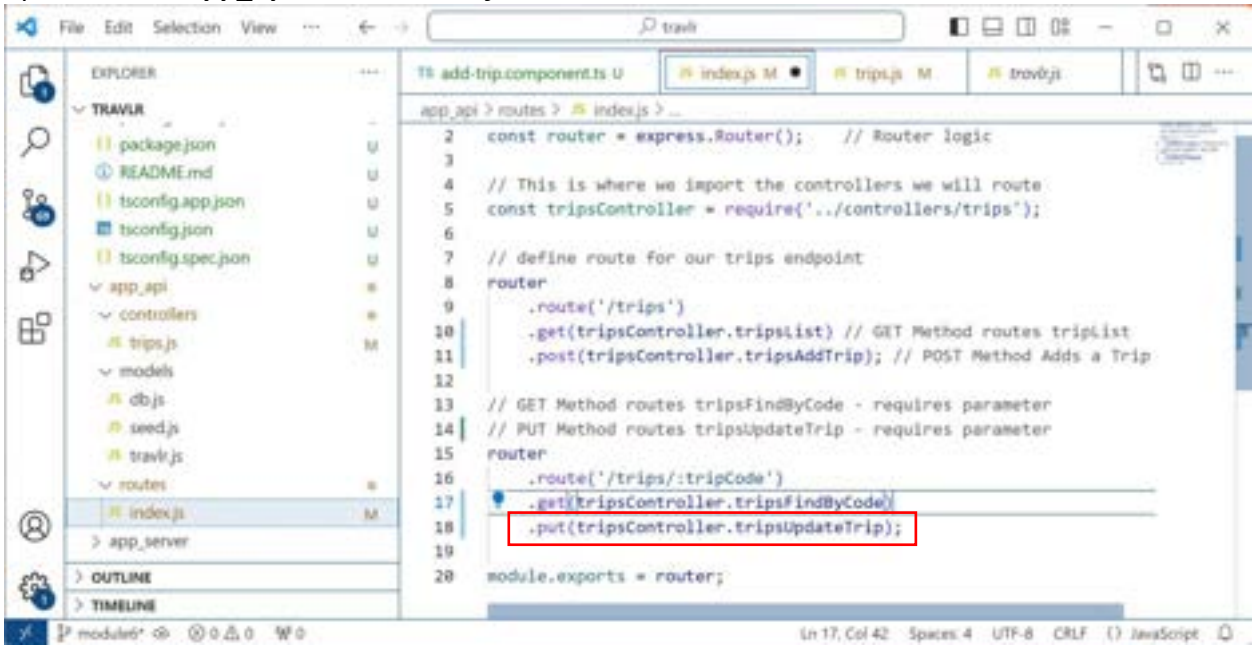
The biggest difference between the update method and the add method is that we are using the findOneAndUpdate method from Mongoose. This directs the database to locate the specified record (which we are selecting with a parameter for our API endpoint) and update the document based on the fields that are passed in. In this case, we are passing in a complete document, but you could also pass in only the field that needs to be updated.

Make sure that you add the new method to the ***module.exports*** structure:

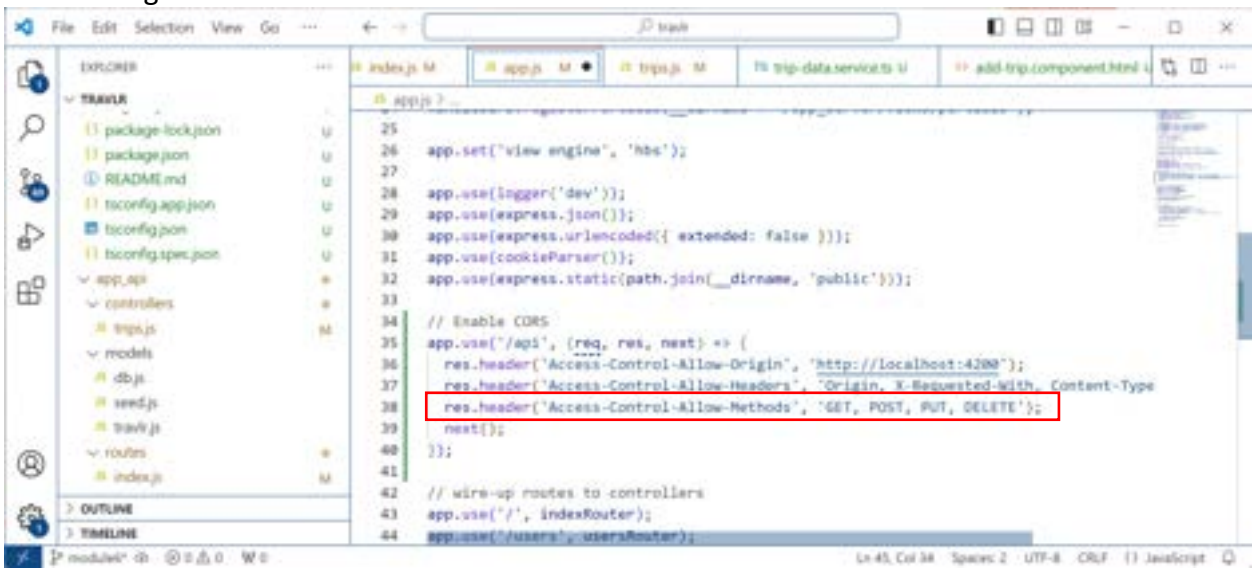


2. Now that we have our new method constructed, we need to add a route so that the Express application can find the method to call when it recieves a PUT request. This requires an

update to the ***app_api/routes/index.js*** file.



3. That will add a route for our Express backend for the HTTP PUT verb. However, our application will not work quite yet. When we enabled CORS in our application to support the APIs, we didn't specify the HTTP configuration we would utilize, so that only enables GET and POST by default. As a consequence, we will need to edit our app.js file and add one more configuration line:

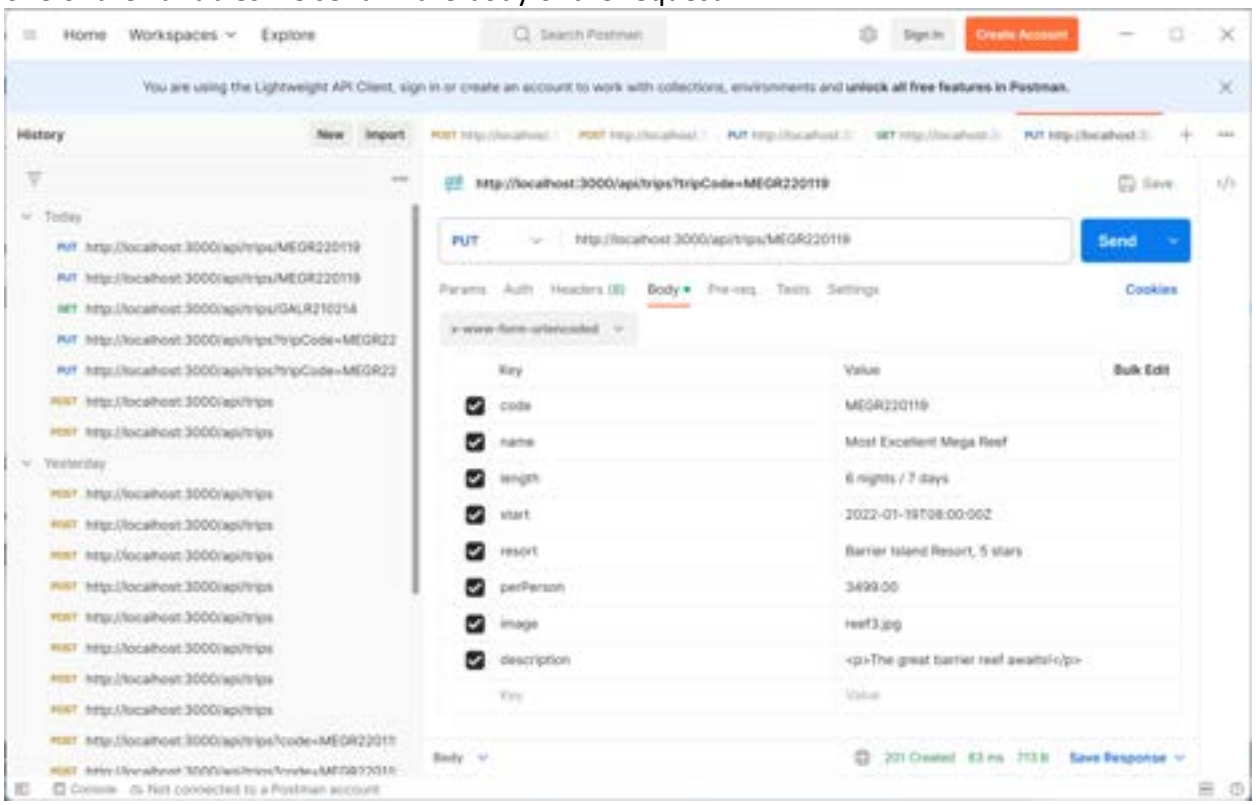


We added one line to our definition of our /api endpoint to enable access to the additional HTTP verbs.

```
res.header('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE');
```

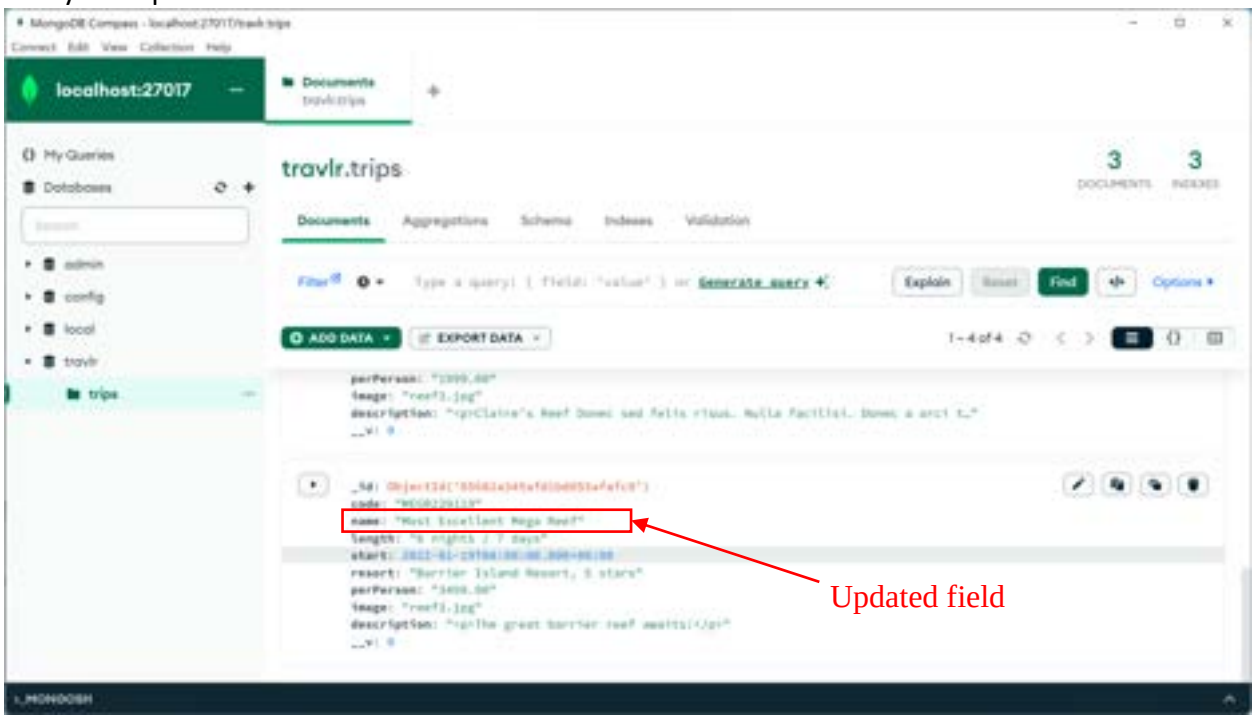
Once you have completed this step, please restart your Express backend server process.

4. Before we proceed to make the changes to enable this for the Angular front end, we will take the opportunity to test with Postman. We will make a small change to the query that we previously sent. We will add the *tripcode* parameter to the end of our URL and set the value to the code we just added in the previous section. Then we will change the value of one of the variables we send in the body of the request.



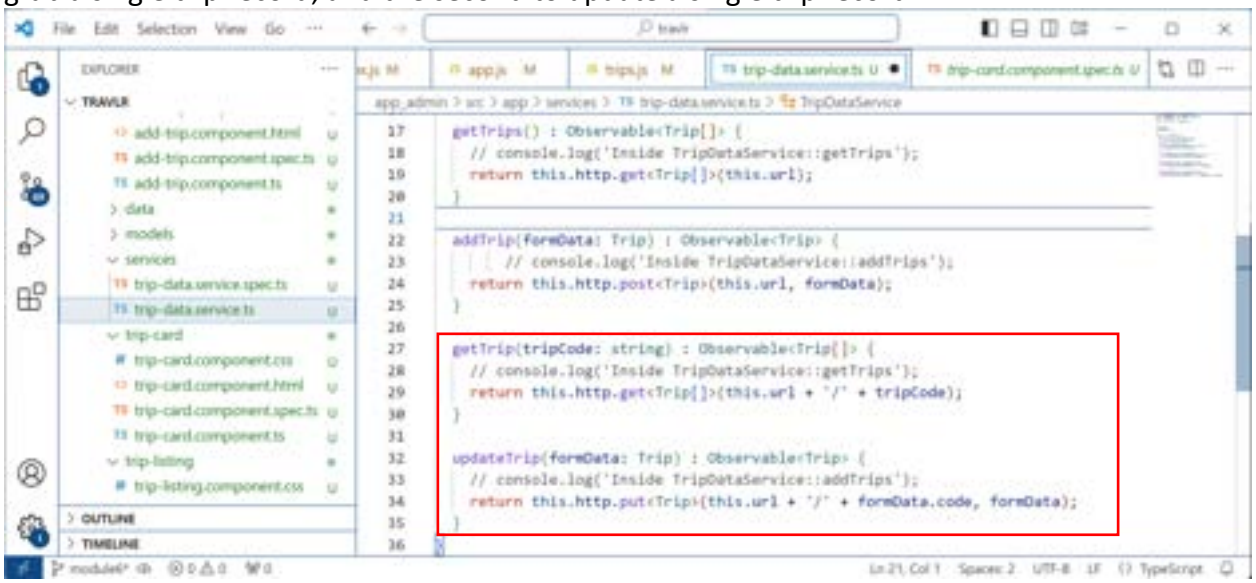
Once you have prepared the query, press the 'send' button and test your API call. You should get a '201 Created' message at the bottom of the window, and if you expand that section you will see the original record that you just updated.

Go back over to Mongo Compass or DBeaver and view the record in the Mongo Database to verify the update:



You can clearly see that our update was successful and our API endpoint is functional!

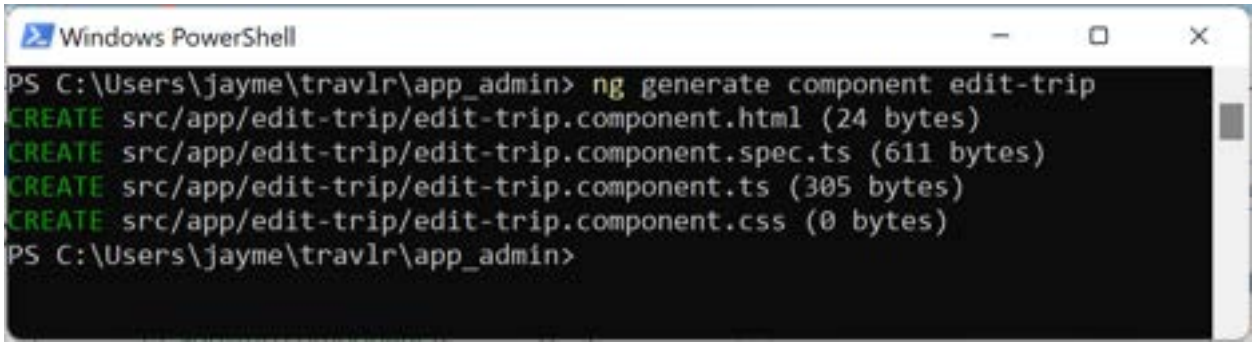
5. Now that we have the backend for the update functionality built, we need to go back and make the adjustment for the Angular application to provide a front-end interface to support updates. We will start by adding two additional methods to our ***TripDataService*** the first to grab a single trip record, and the second to update a single trip record.



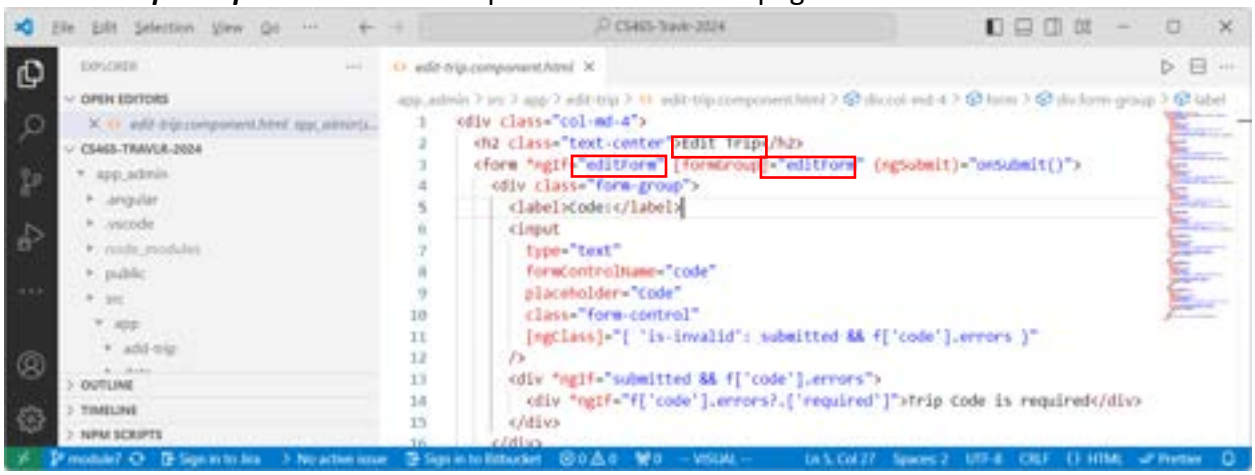
Please Note: For both of these methods we had to make sure we extended the URL to support the addition of the parameter for the tripCode that will specify an individual record.

6. For the front-end we are going to need another component to manage the editing process, so we will create a new component called ***edit-trip***.

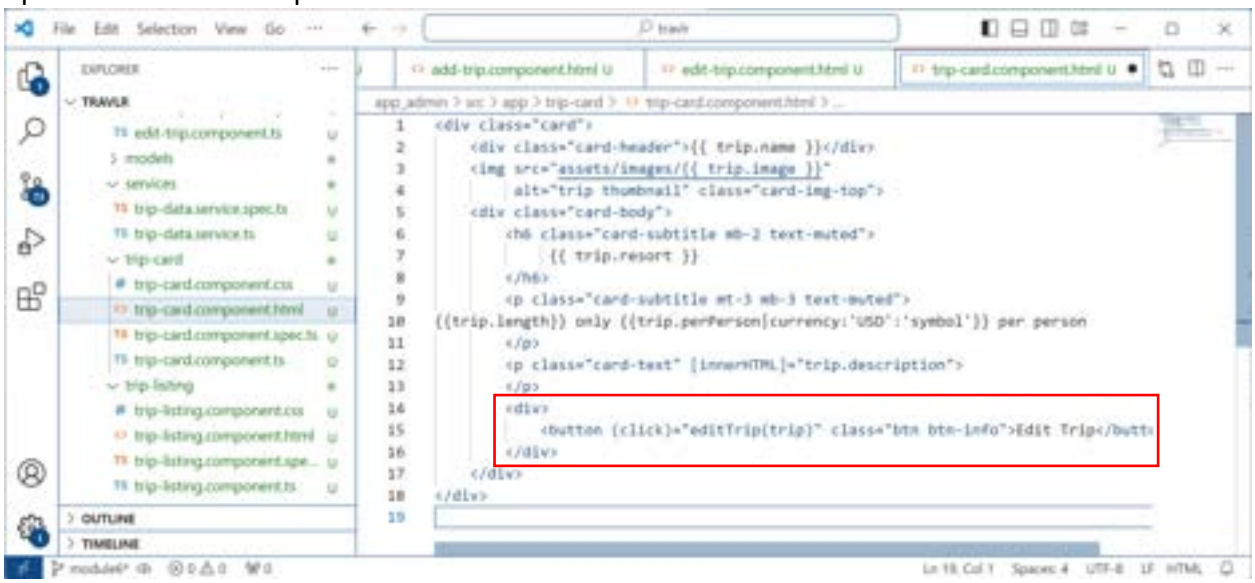
```
ng generate component edit-trip
```



7. The good news here is that the HTML for the form to edit a record is nearly identical to the form for adding a record. We will want to copy the contents of ***add-trip.component.html*** into ***edit-trip.component.html*** and replace the title of the page and the name of the form.



