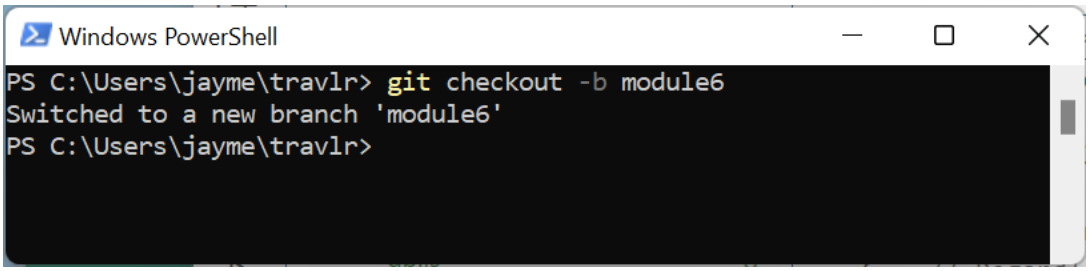
Module 6: SPA (Single Page Application)

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.

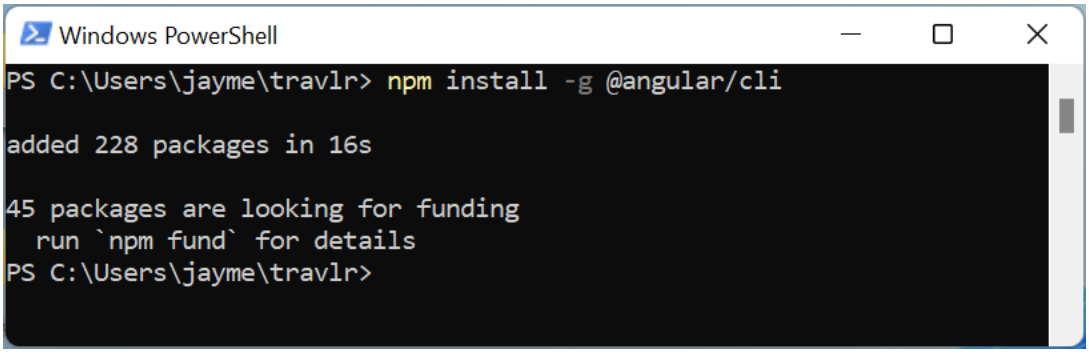
The first thing that needs to be accomplished in order to begin this development process is to install Angular and create our Admin site.

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.

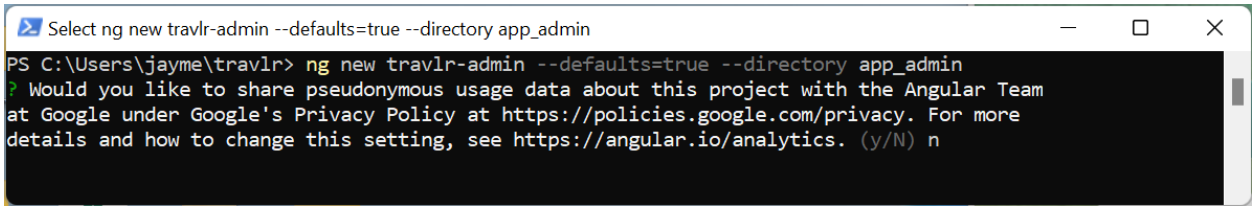
When installing a package beneath @angular, you should have confidence that the package was developed and published by the Angular development team.

```
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.