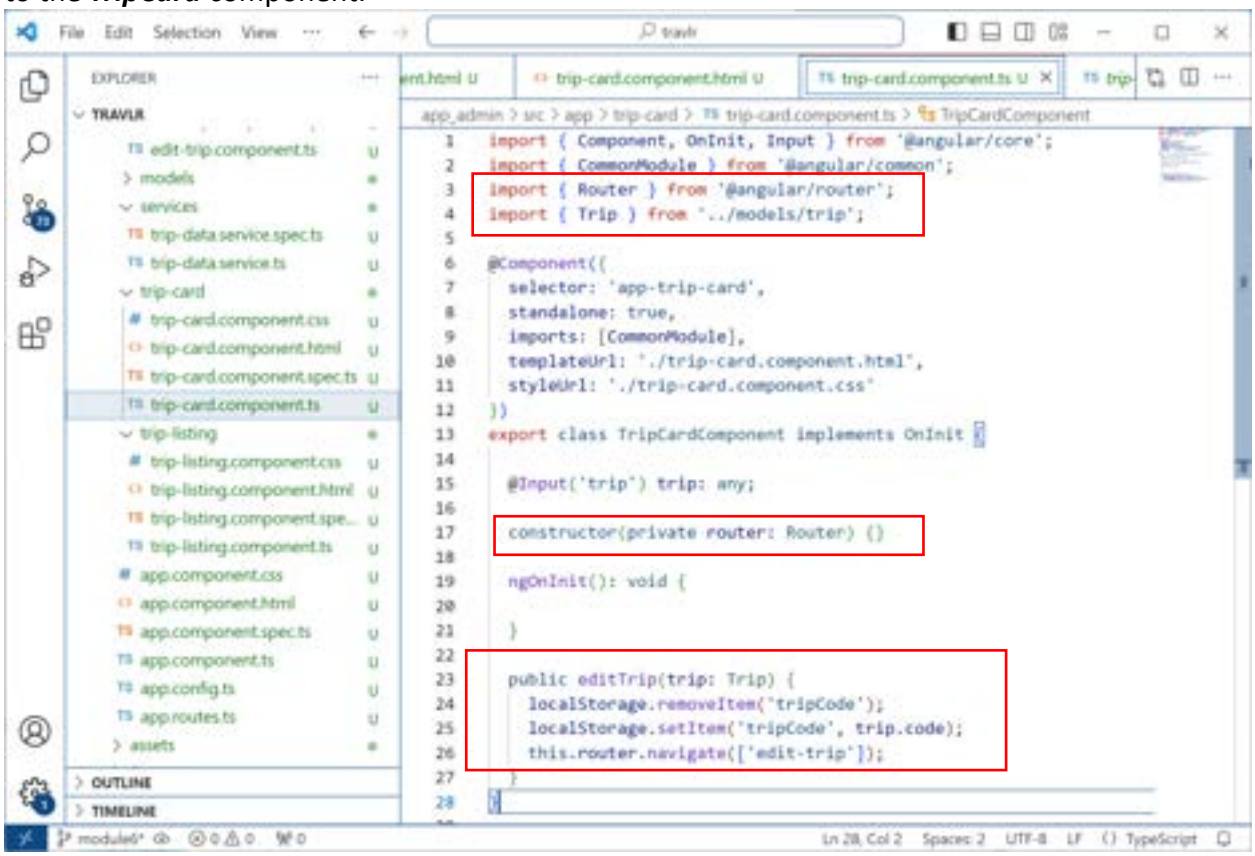
8. Now we will add an edit button to the bottom of the trip-card so that it renders the 'edit' option with each card presented.



```
<div class="card">
    <div class="card-header">{{ trip.name }}</div>
    <img src="assets/images/{{ trip.image }}"
        alt="trip thumbnail" class="card-img-top">
    <div class="card-body">
        <h6 class="card-subtitle mb-2 text-muted">
            {{ trip.resort }}
        </h6>
        <p class="card-subtitle mt-3 mb-3 text-muted">
{{trip.length}} only {{trip.perPerson|currency:'USD':'symbol'}} per person
        </p>
        <p class="card-text" [innerHTML]="trip.description">
        </p>
        <div>
            <button (click)="editTrip(trip)" class="btn btn-info">Edit Trip</butt
        </div>
    </div>
</div>
```

When the Edit Trip button is pressed the editTrip method will be called with a parameter of the trip displayed in the current trip-card.

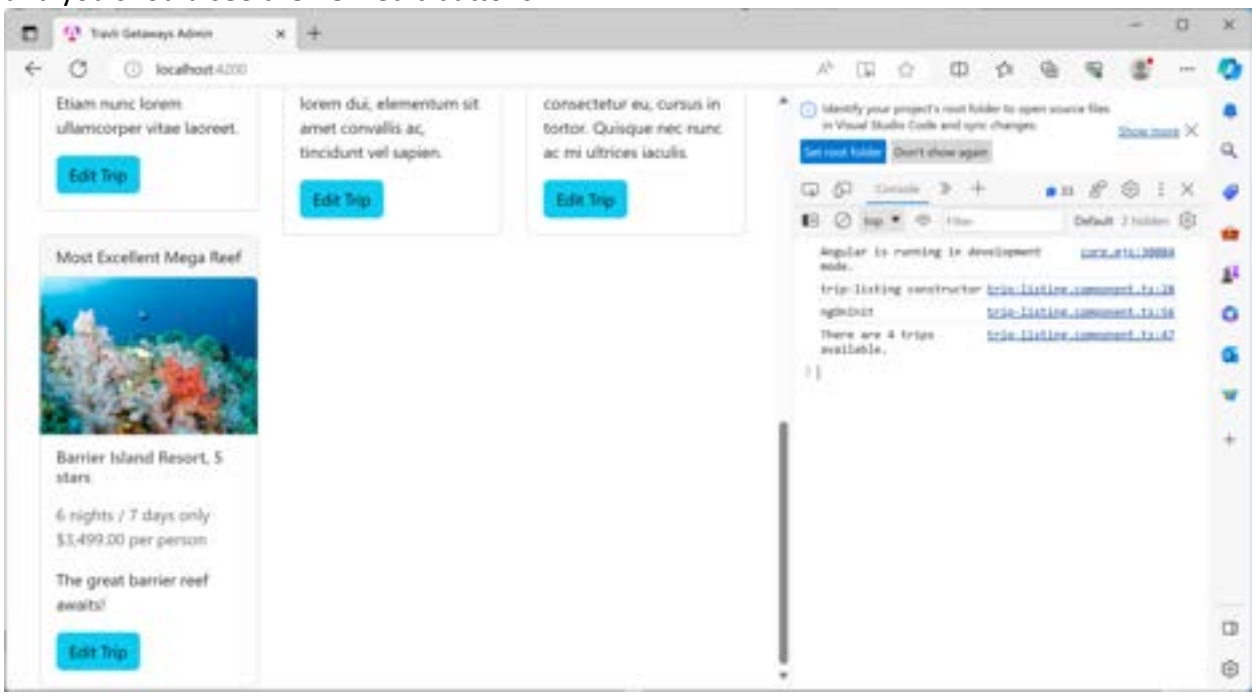
9. Now that we have edited the display side of the project, we need to add an *editTrip* method to the ***TripCard*** component.



```
import { Component, OnInit, Input } from '@angular/core';
import { CommonModule } from '@angular/common';
import { Router } from '@angular/router';
import { Trip } from '../models/trip';

@Component({
  selector: 'app-trip-card',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './trip-card.component.html',
  styleUrl: './trip-card.component.css'
})
export class TripCardComponent implements OnInit {

  @Input('trip') trip: any;

  constructor(private router: Router) {}

  ngOnInit(): void {

  }

  public editTrip(trip: Trip) {
    localStorage.removeItem('tripCode');
    localStorage.setItem('tripCode', trip.code);
    this.router.navigate(['edit-trip']);
  }
}
```

You will notice that we added two items to the import section – Router and Trip. We needed to add the Router so that the method can get angular to route the component. We added Trip so that we could have easy access to the data fields in the editTrip method.

Additionally, we modified the constructor to bind the Router object we imported. In our editTrip method, we take advantage of local storage in the browser to set the tripCode variable so that we can use it later on. Once you save this file, your browser should update and you should see the new edit-buttons.



10. Even though the buttons are now showing, they will not activate the ***edit-trip*** component yet because we have to write the logic into the ***edit-trip*** component to handle the heavy lifting. The component has to grab the existing data record and present it for update. In order to do this, we have to edit several sections in the ***edit-trip*** component.

    a. We need to add sections to the imports. These are exactly the same as what we had in the ***add-trip*** component and can be copied from there.

```
import { Component, OnInit } from '@angular/core';
import { CommonModule } from '@angular/common';
import { Router } from '@angular/router';
import { FormBuilder, FormGroup, Validators, ReactiveFormsModule }
from "@angular/forms";
import { TripDataService } from '../services/trip-data.service';
import { Trip } from '../models/trip';
```

    b. We need to adjust our imports statement so that we pull in the ReactiveFormsModule to wire up our form.

```
@Component({
  selector: 'app-edit-trip',
  standalone: true,
  imports: [CommonModule, ReactiveFormsModule],
  templateUrl: './edit-trip.component.html',
  styleUrl: './edit-trip.component.css'
})
```

c. We need to add some local variables. We need these to be able to manipulate our data.

```
  public editForm!: FormGroup;
  trip!: Trip;
  submitted = false;
  message : string = '';
```

d. We need to build the constructor. This enables us to build our form and route our component as well as pulling data from our ***TripDataService***.

```
  constructor(
    private formBuilder: FormBuilder,
    private router: Router,
    private tripDataService: TripDataService
   ) {}
```

e. We need to implement OnInit because the component does some heavy lifting when it is instantiated.

```
export class EditTripComponent implements OnInit {
```

f. We need to populate the ngOnInit method – this is where we do the heavy lifting. We grab the previously stashed tripCode and do a lookup on the existing record. We use that to populate our form and set up our editing process.

```
  ngOnInit() : void{

    // Retrieve stashed trip ID
    let tripCode = localStorage.getItem("tripCode");
    if (!tripCode) {
      alert("Something wrong, couldn't find where I stashed
tripCode!");
      this.router.navigate(['']);
      return;
    }

    console.log('EditTripComponent::ngOnInit');
    console.log('tripcode:' + tripCode);
```

```
  this.editForm = this.formBuilder.group({
    _id: [],
    code: [tripCode, Validators.required],
    name: ['', Validators.required],
    length: ['', Validators.required],
    start: ['', Validators.required],
    resort: ['', Validators.required],
    perPerson: ['', Validators.required],
    image: ['', Validators.required],
    description: ['', Validators.required]
  })

  this.tripDataService.getTrip(tripCode)
    .subscribe({
      next: (value: any) => {
        this.trip = value;
        // Populate our record into the form
        this.editForm.patchValue(value[0]);
        if(!value)
        {
          this.message = 'No Trip Retrieved!';
        }
        else{
          this.message = 'Trip: ' + tripCode + ' retrieved';
        }
        console.log(this.message);
      },
      error: (error: any) => {
        console.log('Error: ' + error);
      }
    })
}
```

g. We need to create the onSubmit method. This is what will be executed when we commit our edit. This drives the communication to the backend and routes the component back to the main screen.

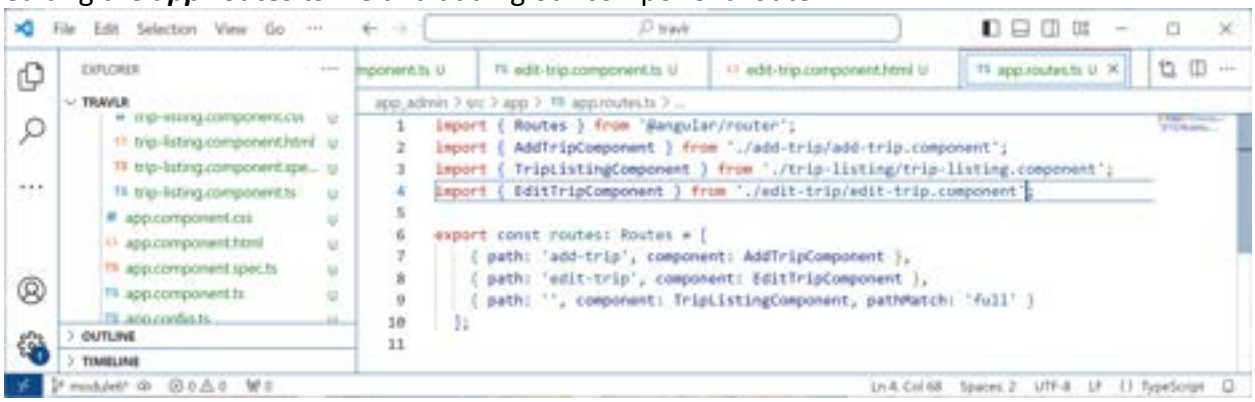
```
public onSubmit()
{
  this.submitted = true;

  if(this.editForm.valid)
  {
    this.tripDataService.updateTrip(this.editForm.value)
    .subscribe({
```

```
        next: (value: any) => {
          console.log(value);
          this.router.navigate(['']);
        },
        error: (error: any) => {
          console.log('Error: ' + error);
        }
      })
    }
}
```

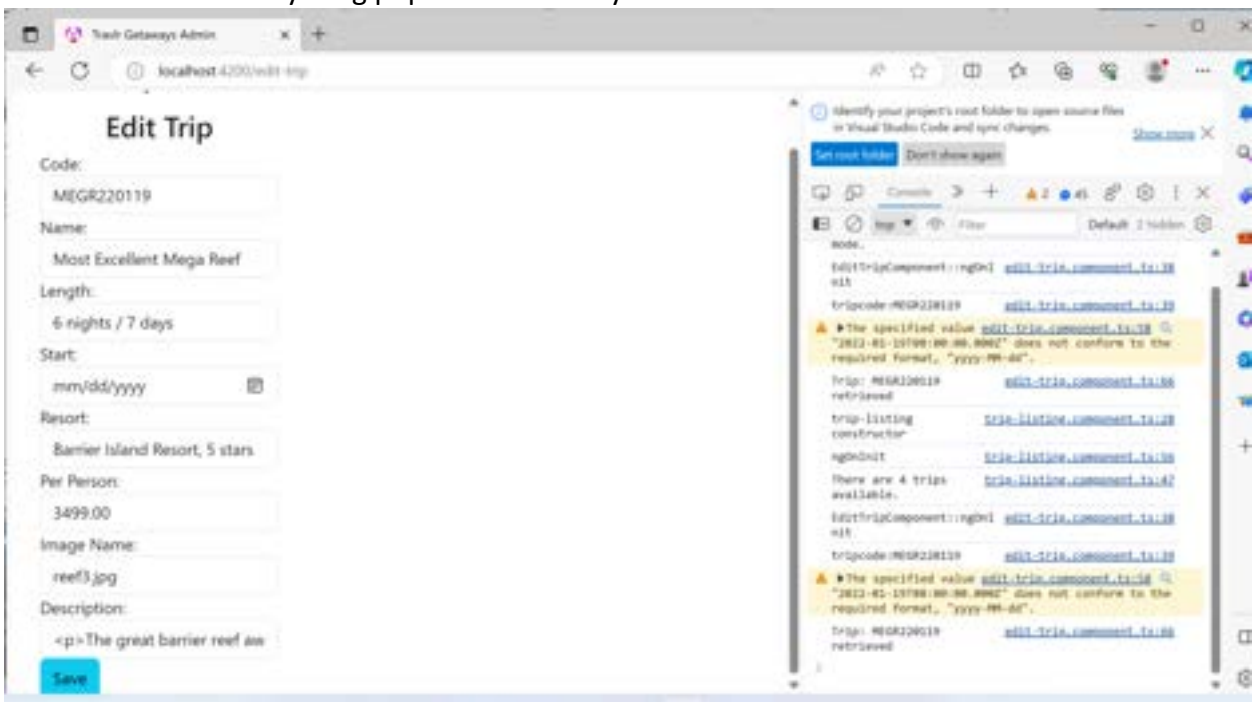
h. We need to add a quick access method to get at the form fields. This will be identical to the method we built in the ***add-trip*** component.

```
// get the form short name to access the form fields
get f() { return this.editForm.controls; }
```

11. In order to really activate this component, we need to add it to our router. We do this by editing the ***app.routes.ts*** file and adding our component route.



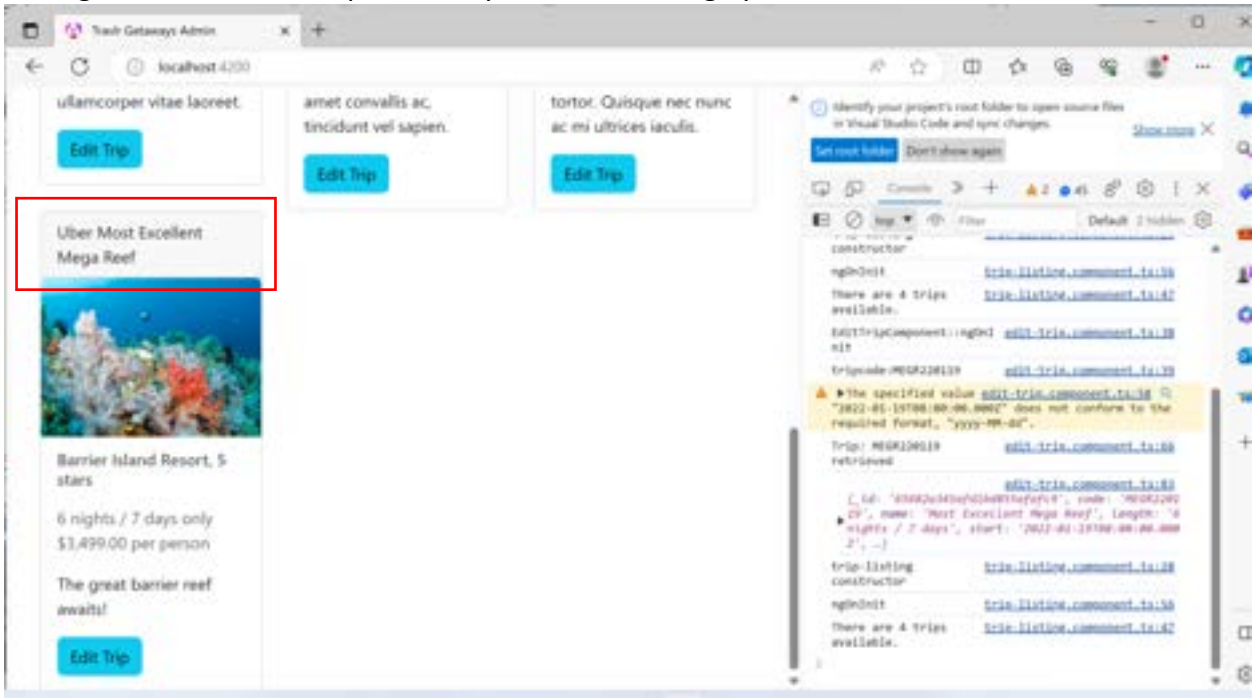
12. Now we can go back to our application and see what we have. We will select one of our trips to edit and see if everything populates correctly:



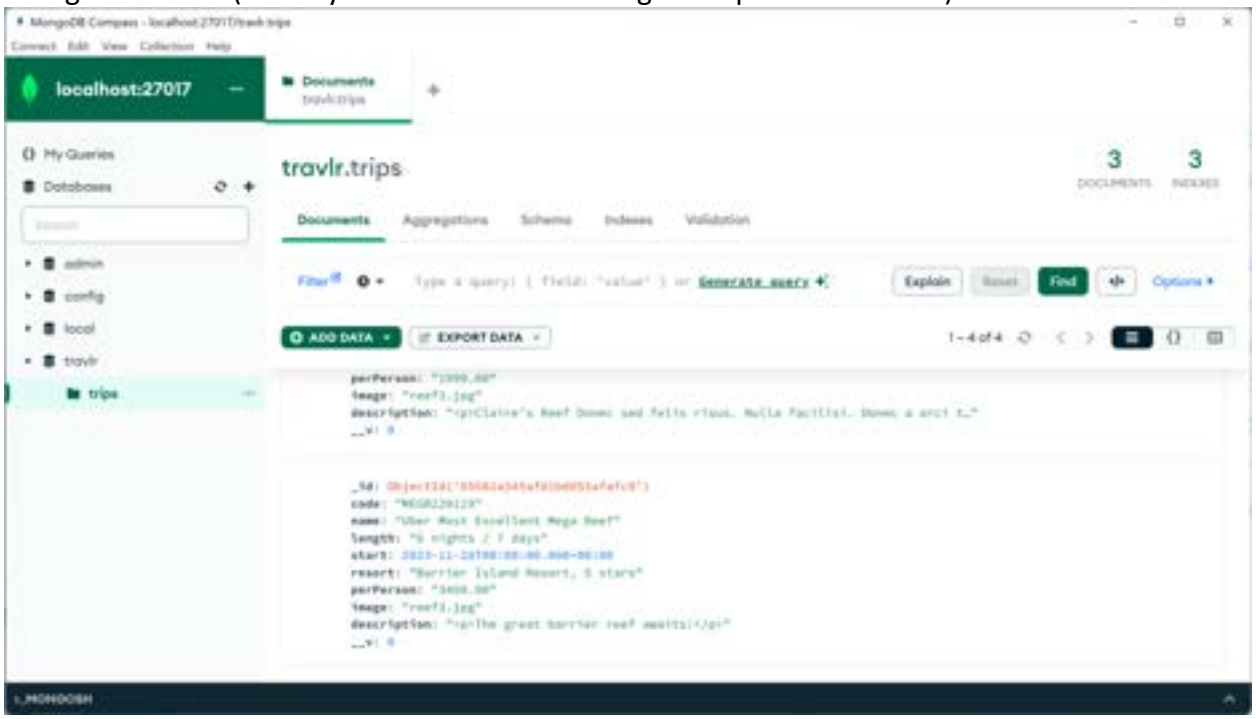
You will notice an error in the console because the value that we receive from the database is in a different format than the simple date widget we selected for our panel. It is a useful exercise to think about how you would address the difference between the data and the form. As a challenge activity – modify the code for the edit-trip component to update the date picker properly from the record retrieved from the database.