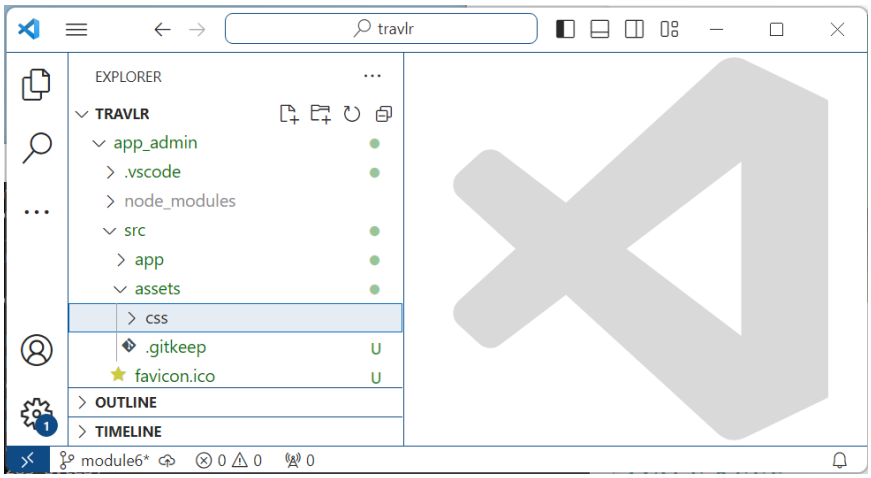
```
ng new travlr-admin --defaults=true --directory app_admin
```



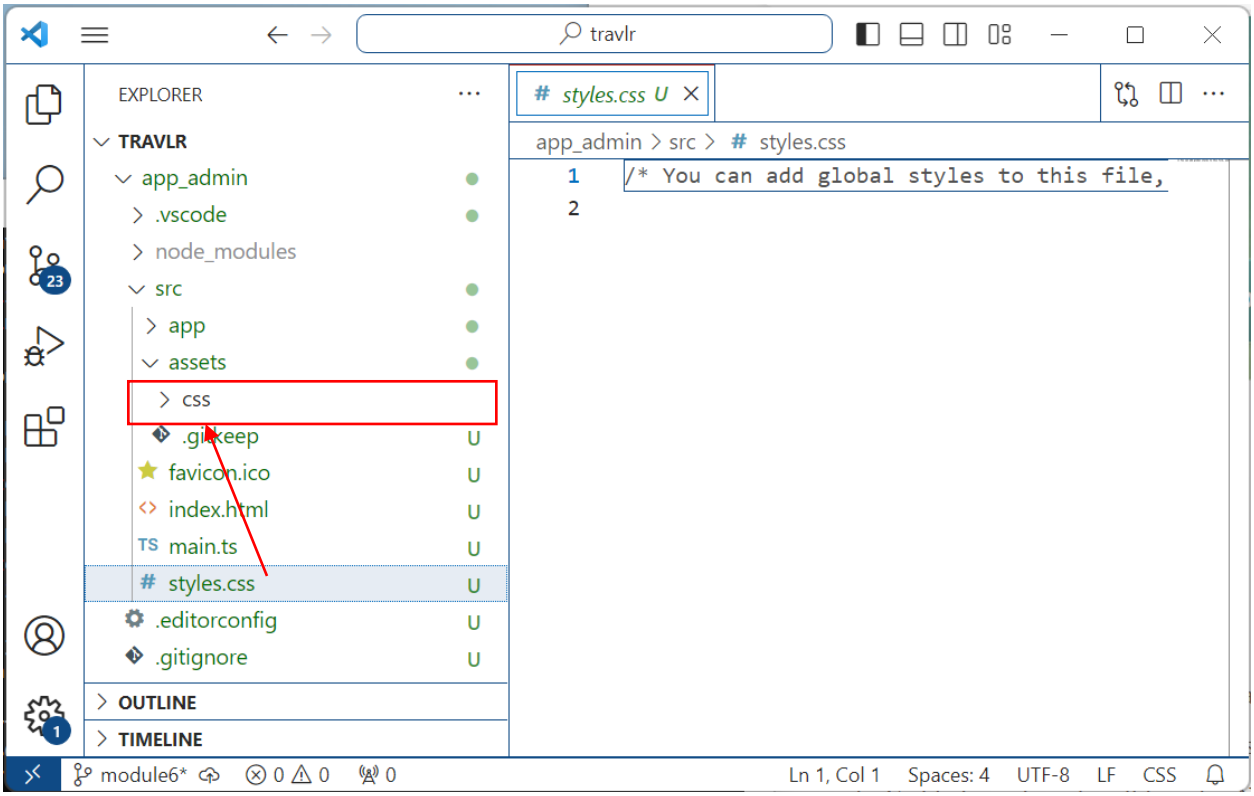
You will be asked if you want to share usage data for what you are building with Google for the purpose of analytics. It is safe to answer this question with an ‘n’ for no.