In the mean time, select a date as part of your editing process and make a small change to any other field except for the trip code. Make your selections and press save.

13. Having selected the save option, can you see the change you made reflected in the card?

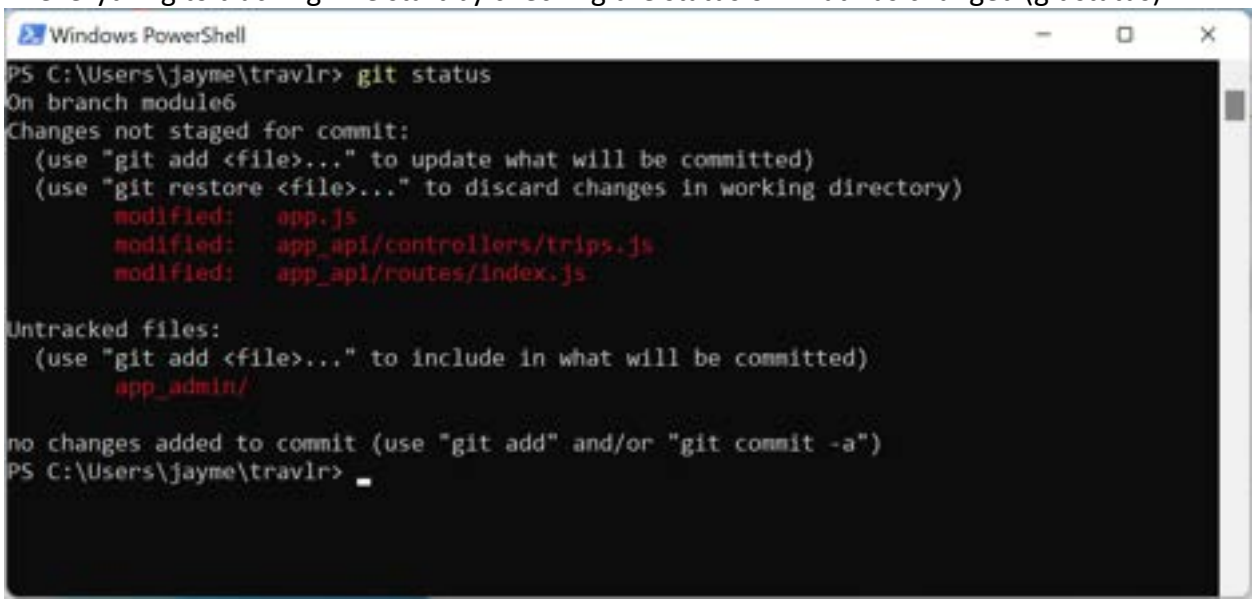


You can clearly see the edit that we made reflected in the page. If you close your browser and re-connect, you can see that the change is persistent and it has been saved in the Mongo Database (which you can also see in Mongo Compass or DBeaver).



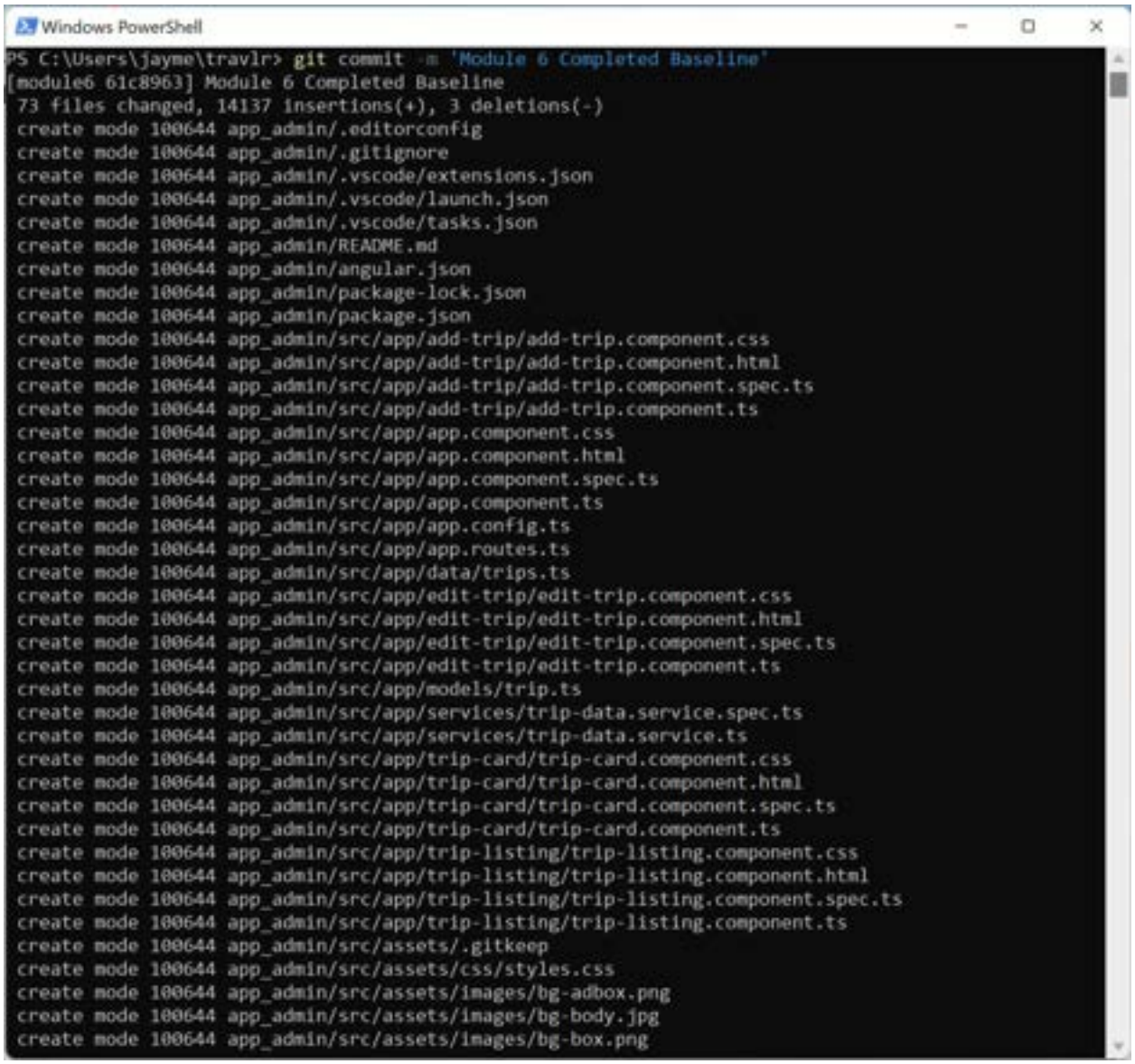
Finalizing Module 6

1. Now that we have completed Module 6, we go back to git and make sure that we add everything to tracking. We start by checking the status of what has changed (git status):



2. Then we add all of those changes into tracking (`git add .`):

3. Now we commit the changes (`git commit -m 'Module 6 completed baseline'`):



4. We push the changes back to GitHub for safekeeping (`git push --set-upstream origin module6`):

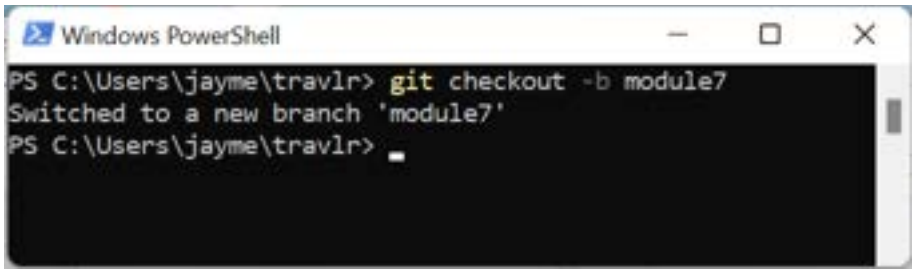
Module 7: Security

In this module, we are going to take the next logical step in creating our application by applying security to both the front-end and the back-end of the application. There are many different aspects of application security, and this Full-Stack guide will not attempt to cover all of them. However, we will be addressing some very common security related items and issues that will set you on the path to develop secure software following best-practices common throughout industry.

Create Git Branch for Module 7

Before you begin, it is important to make sure that you have created your new branch in git for Module 7. To accomplish this, we will perform the following command in a PowerShell window in the ***travlr*** project directory:

```
git checkout -b module7
```



Introduction

As you are aware, one of the key components of developing an application is security. This is a complex subject with many facets to explore across the landscape of software design and implementation and any course can only cover a portion of the overall topic. What we will be exploring in this full stack guide will cover securing the Express application and Angular application by requiring the user to login and authenticating their credentials.

Aspects of adding security to support the authentication of users is covered in Chapter 11 of your textbook. This Full-Stack guide will walk you through the implementation of a security plan that fits the ***Travlr*** application, and indicate the decision points where you would have to make an implementation choice if you were performing a similar task in a different application.

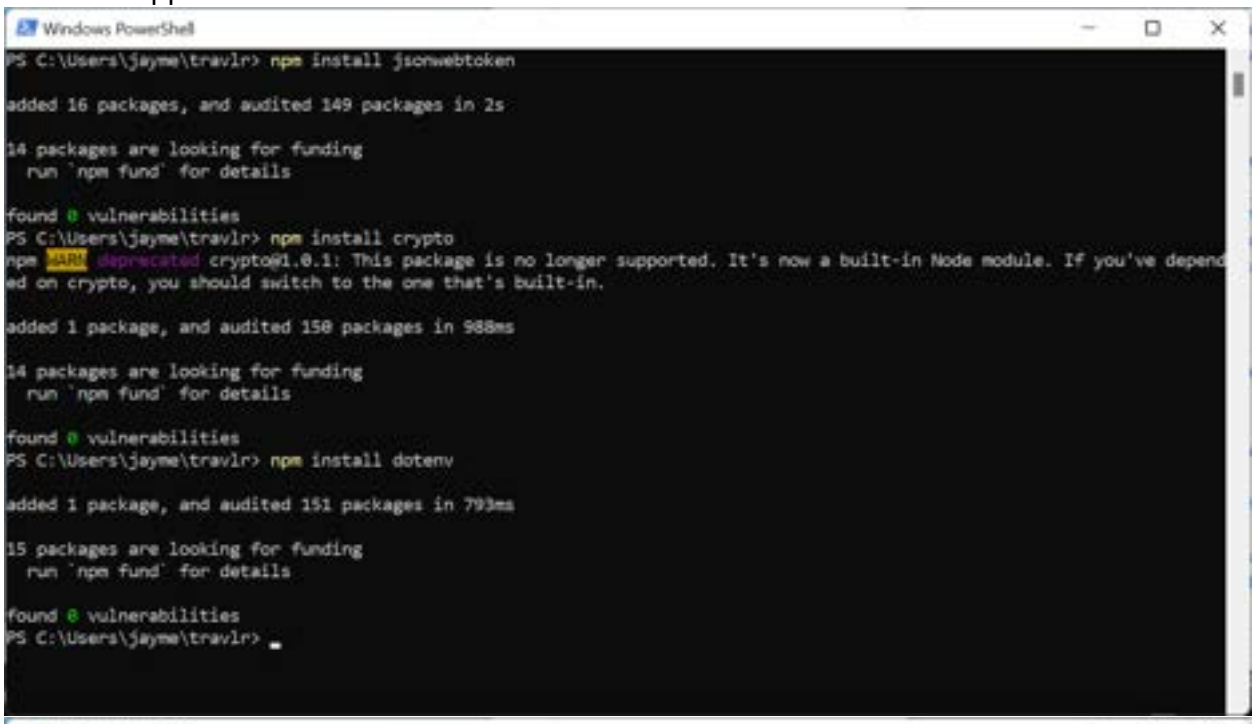
Adding User Registration and Login to the Express Backend

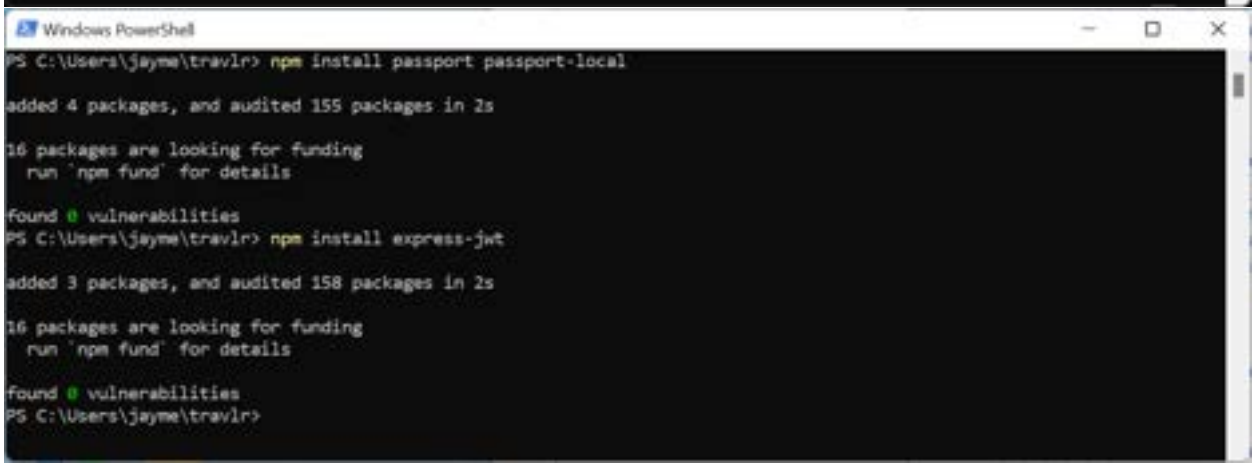
In order to begin the process of adding authentication to the Express backend for our application, we need to add some additional packages in our node environment. These packages are:

jsonwebtoken – Package for manipulating JSON Web Tokens
crypto – Package for handling cryptographic operations
dotenv – Package for reading environment information from a dot-file in the app
passport passport-local – Packages for handling authentication, and a strategy for handling local authentication

express-jwt – Packages for to enable JSON Web Tokens within Express

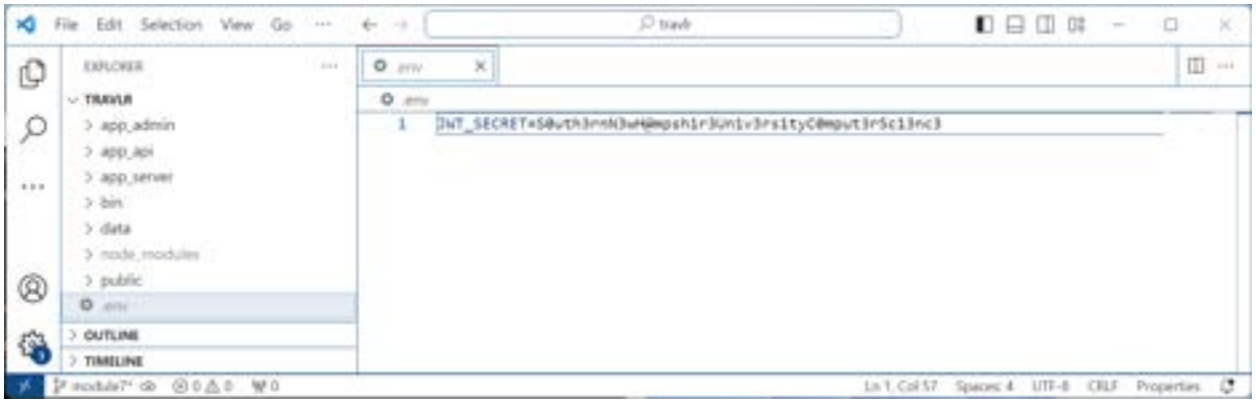
1. We begin by installing the above packages into our Express environment. Start in the root of the ***travlr*** application tree.



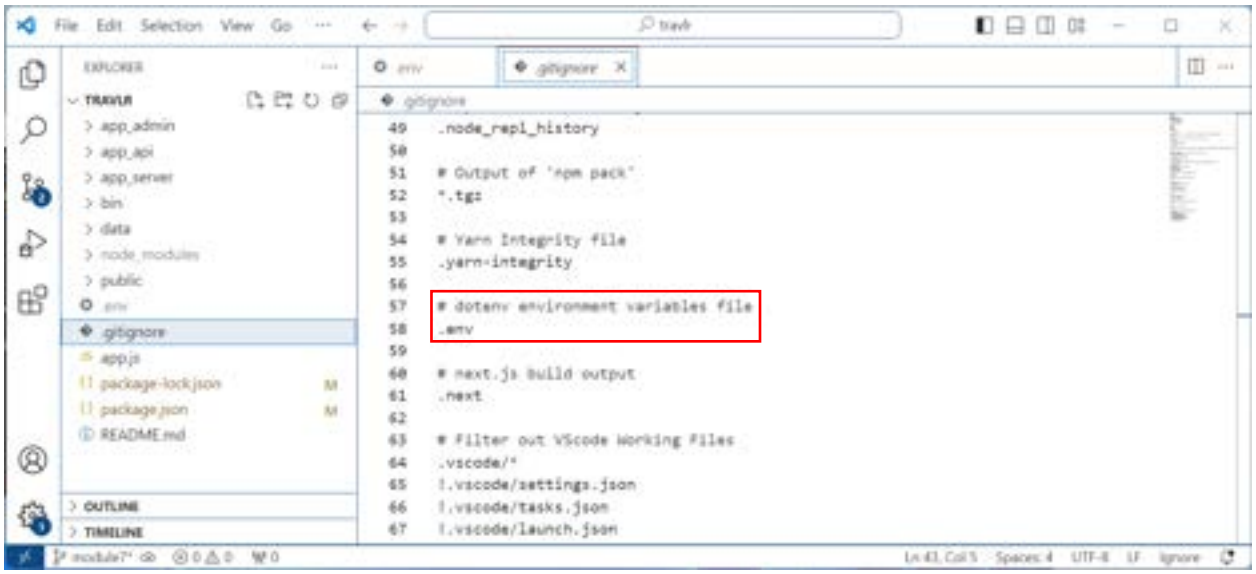


2. The next thing we are going to do is setup a *Secret* – a long text string that would be very difficult to guess – and store it in a file named ***.env*** in the ***travlr*** folder for our application. This file needs to be listed in the ***.gitignore*** file so that it does not get stored in git. It is very bad practice to store any of your secrets in a source-code repository where they can be seen by anyone that can view the repository. We will put the following string in the ***.env*** file:

```
JWT_SECRET=S0uth3rnN3wH@mpsh1r3Un1v3rs1tyC0mput3rSc13nc3
```

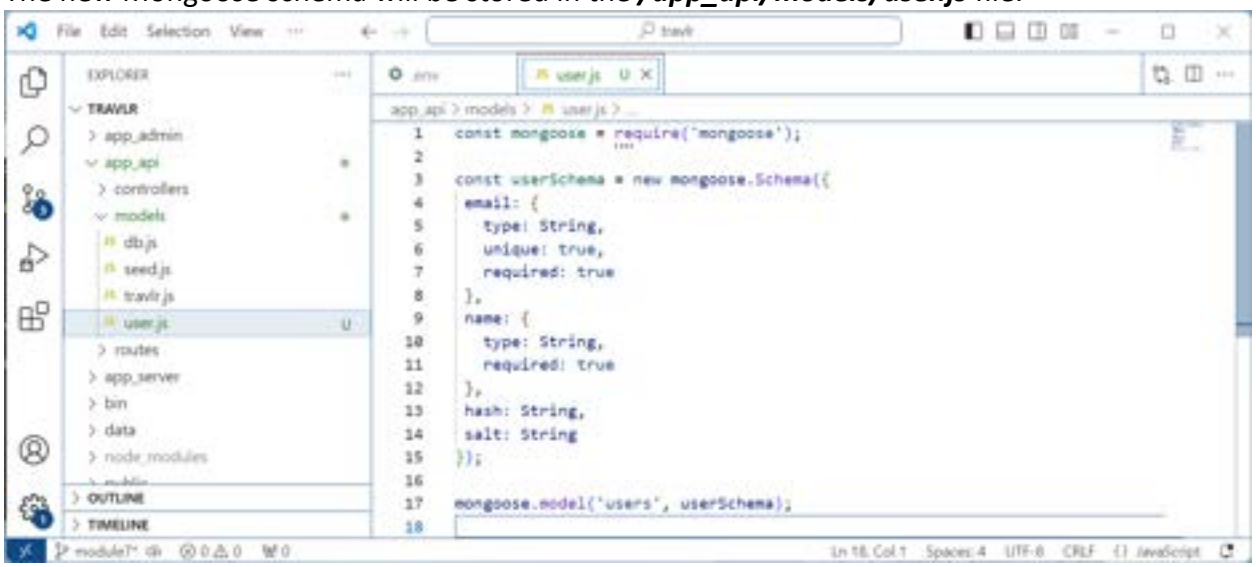


And we verify that .env files are listed in our ***.gitignore*** file so they don't end up in our repository.



3. The next step we must undertake is to add an additional schema to our Mongo Database to handle users. This is necessary as we are planning on doing local authentication of users and we need a location to store the necessary information so that we can authenticate individual users later. The User schema is going to need to be able to hold four different datapoints for each record: email, name, hash, and salt. These four attributes will allow us to identify individual users and verify their authenticity by comparing an encryption of the password they provide with the encrypted hash stored in the user record.

The new Mongoose schema will be stored in the ***/app_api/models/user.js*** file.



4. Unlike some of the other files we have in our models folder, we have to make some additions to the *user* model so that we can use the user objects throughout our application. In this respect, we will be adding three additional methods to our file:

   `setPassword` – a method to set the password for the user record
   `validPassword` – a method to verify that the password submitted was the same as the one stored in the user record
   `generateJWT` – a method to return a JSON Web Token for the specific user record

   a. First, we need to add two additional constants to our file to enable both the cryptography aspect of our functionality and the JSON Web Tokens.

   ```
   const crypto = require('crypto');
   const jwt = require('jsonwebtoken');
   ```

   b. Next, we look at our *setPassword* method. This is a void method that takes a single argument, the user's new password, and generates a cryptographically random 16-byte salt to be utilized to generate our hash. It stores the salt value, and the resulting hashed password in the user record. This is necessary because without the salt, the password hash cannot be compared. These processes lean heavily on the ***crypto*** module and the two methods:

      `randomBytes(bytes)` – Generates number of cryptographically random bytes based on the argument that is used to seed cryptographic operations.
      `pbkdf2Sync(password, salt, iterations, keylen, digest)` – Password-Based Key Derivation Function that will provide ab unique key based on the password, salt, number of iterations, specified length, and digest algorithm specified. We will be using a 16-byte salt and 1000 iterations for this

application. Increasing either or both values makes the password slightly more secure with the trade-off of increased processing time.

The code for this method looks like this:

```
// Method to set the password on this record.
userSchema.methods.setPassword = function(password){
    this.salt = crypto.randomBytes(16).toString('hex');
    this.hash = crypto.pbkdf2Sync(password, this.salt,
      1000, 64, 'sha512').toString('hex');
};
```

c. Next we look at our *validPassword* method which will provide us with an indication of whether or not the password provided matches the stored value, although it cannot be a direct match as the stored value has been hashed. This is a Boolean method that takes a single argument, the user's password. Again, this method makes use of the ***pbkdf2Sync*** method from the *setPassword* method to generate the password hash that is compared to the stored value. The return value is the result of a Boolean evaluation of equality between the stored value and the calculated value based on the given input. The processing in this method must match the processing in the previous method for a test of equality to be valid.

   The code for this method looks like this:

```
// Method to compare entered password against stored hash
userSchema.methods.validPassword = function(password) {
    var hash = crypto.pbkdf2Sync(password,
      this.salt, 1000, 64, 'sha512').toString('hex');
    return this.hash === hash;
};
```

d. Next, we will look at the *generateJWT* method that we will use to create the JSON Web Token that our application will use to pass the application and indicate that we have the necessary permissions to act in the controlled portion of the application. For this method, we are going to leverage the *sign* method from the ***jsonwebtoken*** package. The format that we will be using will utilize three parameters for the call:

   `payload` – This is the JSON object that we want to pass as part of the authentication process. This can contain values that the application can then use to make decisions on to determine access permissions.
   `secret` – This is the secret that we have stored in the ***.env*** file. This method of generating the ***JWT*** is not the only method, but it is a straight-forward application that we will be utilizing here.

`expiration time` – We are passing this as the third parameter because it is a more clear implementation than adding and *exp* value in the JSON payload.

The code for this method looks like this:

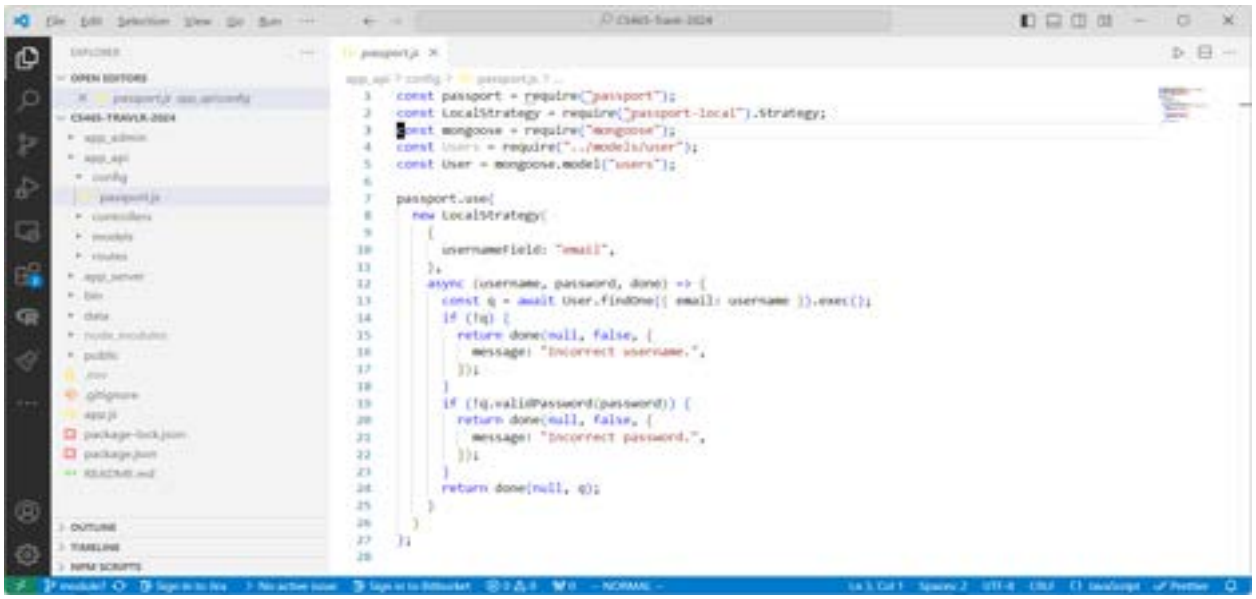
```
// Method to generate a JSON Web Token for the current record
userSchema.methods.generateJWT = function() {
    return jwt.sign(
    { // Payload for our JSON Web Token
      _id: this._id,
      email: this.email,
      name: this.name,
    },
    process.env.JWT_SECRET, //SECRET stored in .env file
    { expiresIn: '1h' });   //Token expires an hour from creation
};
```

e. Finally, we make one last change to our ***user.js*** file. At the end of the file where we have defined the model name of users and bound it to our schema, we are going to define a constant and export that constant from our module. This will make the utilization of this schema very straightforward in other modules.

```
const User = mongoose.model('users', userSchema);
module.exports = User;
```

5. The next step in this process is to build a configuration for the passport module we are using to process our authentication verifications. Passport is a 'Strategy-Based' authentication module, meaning that it can use various strategies to authenticate a user based on programmatic design. The reason we are using Passport here is because it supports a local-authentication strategy that is compatible with our Mongo Database. However, by simply changing the authentication strategy you can change the mechanism from using our local Mongo Database to a variety of other mechanisms from authenticating against a corporate LDAP directory to signing in with google.

   We will begin by creating the folder ***config*** beneath ***app_api*** and creating a ***passport.js*** file within that folder. The contents of the file should look like this:

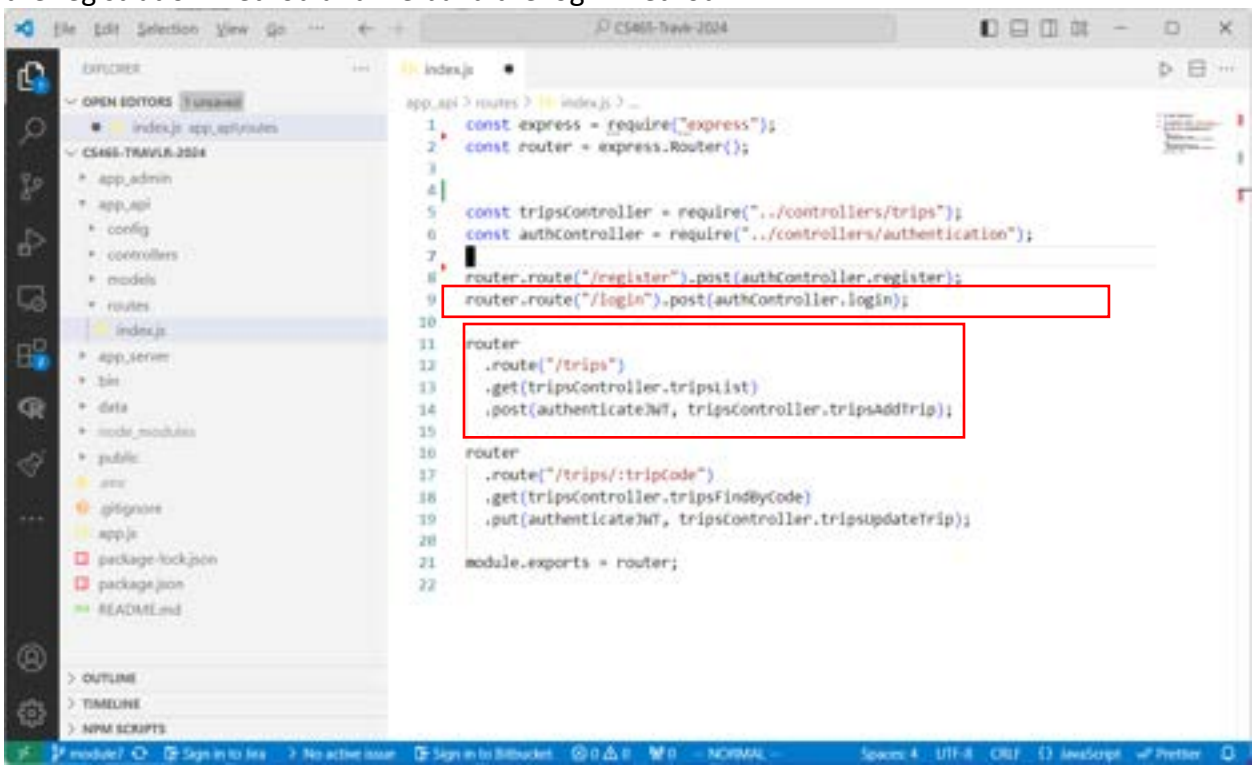


```
const passport = require("passport");
const LocalStrategy = require("passport-local").Strategy;
const mongoose = require("mongoose");
const User = mongoose.model("users");

passport.use(
  new LocalStrategy(
    {
      usernameField: "email",
    },
    async (username, password, done) => {
      const q = await User.findOne({ email: username }).exec();
      if (!q) {
        return done(null, false, {
          message: "Incorrect username.",
        });
      }
      if (!q.validPassword(password)) {
        return done(null, false, {
          message: "Incorrect password.",
        });
      }
      return done(null, q);
    }
  )
);
```

6. The next step will be to create a controller for authenticating users. This will be created in the file ***/app_api/controllers/authentication.js***. We will begin by creating a controller that will handle user registration. This will allow us a convenient point to test what we have accomplished so far prior to moving forward with a login capability.

```
const mongoose = require('mongoose');
const User = require('../models/user');

const register = async(req, res) => {
    // Validate message to insure that all parameters are present
    if (!req.body.name || !req.body.email || !req.body.password) {
        return res
            .status(400)
            .json({"message": "All fields required"});
    }

    const user = new User(
        {
          name: req.body.name,          // Set User name
          email: req.body.email,        // Set e-mail address
          password: ''                  // Start with empty password
        });
    user.setPassword(req.body.password) // Set user password
    const q = await user.save();

    if(!q)
    {
        // Database returned no data
        return res
            .status(400)
            .json(err);
    } else {
        // Return new user token
        const token = user.generateJWT();
        return res
            .status(200)
            .json(token);
    }
};

module.exports = {
    register
};
```

7. In the next step, we will register the new route for the ***registration*** controller in ***/app_api/routes/index.js***. This is a repeat of what we have already done for the ***trip*** controller and should be becoming more familiar. We will do this again once we have tested the registration method and we build the login method.

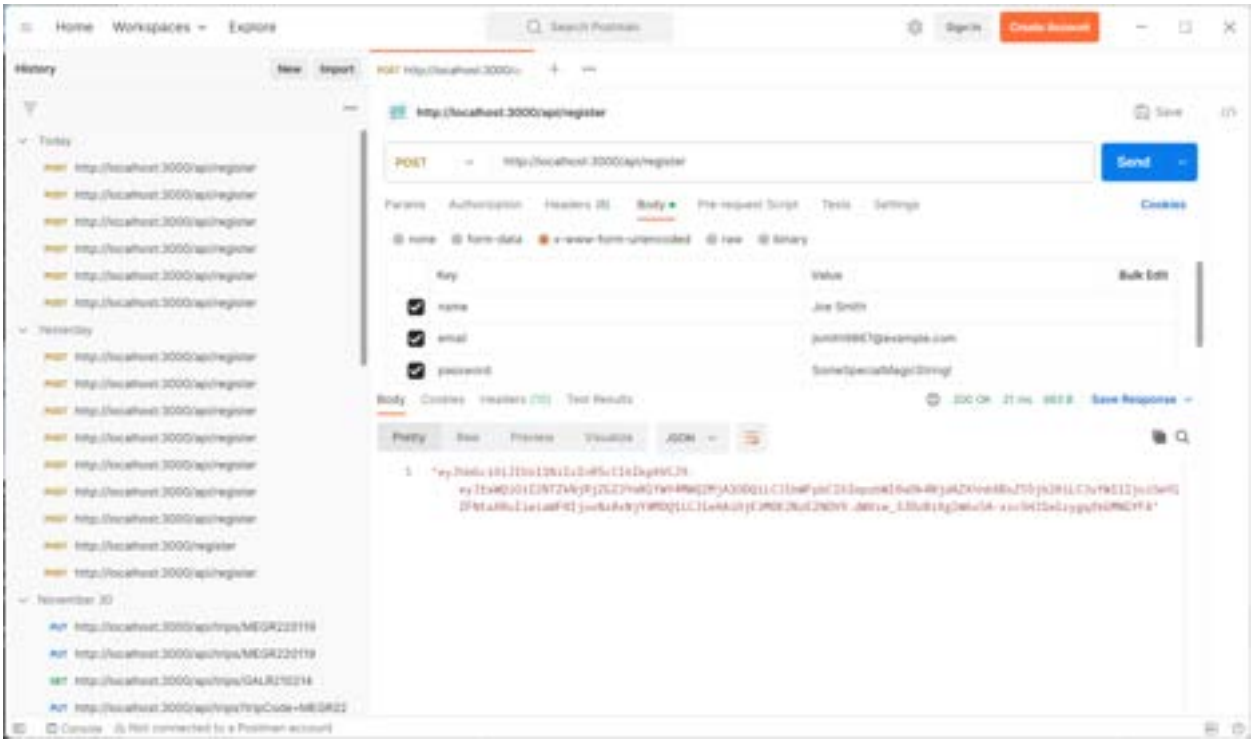


8. There is one more change we have to make prior to testing our registration endpoint, and that is adding the code to the ***app.js*** file to pull in the contents of our ***.env*** file. For this, we need only add the following line to the ***app.js*** file:

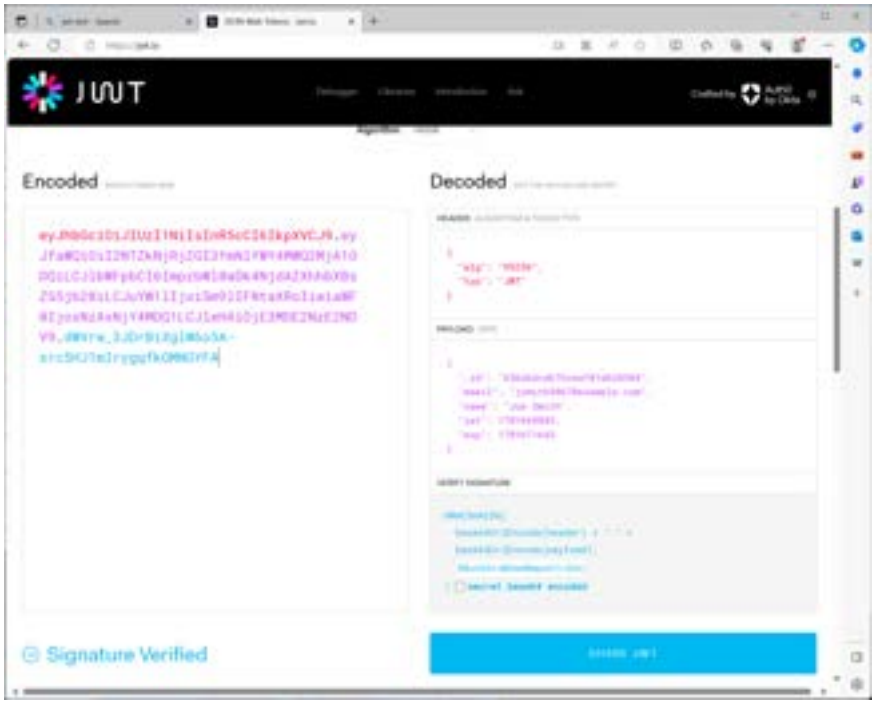
```
require('dotenv').config();
```

This line pulls in the ***dotenv*** module, and the *config()* method reads the ***.env*** file, allowing us to bring the variables defined in the file into our memory space. Make sure that you restart your server after making this change.

9. Now that we have our first of the two new controllers wired up to our application it is time to try and test the controller with Postman. To do this, we need to setup a Postman query that will send the name, email, and password attributes in the x-www-form-encoded format in the body of the post request. When we execute send, it will send the request to the register endpoint, create a new user in our Mongo DB, and return a JSON Web Token representative of that user.



Once we have this JSON Web Token, we can verify that it is correct by going to the online-verifier at https://jwt.io and pasting our token into the Encoded box on the left and pasting our secret (everything to the right of the = in the .env file) in the textbox on the lower right in the 'Verify Signature' block. You should see the note: 'Signature Verified' in the lower left-hand side of the page:



10. Now that we have the first endpoint built that allows us to register a user, we need to build the second endpoint. We will begin this task by building the second controller that we will

be using for the /login endpoint. There are three parts to this process, and all of the code for this controller will go in the ***/app_api/controllers/authentication.js*** file.