```
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>
```

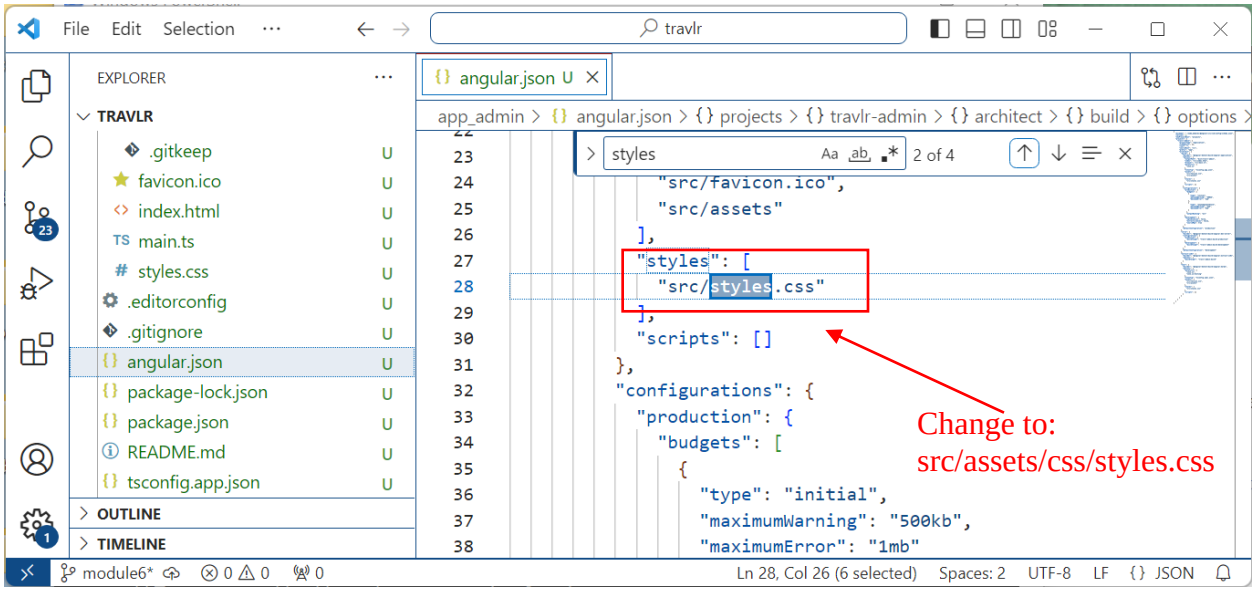
3. Now that the ***Angular*** app has been created in our ***travlr*** project tree, there are two structural changes we need to make to conform to industry best practices for application organization. We will first open our project in VS Code and see the new ***app_admin*** folder. We will want to descend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



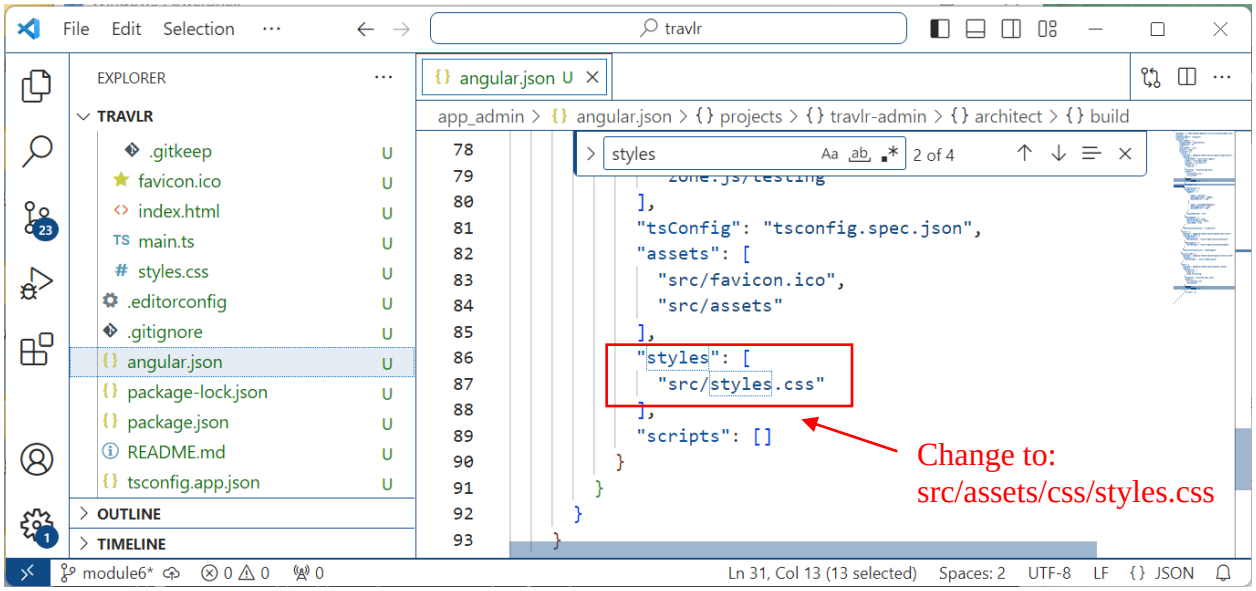
4. Once we have created the ***css*** folder, we will drag the **style.css** file from ***app_admin/src*** into the ***css*** folder. This will help to provide organizational consistency – but it will also