The first part of this process will be to pull in one more package for use by the login controller. The following should be placed at the top of the file to pull in the passport module:

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

The next part of the controller will be the controller method itself. This method will delegate the authentication process to the passport module and process the results of that authentication call. If everything works, then a new JSON Web Token will be returned.

```
const login = (req, res) => {
    // Validate message to ensure that email and password are present.
    if (!req.body.email || !req.body.password) {
      return res
        .status(400)
        .json({"message": "All fields required"});
    }

    // Delegate authentication  to passport module
    passport.authenticate('local', (err, user, info) => {
      if (err) {
        // Error in Authentication Process
        return res
          .status(404)
          .json(err);
      }

      if (user) { // Auth succeeded - generate JWT and return to caller
        const token = user.generateJWT();
        res
          .status(200)
          .json({token});
      } else { // Auth failed return error
        res
          .status(401)
          .json(info);
      }
    })(req, res);
};
```

The third and final part of this change is adjusting the exports statement for this module to also export the login controller in addition to the already running register controller. To make this work, please update the ***module.exports*** block as follows:

```
// Export methods that drive endpoints.
module.exports = {
    register,
    login
};
```

11. Now we must go back and add the route for our new login endpoint. This is an edit to the ***/app_api/routes/index.js*** file. We could have made this change when we were adding the register endpoint, but it is generally good practice to restrict changes to one thing at a time when building an application to reduce the complexity of any debugging operations. Since we have already added the ***register*** endpoint we don't need to pull in the authentication module again, we just have to add the code for the login endpoint.

```
// define route for login endpoint
router
    .route('/login')
    .post(authController.login);
```

12. The next portion of wiring up the *login* controller in our application requires some additional changes to the ***app.js*** file. We need to wire in the ***passport*** module so we will be adding two stanzas to our file at the bottom of the variable section:

```
// Wire in our authentication module
var passport = require('passport');
require('./app_api/config/passport');
```

And we must add an initializer for our passport module which we will place beneath the *express.static* statement like this:

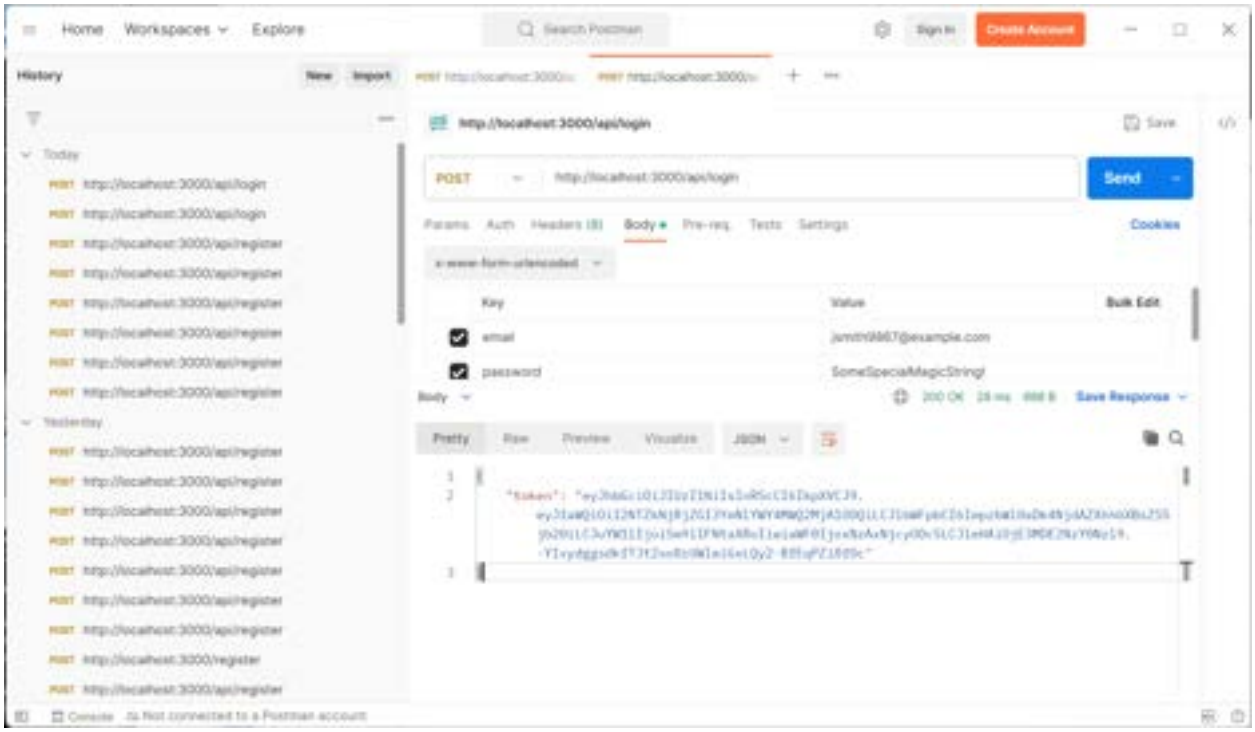
```
app.use(express.static(path.join(__dirname, 'public')));
app.use(passport.initialize());
```

And we must add an additional tag to our 'Allow-Headers' line in the block that we use to define our CORS capabilities. The tag that we will add allows Authorizations.

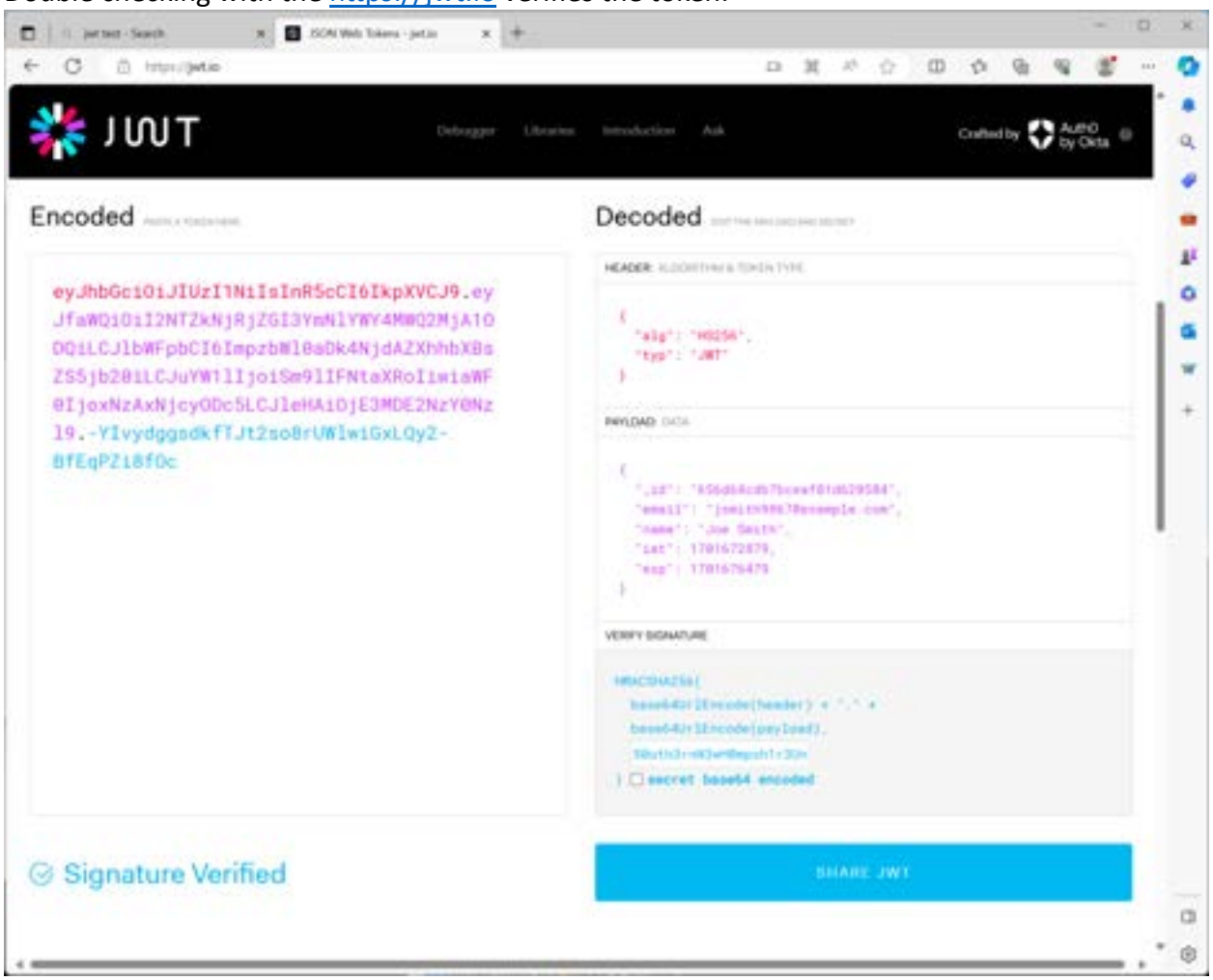
Finally, we add a block to catch an error that would indicate an Unauthorized login attempt.

```
// Catch unauthorized error and create 401
app.use((err, req, res, next) => {
  if(err.name === 'UnauthorizedError') {
    res
      .status(401)
      .json({"message": err.name + ": " + err.message});
  }
});
```

13. Once we have made these changes, we can restart our application and test with Postman. For these tests, we will use the user we created in step 9. For this process, we need to define a POST body with *email* and *password* variables defined. Pressing send should result in the return of a new JWT token:



Double checking with the https://jwt.io verifies the token:



Wrapping Express API Calls for Authentication

Now that we have our endpoints created to allow for user registration and login, we need to determine how we are going to require our API methods that change data in the database to require authenticated users. While this is challenging, it is not the only challenge in dealing with application security as this covers authentication but not authorization. This Full-Stack guide will complete the authentication portion of development but it will be left to the reader to experiment with the code and determine how to add authorization capabilities into the codebase. For now, we will assume that any authenticated user is authorized to make changes (dangerous in production but common in a development environment).

To address the authentication portion, we will be creating a method based on the ***jsonwebtoken*** package and inserting the function as middleware in our route model. This will insure that the authentication method is run whenever a protected route is accessed. The method will grab the authentication token from the request header and validate the token. If the token is broken, malformed, or expired, the access request will be rejected with a 401-error message.

We will begin by inserting one more line of code at the top of the ***/app_api/routes/index.js*** file to pull in the ***jsonwebtoken*** package.

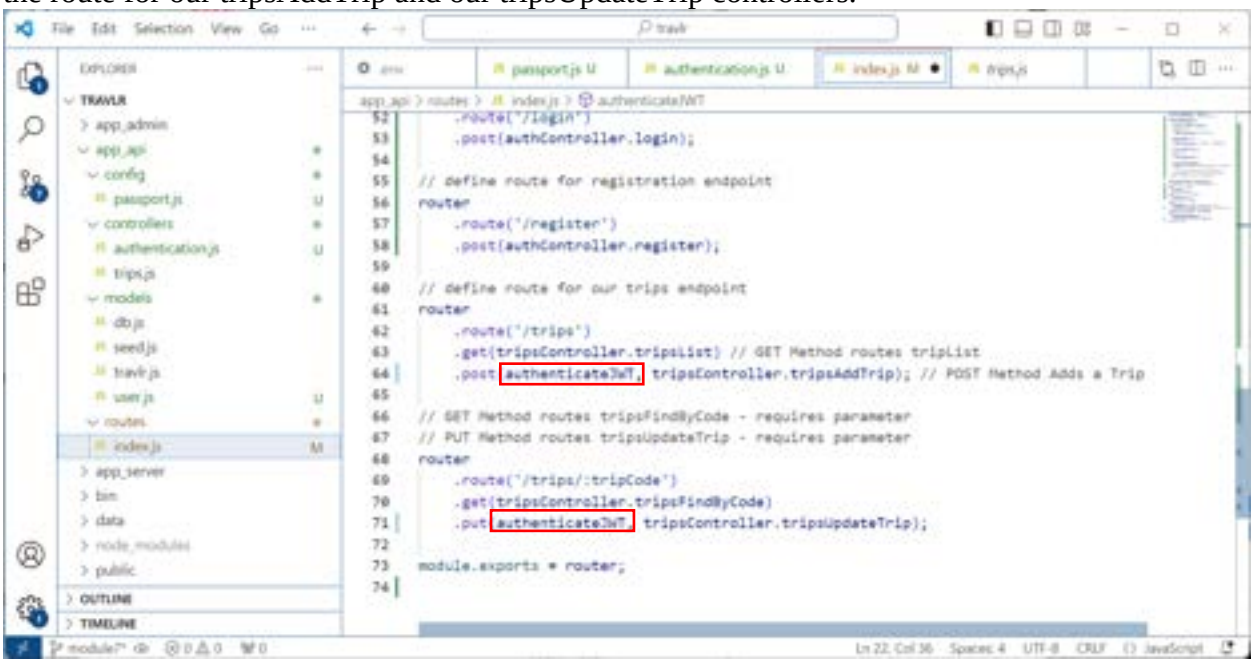
```
const jwt = require('jsonwebtoken'); // Enable JSON Web Tokens
```

Then we will create a method that we will name ***authenticateJWT***, because its job is to authenticate JWTs. You will note that there are several ***console.log*** statements here. The purpose of these statements is to give you an opportunity to see what is happening while the code is running as the console.log will print to your PowerShell window where you are running your Express application. The process is straight-forward. Pull the authorization header from the request, parse out the JWT and verify. If the token verifies, put the decoded token back onto the request object as the auth attribute, if not return a HTTP 401 error. The code for this method follows:

```
// Method to authenticate our JWT
function authenticateJWT(req, res, next) {
    // console.log('In Middleware');

    const authHeader = req.headers['authorization'];
    // console.log('Auth Header: ' + authHeader);

    if(authHeader == null)
    {
        console.log('Auth Header Required but NOT PRESENT!');
        return res.sendStatus(401);
    }

    let headers = authHeader.split(' ');
    if(headers.length < 1)
    {
        console.log('Not enough tokens in Auth Header: ' +
headers.length);
        return res.sendStatus(501);
    }

    const token = authHeader.split(' ')[1];
    // console.log('Token: ' + token);

    if(token == null)
    {
        console.log('Null Bearer Token');
        return res.sendStatus(401);
    }

    // console.log(process.env.JWT_SECRET);
```

```
    // console.log(jwt.decode(token));
    const verified = jwt.verify(token, process.env.JWT_SECRET, (err,
verified) => {
        if(err)
        {
            return res.sendStatus(401).json('Token Validation Error!');
        }
        req.auth = verified; // Set the auth paramto the decoded object
    });
    next(); // We need to continue or this will hang forever
}
```

There is one more thing that we need to do. We need to inject our new middleware method into the route for our tripsAddTrip and our tripsUpdateTrip controllers.

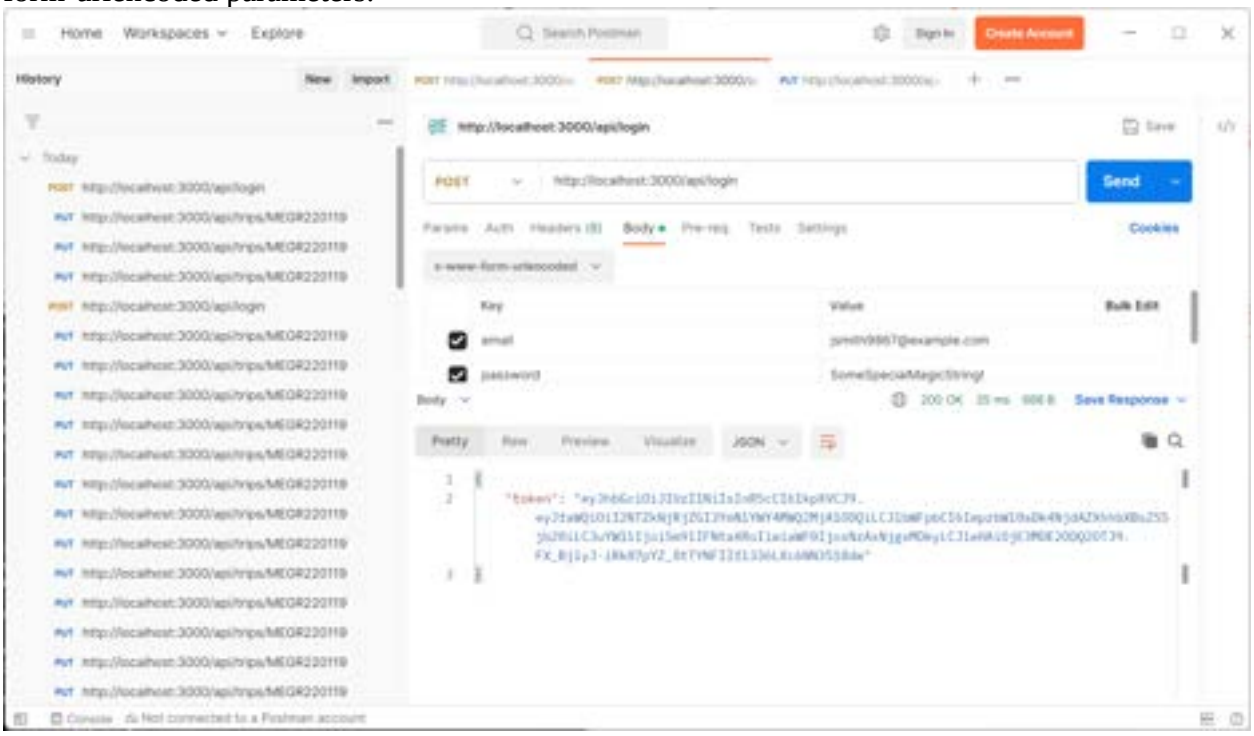


Once we have saved the file and restarted our application, we will no longer be able to test these endpoints with Postman without first authenticating.

Testing API Calls that Require Authentication

Fortunately, Postman provides a very good resource for testing API calls that require authentication. We will first go back to the test that we performed when we built the ***login*** API endpoint. We will setup a Postman call to login to our API and get a clean JWT. This is often necessary as the token is currently configured to expire after an hour – so when you are testing it is often advisable to hit the ***login*** endpoint first to avoid chasing a problem that is really an expired token.

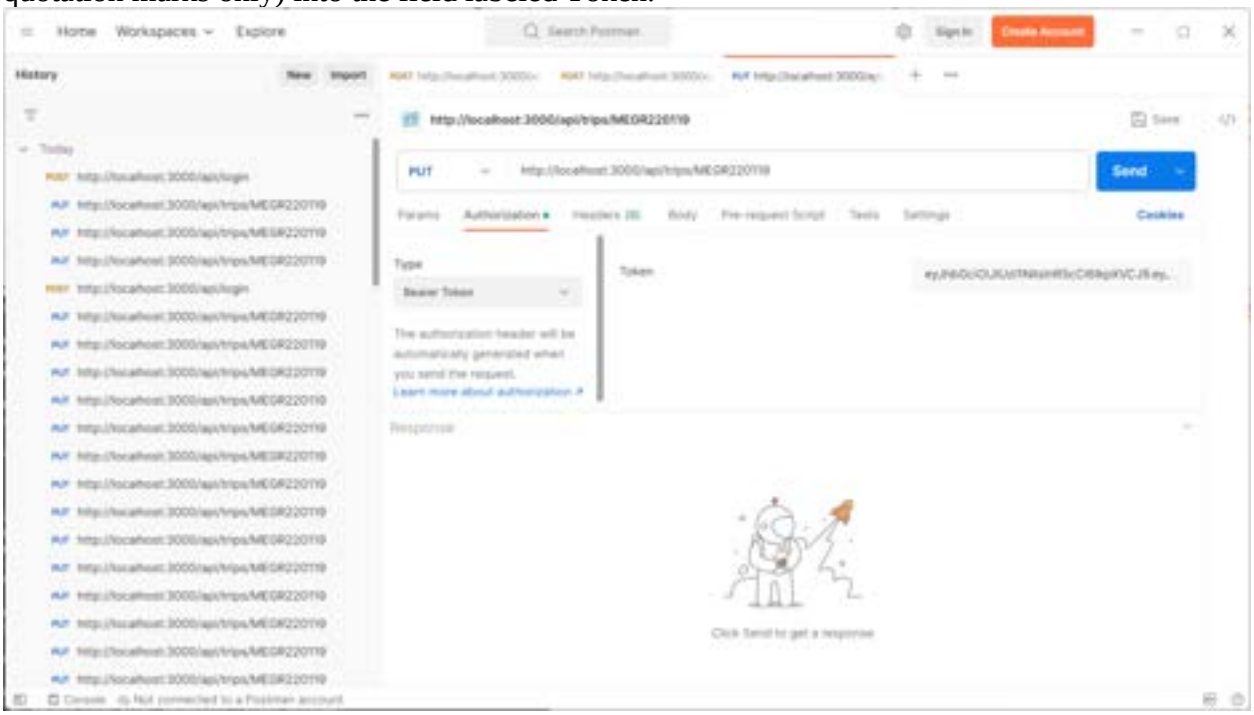
As we tested previously, you need to create a POST call to http://localhost:3000/api/login and you need to make sure that you add the ***email*** and ***password*** attributes to the body as x-www-form-urlencoded parameters.



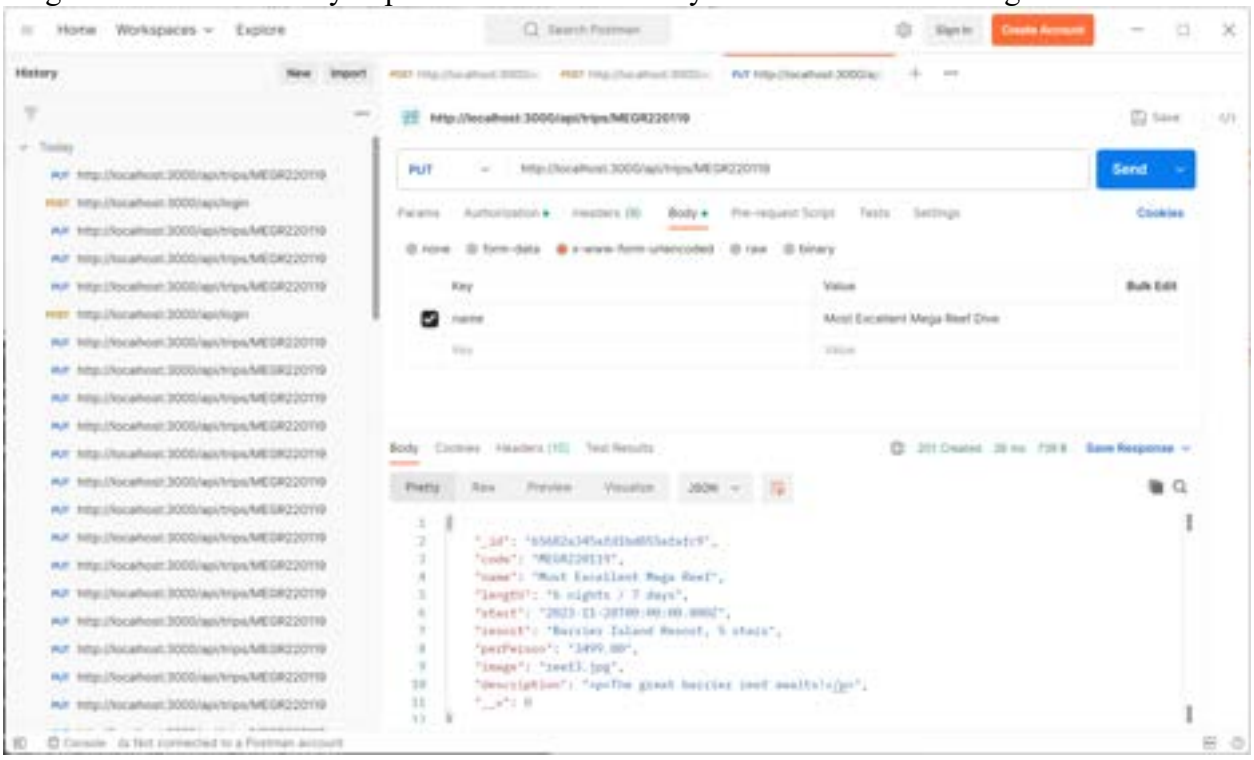
This will result in the API returning a JSON object and the value of the 'token' parameter that is returned is your JWT. To make the next test, you will want to create another Postman call. We will use the PUT verb to make a change to one of our trips. Before we change anything, we are going to want to add some data to the Authorization tab. This is where Postman handles the necessary mechanics to provide Authorization headers for your call.