require some additional editing of the default application.



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**.



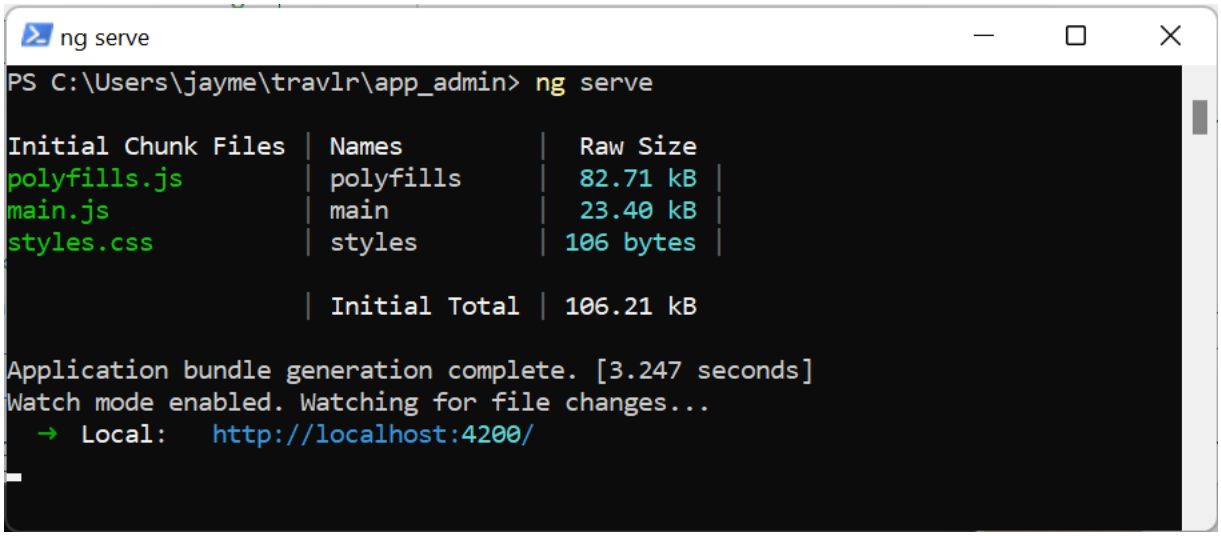
and



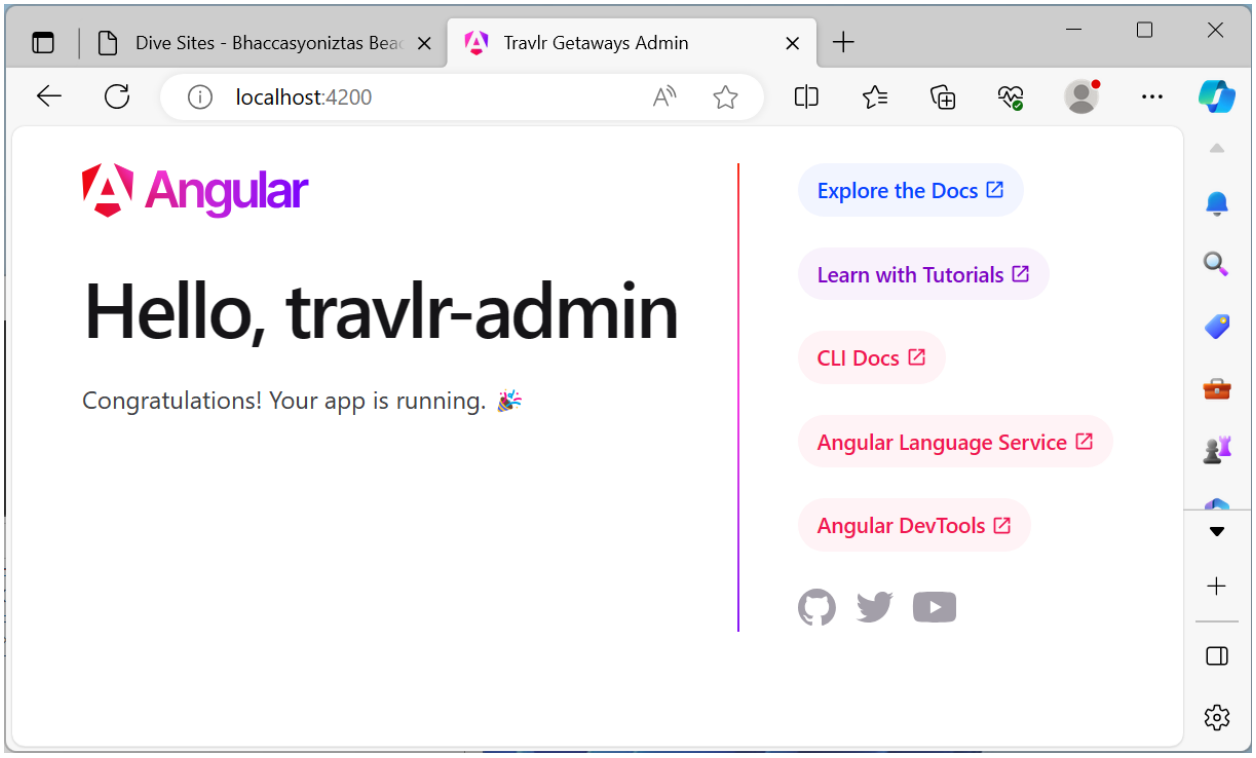
6. We are also going to want to replace the title and add a loading message. This will require an edit of the ***app_admin/src/index.html*** file.