The type of Authorization you will want to select is called 'Bearer Token'. This is the category of Authorization Tokens that a JWT belongs to. You then need to copy the contents (between the

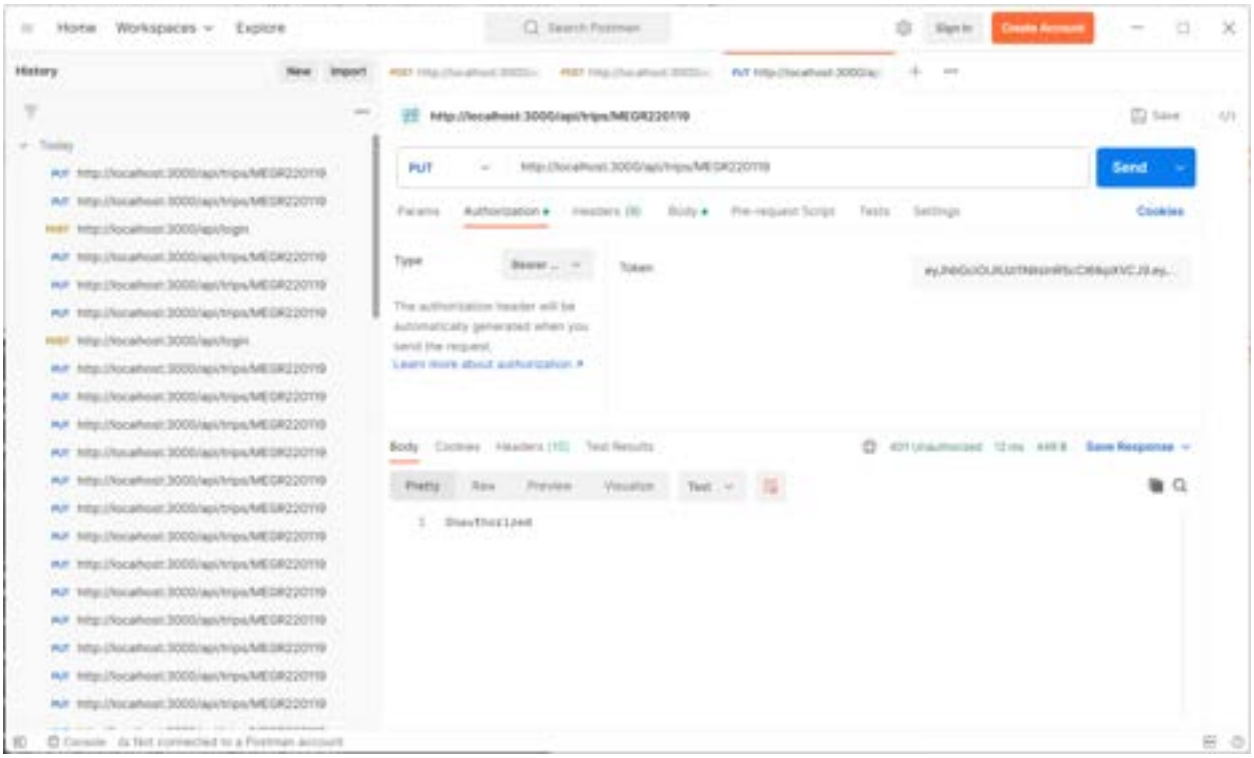
quotation marks only) into the field labeled Token.



This allows Postman to send the necessary headers to our API. The next thing we will do to setup our test is to add a parameter to the ‘Body’ tab. This will be the parameter we want to change for our trip. In this case, I am going to change the name of the trip to “Most Excellent Mega Reef Dive”. When you press the ‘Send’ button you should see something like:



As you can see, the change was made as expected. But this just tests what is supposed to happen if everything goes well. So what does it look like if there are problems? Let's test this by changing our Authorization configuration. If you go back to your Authorization page, change one character in the Token (it doesn't matter which one or how you change it), and re-send the query. In my example, I am changing the last character in the Token from a ***w*** to an ***x***. And this is the result:



As you can see from this test, our middleware is working as we have designed it and will prevent access from anyone that does not have a valid bearer token.

Add Authentication to the Angular SPA Frontend

Now that we have the backend squared away and requiring authentication for the methods that permit data to be altered in our database, we must address the front-end application. You will note that at this point, the Angular application will still run and display the trip cards, but trying to perform an Add or Edit function will fail. So we need to determine what has to happen in order for us to utilize the new protected API calls that we implemented in our Express application. Chapter 12 in the textbook covers some of the aspects of adding security to support user authentication and protecting associated programmatic resources and is a good reference for the structure behind what we will now attempt.

If we were to list the items that need to be addressed to accomplish this, it would look something like this:

1. Determine and implement how we are going to handle the storage of the JWT in our Angular application.

2. Determine and implement how we are going to represent a user in our Angular application.
3. Determine and implement how we are going to handle authentication from our Angular application to the Express back end.
4. Determine and implement the necessary chances to the ***trip-data-service*** component to include the necessary authentication calls to the API endpoints.
5. Determine and implement a mechanism that will provide the user a visual queue to 'Login' to our application.
6. Implement the necessary component construction to support the 'Login' process identified by our visual queue.
7. Implement the necessary routing for our new components.
8. Modify the ***trip-card*** component to add any applicable display logic for the buttons.

Data Storage for Angular Applications

When dealing with data items that must persist through some portion of an application there are several possible choices that need to be evaluated each with their own considerations. In the case of this application, we are dealing with the JWT and must be able to provide it back to the server-side APIs as part of each call. Because the server provides us with the JWT in the body of the response, we can remove the option of storing the data in a Cookie which is one of the more common storage methods for web applications. This has the fortunate benefit of protecting the application against Cross-Site Request Forgery (XSRF) attacks but comes with the downside of needing to manage the interaction with the JWT manually instead of allowing the browser to manage the cookie transactions automatically.

A practical choice of handling the JWT is to take advantage of the Local Storage API which allows us to store Key/Value pairs in a manner where they are accessible to the application running in the browser. With this mechanism we will still be vulnerable to script-injection attacks (XSS) but there are always trade-offs to be considered when designing applications.

We will begin by setting up a storage class for our Angular application that will allow us to inject access to our local data to any of our other components.

```
ng generate class storage
```

The storage class should be created in the ***/app_admin/src/app*** folder and should contain the following code:

```
import { InjectionToken } from '@angular/core';

export const BROWSER_STORAGE = new
      InjectionToken<Storage>('Browser Storage', {
      providedIn: 'root',
      factory: () => localStorage
      });
```

```
export class Storage {
}
```

The constant BROWSER_STORAGE will be the token that is utilized when pulling the local storage capabilities into other components in our application.

Representing User Data in our Angular Application

The next thing that we must decide is how we will represent the data associated with our user in our application. The scope of this data structure will be reduced from how the data is represented in the Express application as we do not need to concern ourselves with storing and representing the password hash or the associated salt. We will create a utility class to hold the data objects that we will need to track, and we will move it into the ***/app_admin/src/app/models*** folder.

```
ng generate class user
```

The contents of the file should be quite simple as it will only need to track ***email*** and ***name*** attributes.

```
export class User {
    email: string;
    name: string;

    constructor()
    {
        this.email = '';
        this.name = '';
    }
}
```

Handling Authentication in our Angular Application

Having decided upon the storage requirements for our JWT object and creating the data model to represent the user information we need to proceed with developing the mechanisms necessary to handle the authentication process. With the architecture provided by the Angular framework one of the common ways of addressing this requirement is to generate both a class to handle the representation of the JWT and a service to handle the authentication process.

1. We will begin by creating a class to represent the results of the authentication process. We will call this AuthResponse and it will also live in our ***models*** folder. This is purposeful because we can then utilize the same structure if we change our overall architecture in the future to utilize something other than JWTs.

```
ng generate class AuthResponse
```

The contents of this class are very minimal as the only attribute to track right now is the JWT.

```
export class AuthResponse {
    token: string;

    constructor()
    {
        this.token = '';
    }
}
```

2. The next step is considerably more complex as we will build a new service to handle the operations associated with authentication. This service once created will be moved into our existing ***services*** folder to maintain the application structure.

```
ng generate service authentication
```

This generates the base contents of the new service which we will move into our ***/app_admin/src/app/services*** folder. Because of the complexity of this service, we will discuss each piece individually. The first will be the items that we need to import into the service to provide the access and features needed to develop our actions.

```
import { Inject, Injectable } from '@angular/core';
import { BROWSER_STORAGE } from '../storage';
import { User } from '../models/user';
import { AuthResponse } from '../models/auth-response';
import { TripDataService } from '../services/trip-data.service';
```

We need *Inject* and *Injectable* to manage our access to the Storage provider we created and because the service we are creating is itself *Injectable* in other Angular components. We need *BROWSER_STORAGE* as this provides access to our local Storage provider. The *User* object provides us a means of representing our user data, while AuthResponse provides representation for our JWT. Lastly, TripDataService is needed as the service will be modified to add our *login* and *register* endpoints with which we will interact.

```
// Setup our storage and service access
constructor(
   @Inject(BROWSER_STORAGE) private storage: Storage,
   private tripDataService: TripDataService
 ) { }

// Variable to handle Authentication Responses
authResp: AuthResponse = new AuthResponse();
```

The constructor is straightforward, we *Inject* our storage provider so that its data contents are persistent across any modules or components where it is used. Additionally we wire up our TripDataService so that we can communicate with our API endpoints, and define a variable to handle Authentication Responses.

```
// Get our token from our Storage provider.
// NOTE: For this application we have decided that we will name
// the key for our token 'travlr-token'
public getToken(): string {
  let out: any;
  out = this.storage.getItem('travlr-token');

  // Make sure we return a string even if we don't have a token
  if(!out)
  {
    return '';
  }
  return out;
}

// Save our token to our Storage provider.
// NOTE: For this application we have decided that we will name
// the key for our token 'travlr-token'
public saveToken(token: string): void {
  this.storage.setItem('travlr-token', token);
}

// Logout of our application and remove the JWT from Storage
public logout(): void {
  this.storage.removeItem('travlr-token');
}
```

Next, we build the accessor and mutator methods for moving data in and out of the local storage through our Storage provider. We have one additional method here, ***logout()***, that will clear the local storage in the event that the user logs out. This will necessitate the user reestablishing credentials by logging in again if they wish to continue using the application.

```
// Boolean to determine if we are logged in and the token is
// still valid. Even if we have a token we will still have to
// reauthenticate if the token has expired
public isLoggedIn(): boolean {
  const token: string = this.getToken();
  if (token) {
    const payload = JSON.parse(atob(token.split('.')[1]));
```

```
    return payload.exp > (Date.now() / 1000);
  } else {
    return false;
  }
}

// Retrieve the current user. This function should only be called
// after the calling method has checked to make sure that the user
// isLoggedIn.
public getCurrentUser(): User {
  const token: string = this.getToken();
  const { email, name } = JSON.parse(atob(token.split('.')[1]));
  return { email, name } as User;
}
```

We build a Boolean method to validate whether the user is currently logged in and, if so, if their token is still valid. This is beneficial as we can test whether we need to login again before forcing the user down that path. Specifically, the ***isLoggedIn()*** method should always be called prior to the ***getCurrentUser()*** method in order to avoid an error condition that would arise if there is no data to retrieve.

```
// Login method that leverages the login method in tripDataService
// Because that method returns an observable, we subscribe to the
// result and only process when the Observable condition is satisfied
// Uncomment the two console.log messages for additional debugging
// information.
public login(user: User, passwd: string) : void {
  this.tripDataService.login(user,passwd)
    .subscribe({
      next: (value: any) => {
        if(value)
        {
          console.log(value);
          this.authResp = value;
          this.saveToken(this.authResp.token);
        }
      },
      error: (error: any) => {
        console.log('Error: ' + error);
      }
    })
}

// Register method that leverages the register method in
// tripDataService
```

```
// Because that method returns an observable, we subscribe to the
// result and only process when the Observable condition is satisfied
// Uncomment the two console.log messages for additional debugging
// information. Please Note: This method is nearly identical to the
// login method because the behavior of the API logs a new user in
// immediately upon registration
public register(user: User, passwd: string) : void {
  this.tripDataService.register(user,passwd)
    .subscribe({
      next: (value: any) => {
        if(value)
        {
          console.log(value);
          this.authResp = value;
          this.saveToken(this.authResp.token);
        }
      },
      error: (error: any) => {
        console.log('Error: ' + error);
      }
    })
}
```

These final two methods are identical except for the methods they reference in our ***tripDataService***. These deal with user registration and login. Please Note: these methods both require two parameters, the User and a password. The password is not part of the User object and is expected to be provided by a method or methods external to this service calling these functions.

3. Now that we have built the ***AuthenticationService***, we need to adjust our ***TripDataService*** to provide the additional endpoints for ***login*** and ***register*** that we are relying on in our new ***AuthenticationService***. We need to start by adding three imports to the top of our ***trip-data.service.ts*** file.

```
import { User } from '../models/user';
import { AuthResponse } from '../models/auth-response';
import { BROWSER_STORAGE } from '../storage';
```

We import the ***User*** object so that we can handle the two user parameters ***email*** and ***name***. We import the ***AuthResponse*** object because that will be the type of Observable that we return from ***login*** and ***register***, and we import BROWSER_STORAGE so that we have access to our persistent data. We also have to modify the import line for our ***Injectable*** capability to include ***Inject***.

```
import { Inject, Injectable } from '@angular/core';
```

This allows us to update our constructor to inject our Local Storage provider.

```
  constructor(
    private http: HttpClient,
    @Inject(BROWSER_STORAGE) private storage: Storage
    ) {}
```

Additionally, we will add one more variable, *baseUrl* to our class because we are no longer only using the trips endpoint. This will support additional endpoints in future development as well as providing support for our login and register endpoints.

```
  baseUrl = 'http://localhost:3000/api';
```

Next, we are going to add our methods for login and register. Because these are so similar, differing only in the URL path to their respective endpoints, we are going to refactor most of the code into a helper method so that we only must write and test it once. We will call this method ***handleAuthAPICall***.

```
   // Call to our /login endpoint, returns JWT
  login(user: User, passwd: string) : Observable<AuthResponse> {
    // console.log('Inside TripDataService::login');
    return this.handleAuthAPICall('login', user, passwd);
  }

  // Call to our /register endpoint, creates user and returns JWT
  register(user: User, passwd: string) : Observable<AuthResponse> {
    // console.log('Inside TripDataService::register');
    return this.handleAuthAPICall('register', user, passwd);
  }

  // helper method to process both login and register methods
  handleAuthAPICall(endpoint: string, user: User, passwd: string) :
Observable<AuthResponse> {
    // console.log('Inside TripDataService::handleAuthAPICall');
    let formData = {
      name: user.name,
      email: user.email,
      password: passwd
    };

    return this.http.post<AuthResponse>(this.baseUrl + '/' + endpoint,
formData);
  }
```

4. At this point we have added a great deal of code that isn't yet realized in the Angular application – but the application is still functional to display the available trips. Now we need to concern ourselves with adding some visual components and refactoring the interface to provide the login capabilities we have just added. We will begin this process by creating a navigation bar component to provide access to our new login handler.

   ```
   ng generate component navbar
   ```

   We will need to create a template in the generated ***navbar.component.html*** file to provide our navigation bar complete with our logo and controls to display navigation as well as a login link that will toggle between login and logout based on our current state. Please Note: The angular conditional logic for the navbar items for login/logout. Additionally, the login functionality is not yet available as we haven't built and wired up a login component yet.

```
<nav class="navbar navbar-expand navbar-light bg-light">
    <a class="navbar-brand" href="#"><img
src="/assets/images/logo.png"></a>
    <button class="navbar-toggler" type="button" data-toggle="collapse"
data-target="#navbarNavAltMarkup" aria-controls="navbarNavAltMarkup"
aria-expanded="false" aria-label="Toggle navigation">
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarNavAltMarkup">
        <div class="navbar-nav">
            <a class="nav-link active" routerLink="">Trips<span class="
sr-only ">(current)</span></a>
        </div>
    </div>
    <div class="navbar-end ">
        <a class="nav-item" routerLink="login" *ngIf="!isLoggedIn()">
            <span class="has-icon-left">Log In</span>
        </a>
        <a class="nav-item active" (click)="onLogout()"
*ngIf="isLoggedIn()">
            <span class="has-icon-left ">Log Out</span>
        </a>
    </div>
 </nav>
```

   The ***navbar.component.ts*** file is next to adjust. This requires that we add the *OnInit* capability as well as our ***AuthenticationService***. This will make the imports section of the component look like this:

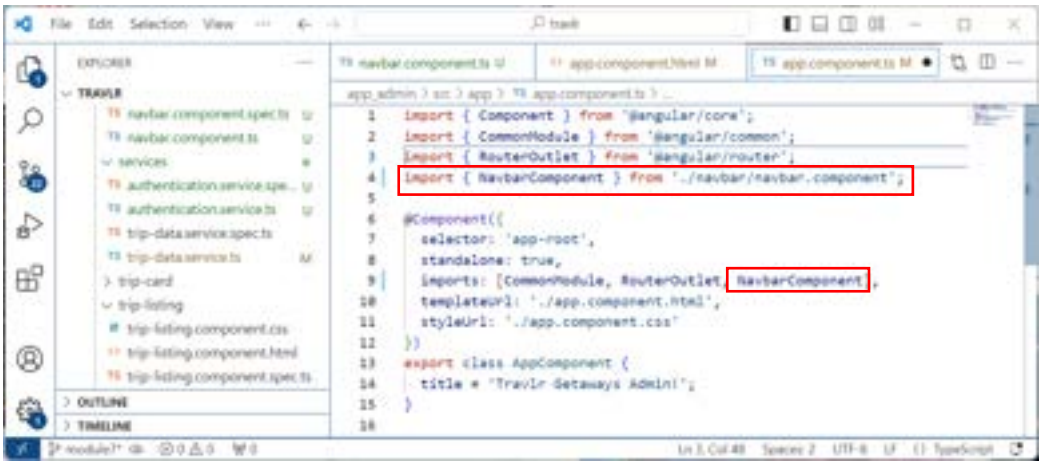
```
import { Component, OnInit } from '@angular/core';
import { CommonModule } from '@angular/common';
```

```
import { RouterLink, RouterLinkActive } from '@angular/router';
import { AuthenticationService } from '../services/authentication.service';
import { RouterModule } from '@angular/router';
```

The body of the class is very straightforward. We need to change it to implement the ***OnInit*** interface, build a constructor to bring in our ***AuthenticationService*** and create two methods: ***isLoggedIn()*** and ***onLogout()***.

```
 @Component({
  selector: 'app-navbar',
  standalone: true,
  imports: [CommonModule, RouterModule],
  templateUrl: './navbar.component.html',
  styleUrl: './navbar.component.css',
})


export class NavbarComponent implements OnInit {

  constructor(
    private authenticationService: AuthenticationService
  ) { }

  ngOnInit() { }

  public isLoggedIn(): boolean {
    return this.authenticationService.isLoggedIn();
  }

  public onLogout(): void {
    return this.authenticationService.logout();
  }
 }
```

5. Now is a good time to pull our new Navigation Bar into our interface and see what it looks like. In order to do that we will make a small edit to our ***app.component.ts*** file to import our new component. We can also take this opportunity to remove our ***TripListingComponent*** import and our ***AddTripComponent*** import as these components are now handled exclusively through the Angular router functionality:



Nothing will happen when you save the file because we must also edit the ***app.component.html*** file and replace the static navigation bar with the Angular selector for our new navbar. This makes our primary html file very minimal as all the processing is being handled by Angular components:

```
<app-navbar></app-navbar>
<div class="container">
 <router-outlet></router-outlet>
</div>
```

6. Now that we have our Navigation bar set and active, we need to add a component so that we can handle the login functionality.

```
ng generate component login
```

This generates our login component and we will want to define a simple HTML template that has three textboxes and a Sign-In button. This will go in the ***login.component.html*** file. There is nothing complicated with this HTML template. Please Note: The data value bindings to the credentials object in our ***login.component.ts*** file.

```
<div class="row">
    <div class="col-12 col-md-8">
        <h2>Login</h2>
        <form (ngSubmit)="onLoginSubmit()">
            <div role="alert" *ngIf="formError" class="alert alert-danger">
                {{ formError }}
            </div>
            <div class="form-group">
                <label for="name">Name</label>
                <input type="text" name="name" placeholder="Enter Name"
                    [(ngModel)]="credentials.name">
```

```
            </div>
            <div class="form-group">
                <label for="email">Email Address</label>
                <input type="email" name="email"
                    placeholder="Enter email address"
                    [(ngModel)]="credentials.email">
            </div>
            <div class="form-group">
                <label for="password">Password</label>
                <input type="password" class="form-control" id="password"
                    name="password" placeholder="e.g 12+ alphanumerics"
                    [(ngModel)]="credentials.password">
            </div>
            <button type="submit" role="button" class="btn btn-primary">
                Sign In!
            </button>
        </form>
    </div>
</div>
```

While the HTML template component of this is not complicated, there is some complexity to the component side of the equation. It starts with the list of imports that is required for this component:

```
import { Component, OnInit } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from "@angular/forms";
import { Router } from '@angular/router';
import { AuthenticationService } from '../services/authentication.service';
import { User } from '../models/user';
```

We need to add the ***FormsModule*** to be able to process the HTML form in our template. We need the ***Router*** module because we would like to redirect the user to another page after they login. We need the ***AuthenticationService*** so that we can process the user login, and we need the ***User*** object so that we have a way of manipulating the ***User*** credentials. We also need to adjust our imports statement for the ***FormsModule***:

```
imports: [CommonModule, FormsModule ],
```

We must define some variables in our ***LoginComponent*** class:

```
public formError: string = '';
submitted = false;

credentials = {
  name: '',
  email: '',
```

```
    password: ''
  }
```

Our constructor will initialize both the ***Router*** capability and the ***AuthenticationService***:

```
  constructor(
    private router: Router,
    private authenticationService: AuthenticationService
  ) { }

  ngOnInit(): void {
  }
```

The first more complex method comes when the user clicks the *Sign-In!* button. This method first checks to make sure that there is data available for all three required parameters. If there is an error with the form, we provide an error message and navigate to the login page and the user can try again. If there is no error with the form and all three fields have data in them, we process the ***doLogin()*** method to advance the authentication process.