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:

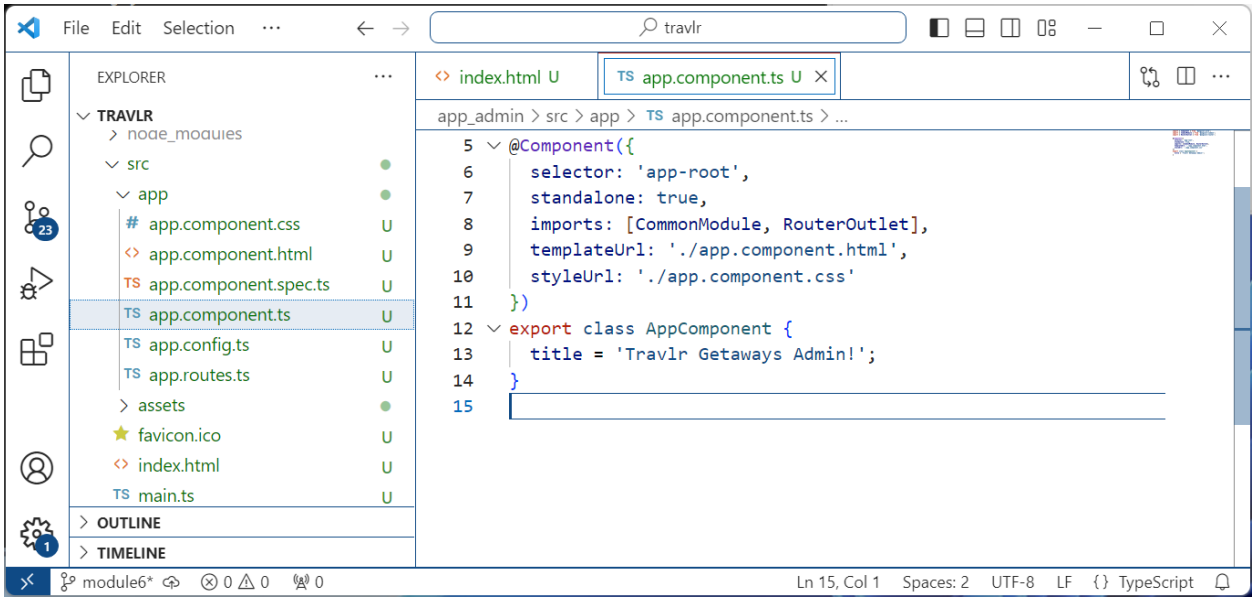
```
ng serve
```



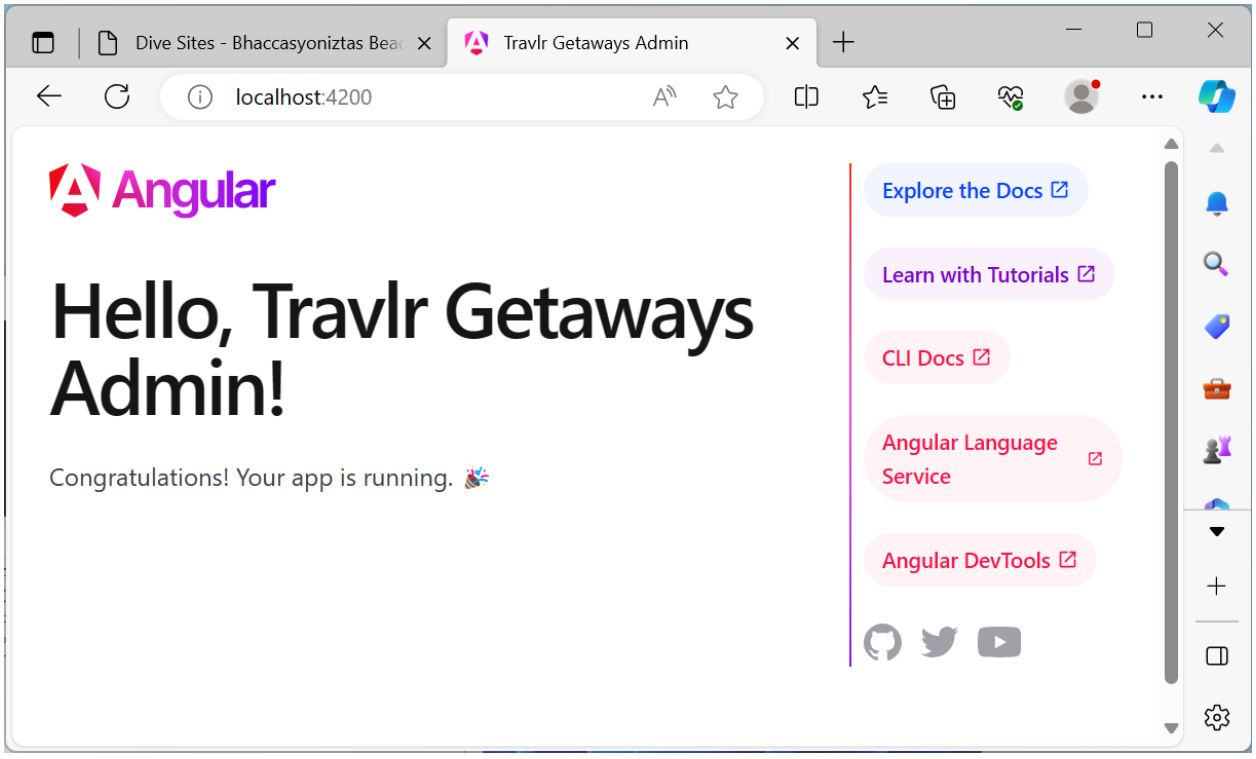
8. Now the application is running and we can attach to it. Open up a new tab in your browser and connect to the app at: http://localhost:4200 .