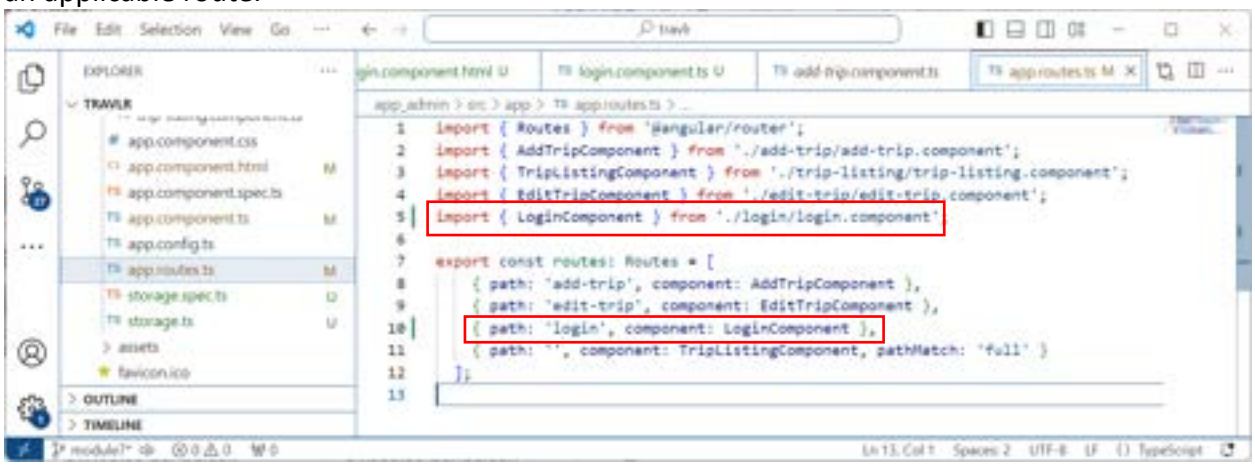
```
  public onLoginSubmit(): void {
    this.formError = '';
    if (!this.credentials.email || !this.credentials.password ||
          !this.credentials.name) {
      this.formError = 'All fields are required, please try again';
      this.router.navigateByUrl('#'); // Return to login page
    } else {
      this.doLogin();
    }
  }
```

Finally, our ***doLogin()*** method does most of the heavy lifting in this component. It takes the form data that we received from the user and builds an ***User*** object that can be used to pass to the login method from the ***AuthenticationService***. It calls the login method, and then checks to see if the user is logged in. If so, it does a re-direct to the trip-list page. If not, it hangs out for 3 seconds and checks again in order to resolve some of the issues related to asynchronous communications over the web.

```
  private doLogin(): void {
    let newUser = {
      name: this.credentials.name,
      email: this.credentials.email
    } as User;

    // console.log('LoginComponent::doLogin');
    // console.log(this.credentials);
```

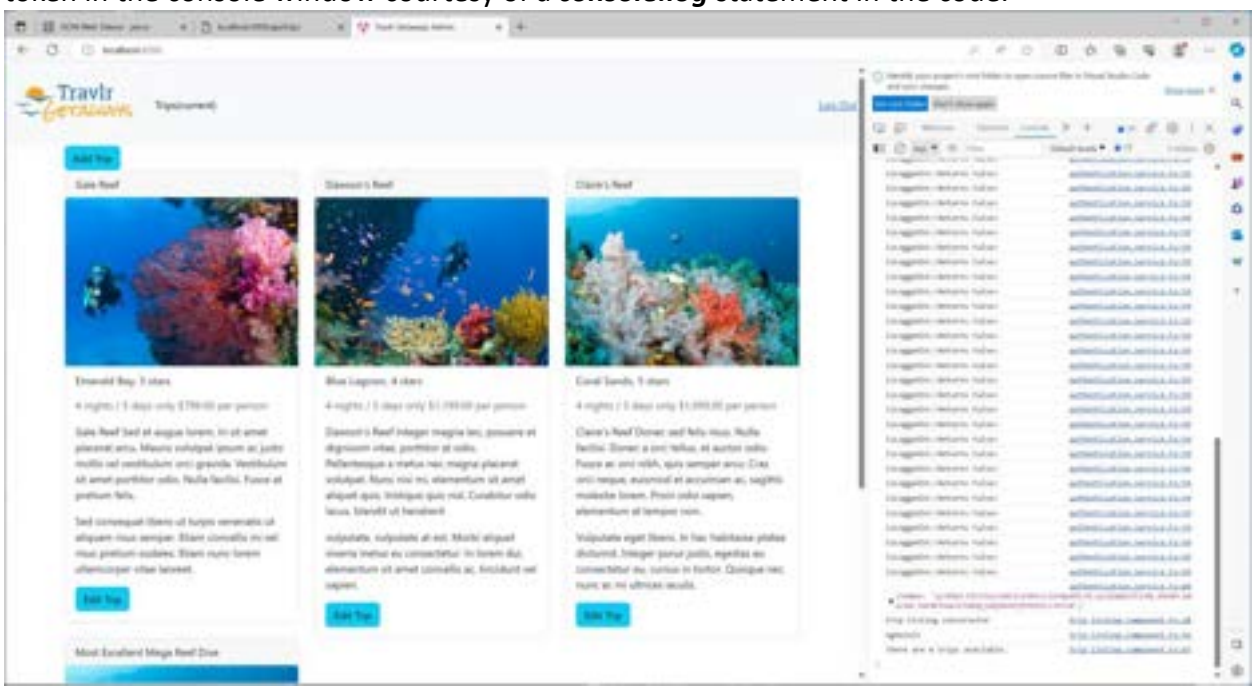
```
        this.authenticationService.login(newUser,
           this.credentials.password);

        if(this.authenticationService.isLoggedIn())
        {
          // console.log('Router::Direct');
          this.router.navigate(['']);
        } else {
          var timer = setTimeout(() => {
          if(this.authenticationService.isLoggedIn())
          {
            // console.log('Router::Pause');
            this.router.navigate(['']);
          }},3000);
        }
      }
```

7. Now that we have our login component, we need to wire up a route for the component. This requires an edit to our ***app.routes.ts*** file to both import our ***login*** component and add an applicable route.



8. At this point, provided you do not have any errors in syntax or logic, you should be able to run your application and login. You can see the 'Log Out' link in the navigation bar and the

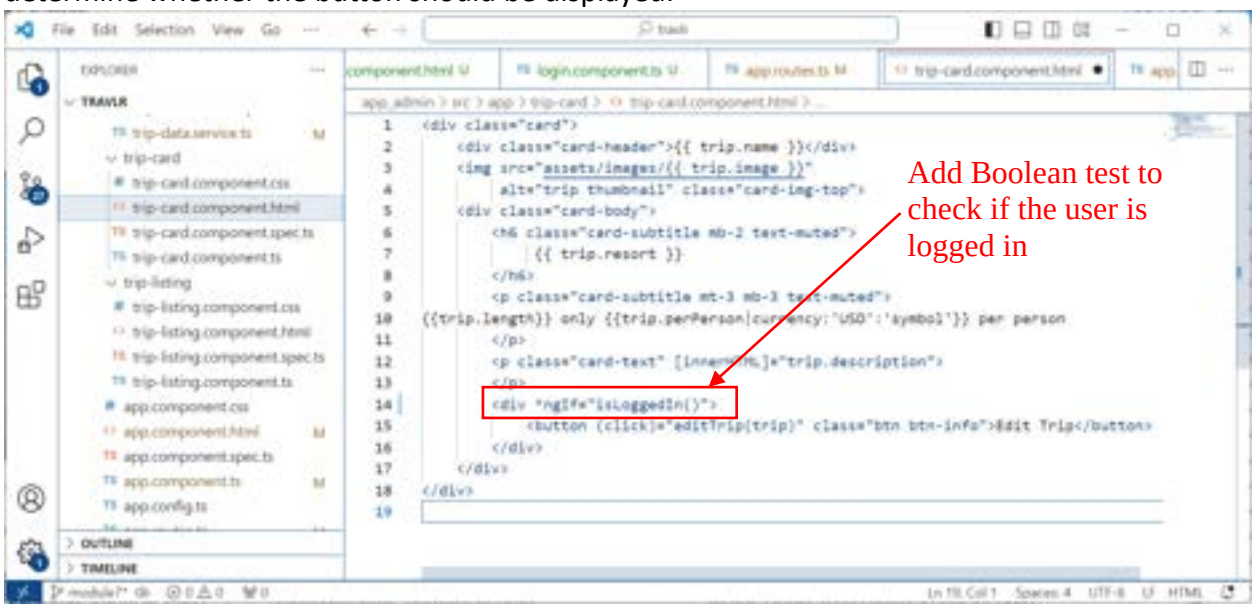
token in the console window courtesy of a ***console.log*** statement in the code.



The login form provided is very generic and includes the name field so that it could be reused as a register form if you chose to implement that on your own. The only real negative to the generic implementation is that while only the email address and password are needed to login, you still have to provide a value in the Name field to actually login. It does not matter what name you provide in that field just that it is not blank. Fixing that logic is a good exercise to try.

9. At this point, you will notice that whether you are logged in or not, the buttons for 'Add Trip' and 'Edit Trip' are still present – even though they will not function properly if you are not logged in. This is something that should be addressed for completeness in your application and is relatively easy to accomplish. With a very small edit to ***trip-card.component.html*** we can add some conditional logic that will take advantage of our ***AuthenticationService*** to

determine whether the button should be displayed.



Editing the single field is all that is needed on the presentation side of the equation, now we need to make a small addition to the ***trip-card.component.ts*** file to add a local function ***isLoggedIn()*** that will delegate to our ***AuthenticationService***. First we add our import for the ***AuthenticationService***:

```
import { AuthenticationService } from '../services/authentication.service';
```

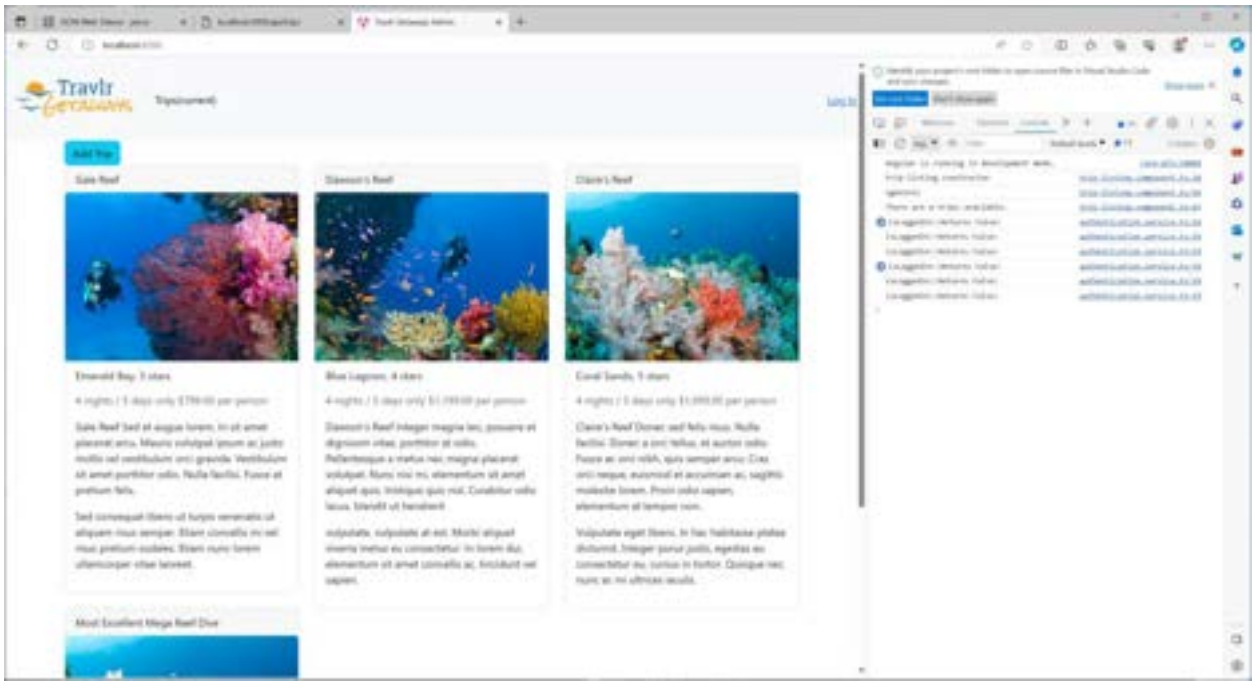
Next we update the constructor to pull the service into our class:

```
  constructor(
    private router: Router,
    private authenticationService: AuthenticationService
    ) {}
```

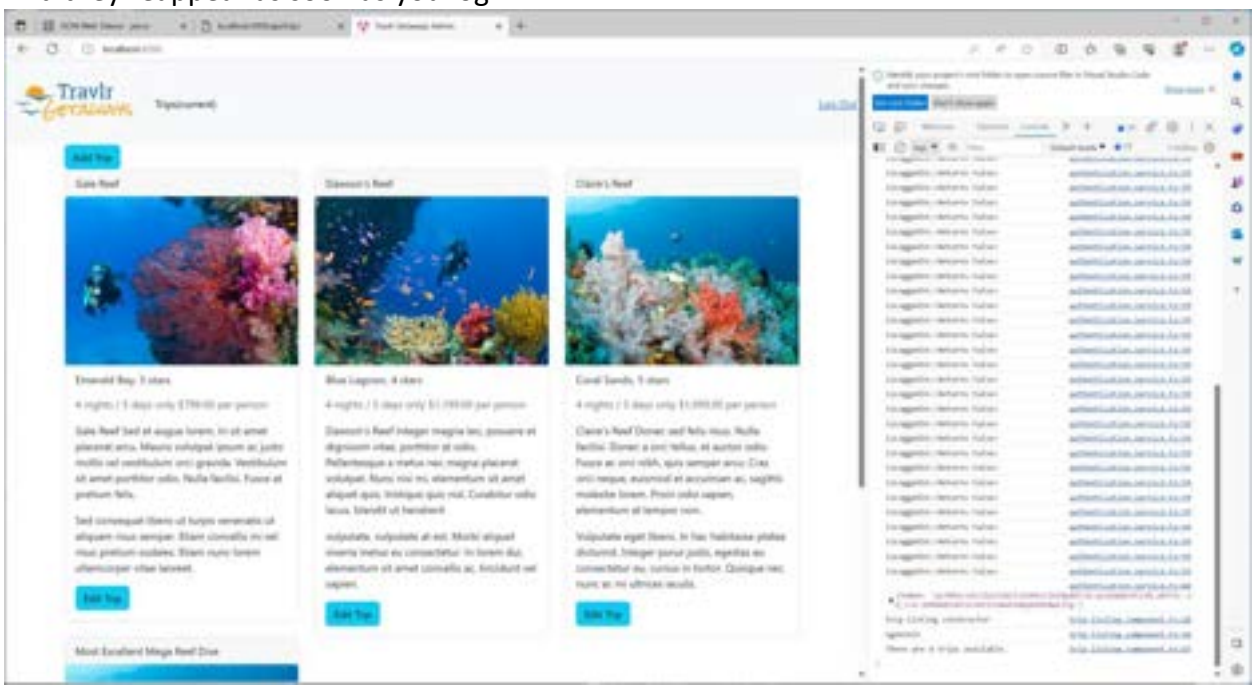
Finally we add a method to delegate the question to our ***AuthenticationService***:

```
  public isLoggedIn()
  {
    return this.authenticationService.isLoggedIn();
  }
```

As soon as we save the files, we see the Angular application update and if we logout, we will see that the edit buttons disappear.

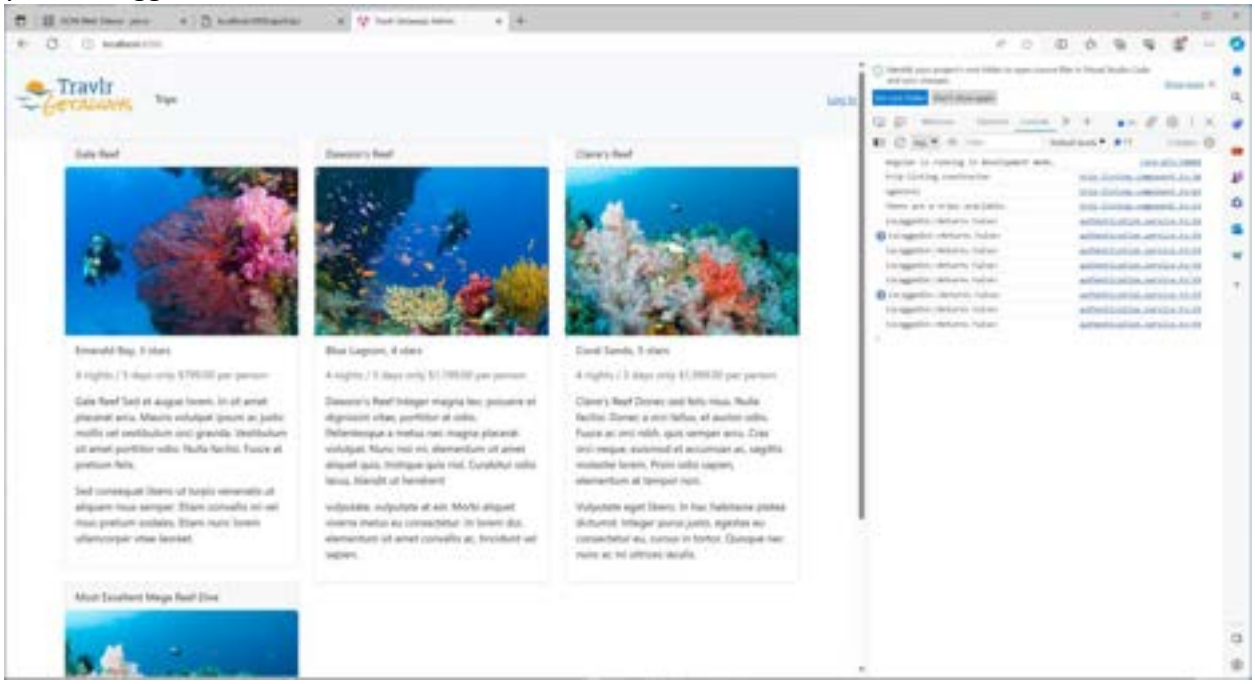


And they reappear as soon as you login:



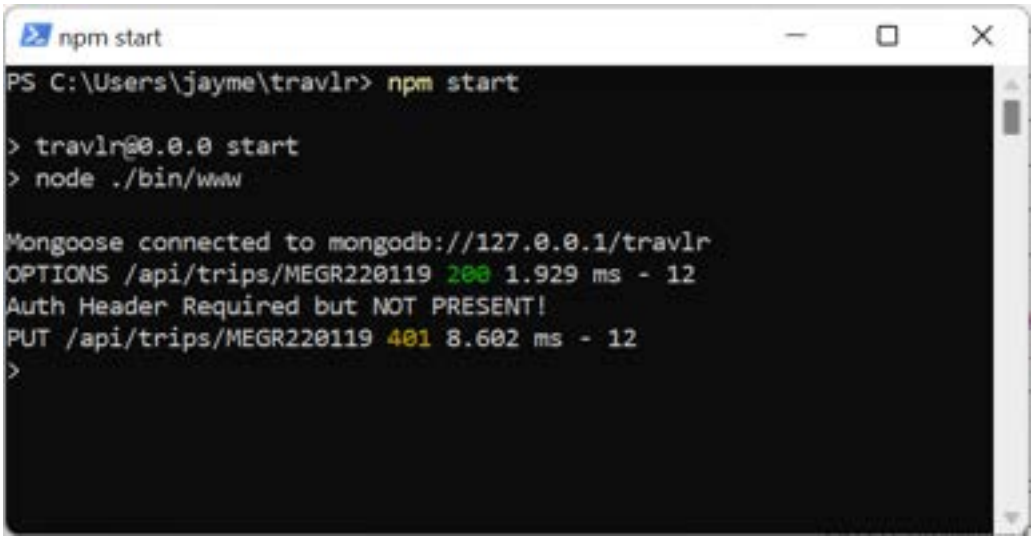
10. To be consistent, we should repeat the editing process for the add-trip component to make the rendering of the 'Add Trip' button conditional as well. Repeat the process detailed in item 9 above with the ***trip-list.component.html*** and ***trip-list.component.ts*** files to accomplish this task. Once your edits are complete, your page should look like this when

you are logged out:



11. At this point we have added the login/logout functionality to the Angular application and made some edits so that the graphical components are displayed sensibly when we are logged out. However, there is something that we have not yet addressed. Even when we are logged in, the 'Add Trip' and 'Edit Trip' buttons will not work.

    While we have gone through many things to make sure that we can login to our application, the one thing that we have not yet done is made sure that our JWT is presented back to the server when we are calling the APIs that make changes. If you were to login to the application, you would see the following error on the console and the edit would not be made:



Now, there are several ways to approach this problem. We could edit each of the calls in our trip-data.service, but that would mean that we would have to make continuous edits to that file if we wanted to make changes and expand our application API in the future. Fortunately, Angular provides a capability that will allow us to remove that type of cross-cutting concern from our application and centralize the code in a single module. To accomplish this, we will create a new type of Angular component, an *Interceptor*. An *Interceptor* is a special module that will intercept data in a pipeline, allow you to perform operations on it, and then return it to the pipeline for further processing. To begin, lets create the module:

```
ng generate interceptor jwt
```

Due to the utility of these components, we are going to want to create a separate directory to store them in so that we can keep our overall application organized. For this purpose, we will create the directory ***/app_admin/src/app/utils*** and relocate the ***jwt.interceptor*** files there. We will start by addressing the imports that we will need for this component:

```
import { Injectable,Provider } from '@angular/core';
import { HttpRequest, HttpHandler, HttpEvent } from '@angular/common/http';
import { HttpInterceptor, HTTP_INTERCEPTORS } from '@angular/common/http';
import { Observable } from 'rxjs';
import { AuthenticationService } from '../services/authentication.service';
```

We use Injectable and Provider to provide the capability to change access and modify data in a pipeline. We utilize the five imports from the http package to interact with the HTML page from a client perspective. Observable is required as RESTful API and http interaction is asynchronous, and we use the ***AuthenticationService*** to handle interaction with our JWTs.

```
@Injectable()
export class JwtInterceptor implements HttpInterceptor {
```

We annotate our class with the ***Injectable*** tag which allows the class to function as a provider, while implementing the HttpInterceptor interface allows us to step into the middle of an http pipeline.

```
constructor(
  private authenticationService: AuthenticationService
) {}
```

Our constructor is used to bring our ***AuthenticationService*** into scope for this class.

```
intercept(request: HttpRequest<any>, next: HttpHandler):
    Observable<HttpEvent<any>> {
  var isAuthAPI: boolean;
```

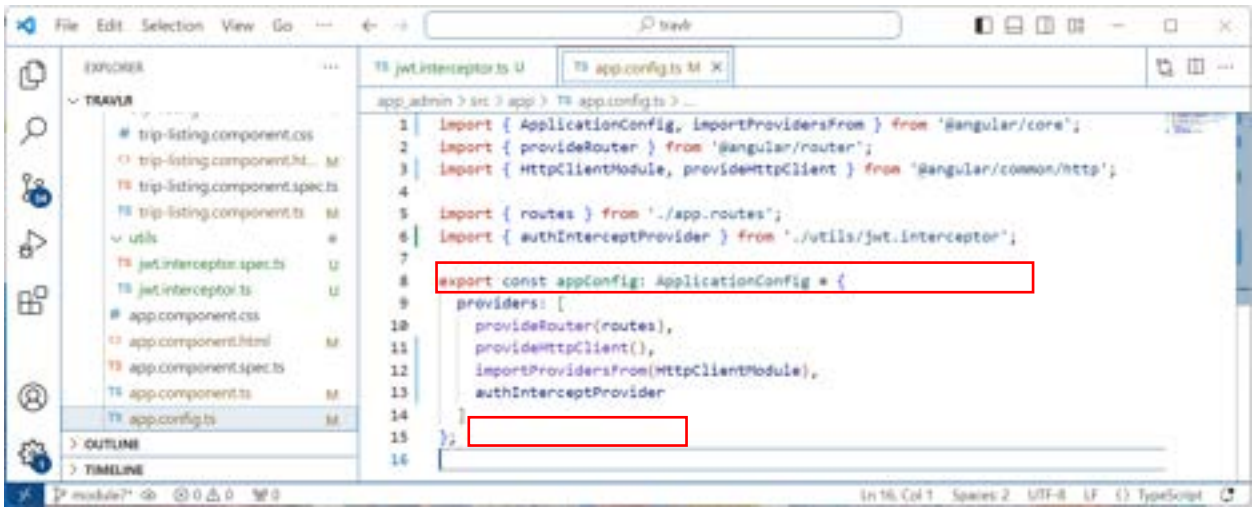
```
    // console.log('Interceptor::URL' + request.url);
    if(request.url.startsWith('login') ||
        request.url.startsWith('register')) {
      isAuthAPI = true;
    }
    else {
      isAuthAPI = false;
    }

    if(this.authenticationService.isLoggedIn() && !isAuthAPI) {
      let token = this.authenticationService.getToken();
      // console.log(token);
      const authReq = request.clone({
        setHeaders: {
          Authorization: `Bearer ${token}`
        }
      });
      return next.handle(authReq);
    }
    return next.handle(request);
  }
}
```

The ***intercept*** method is what provides the heavy lifting in this module. The first thing we do is we define a Boolean flag to identify whether the URL belongs to one of the two AuthAPI URLs that we do not want to try and intercept. We then grab the JWT from the ***AuthenticationService*** clone the http request, inject the new Authorization header, and then handle the cloned request.

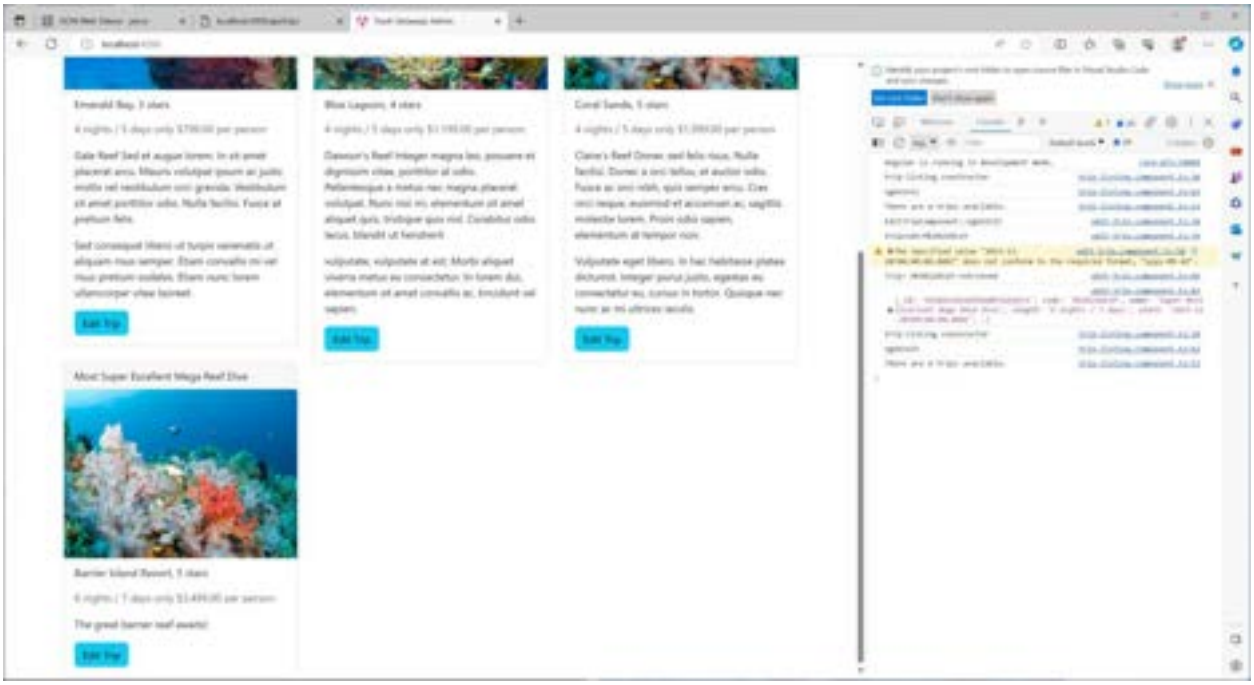
Finally, we export an unique provider from our Interceptor module that we will then pull into our ***app.config.ts*** file to wire the Interceptor into our pipelines.

```
export const authInterceptProvider: Provider =
  { provide: HTTP_INTERCEPTORS,
      useClass: JwtInterceptor, multi: true };
```

12. The last change that needs to be made to activate our Interceptor and therefore enable secure transactions for all our APIs that require authentication. This will be made in the ***app.config.ts*** file. Be sure to add the HttpClientModule import and the importProvidersFrom lines as well. They will get a line through them and show as deprecated, but they are still needed in the project

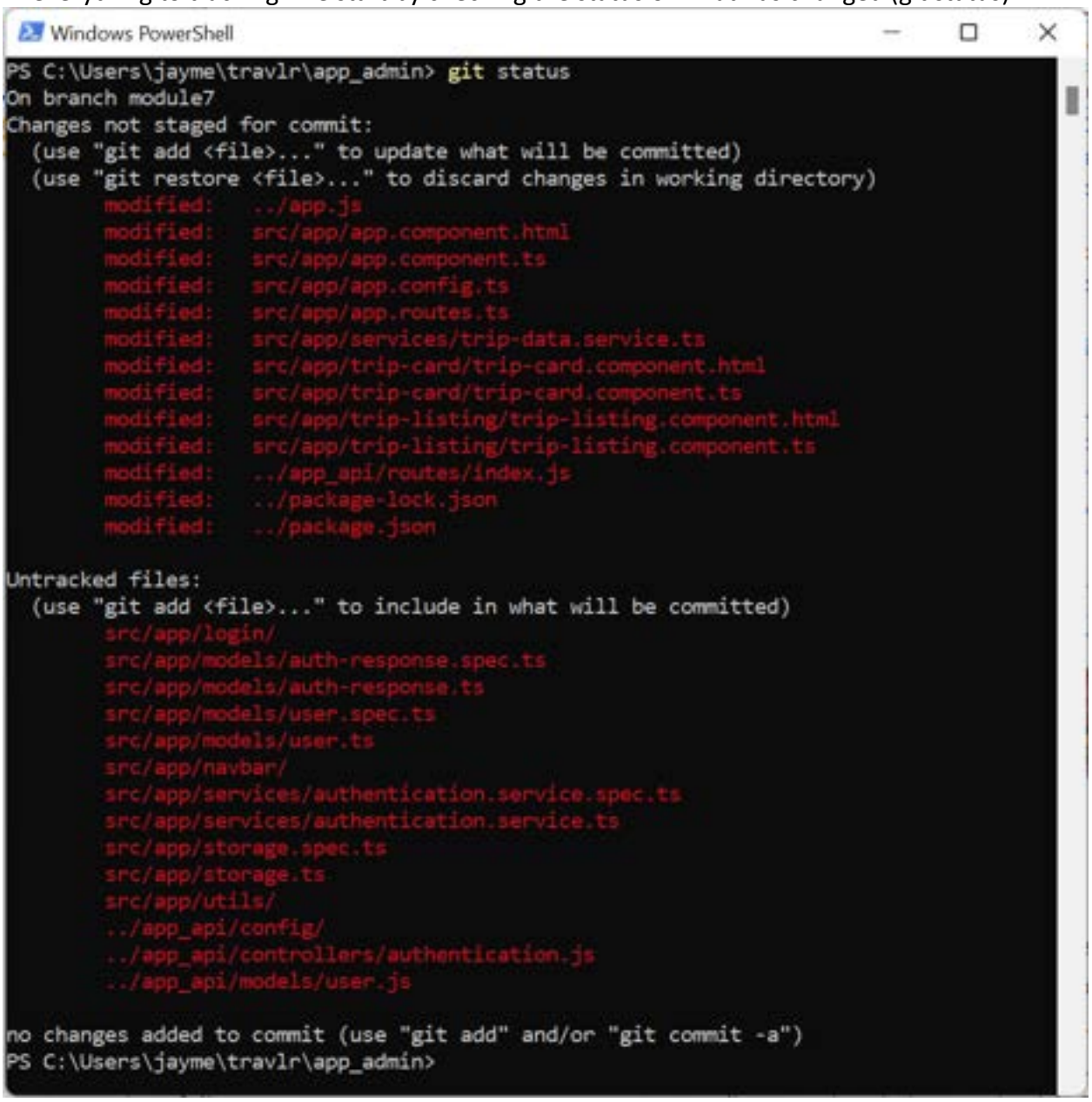


And now, going back to the browser, you can edit an existing trip. In this case, I changed the name for the fourth trip:



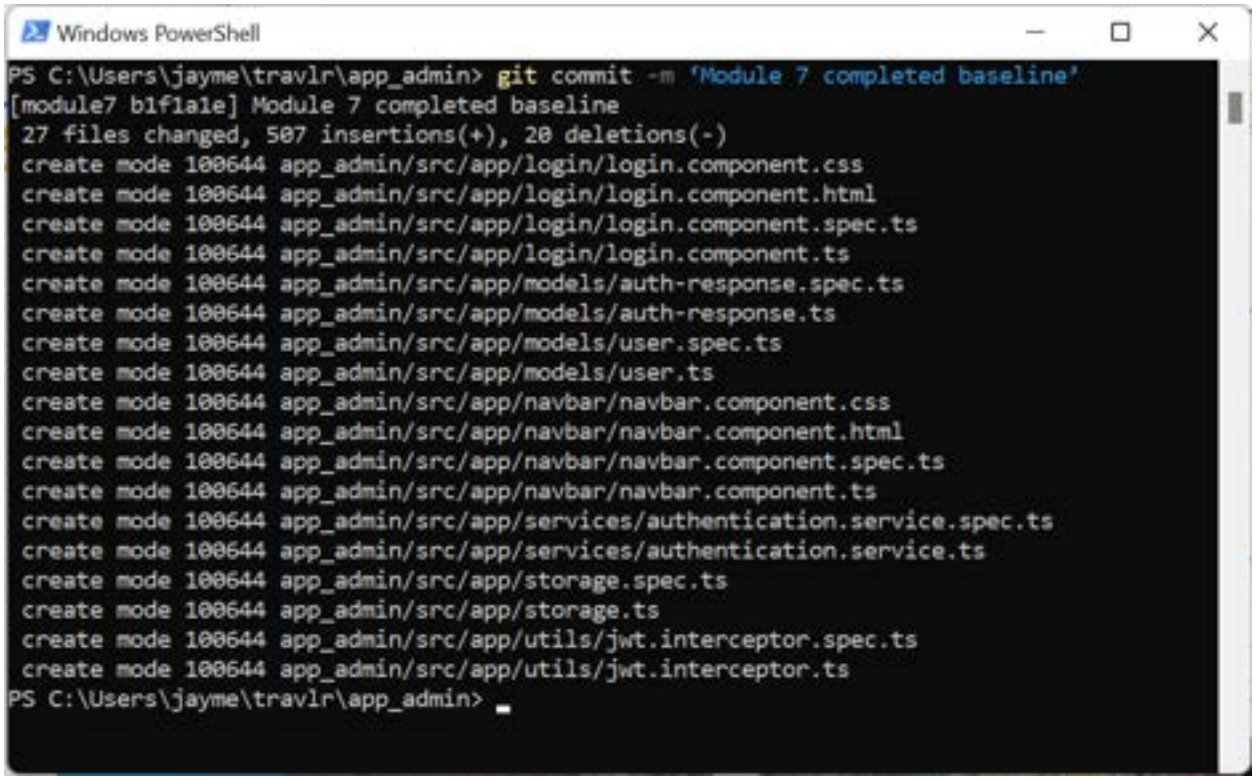
Finalizing Module 7

1. Now that we have completed Module 7, we go back to git and make sure that we add everything to tracking. We start by checking the status of what has changed (git status):

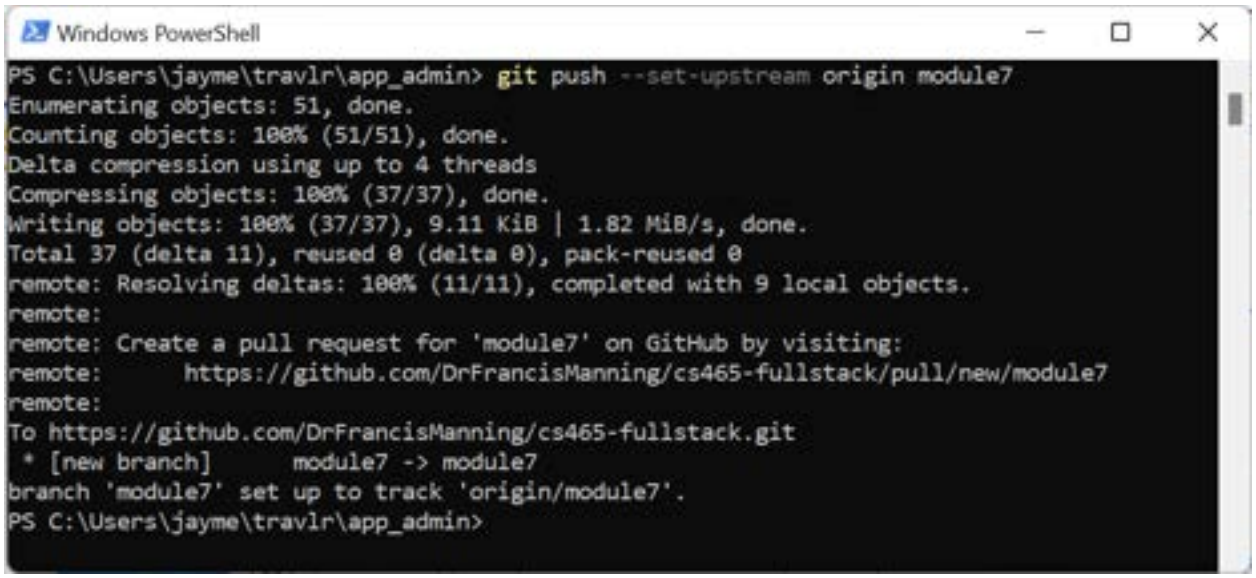


2. Then we add all of those changes into tracking (`git add .`):

3. Now we commit the changes (`git commit -m 'Module 7 completed baseline'`):



4. We push the changes back to GitHub for safekeeping (`git push --set-upstream origin module7`):


Windows PowerShell
PS C:\Users\jayme\travlr\app_admin> git push --set-upstream origin module7
Enumerating objects: 51, done.
Counting objects: 100% (51/51), done.
Delta compression using up to 4 threads
Compressing objects: 100% (37/37), done.
Writing objects: 100% (37/37), 9.11 KiB | 1.82 MiB/s, done.
Total 37 (delta 11), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (11/11), completed with 9 local objects.
remote:
remote: Create a pull request for 'module7' on GitHub by visiting:
remote: https://github.com/DrFrancisManning/cs465-fullstack/pull/new/module7
remote:
To https://github.com/DrFrancisManning/cs465-fullstack.git
* [new branch] module7 -> module7
branch 'module7' set up to track 'origin/module7'.
PS C:\Users\jayme\travlr\app_admin>