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:



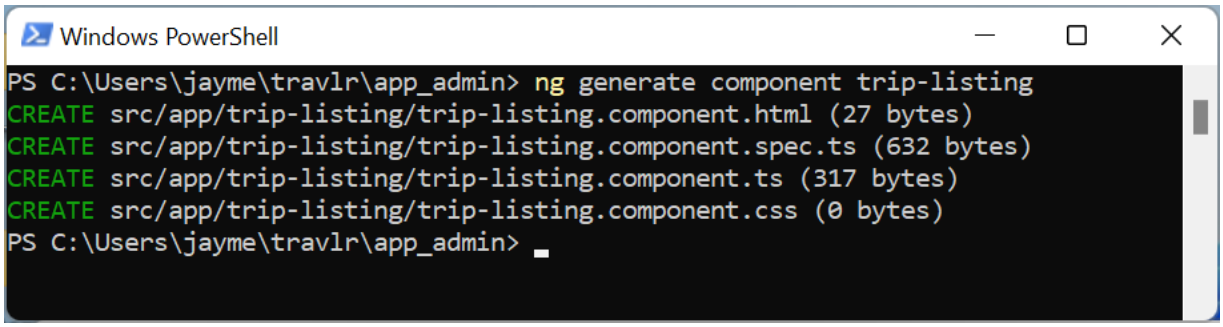
At this point, you have successfully created the basic Angular application that will support the administrative functionality for the Travlr Getaways application!

Create Trip Listing Component

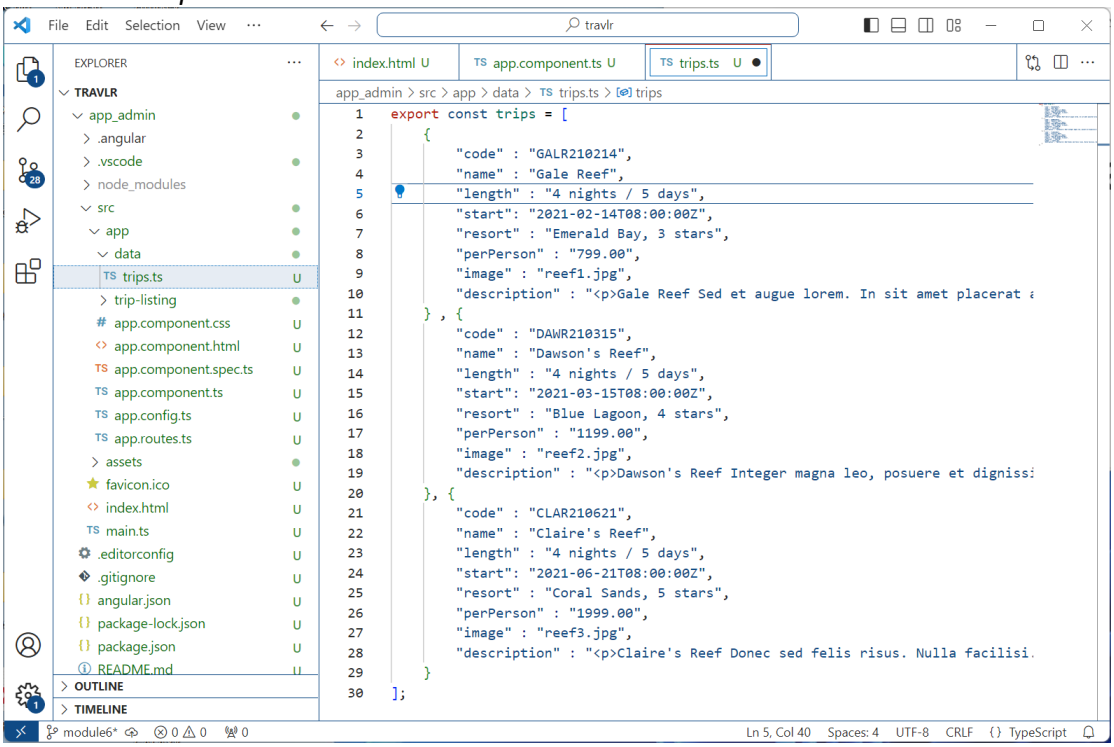
Now that we have the initial application constructed, we need to do something so that it will display useful data. To do this, we are going to need to create an Angular component. This component will be used to display the list of trips.

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.

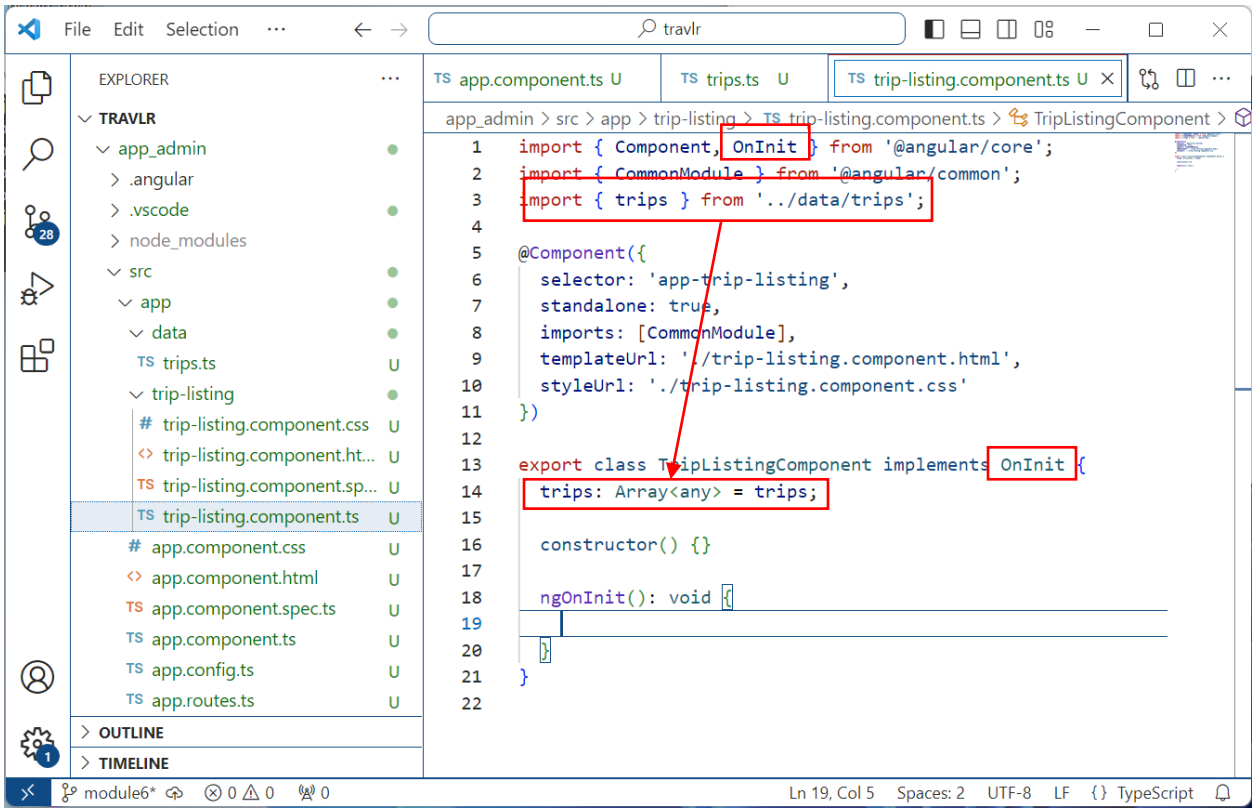
```
ng generate component trip-listing
```



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*.

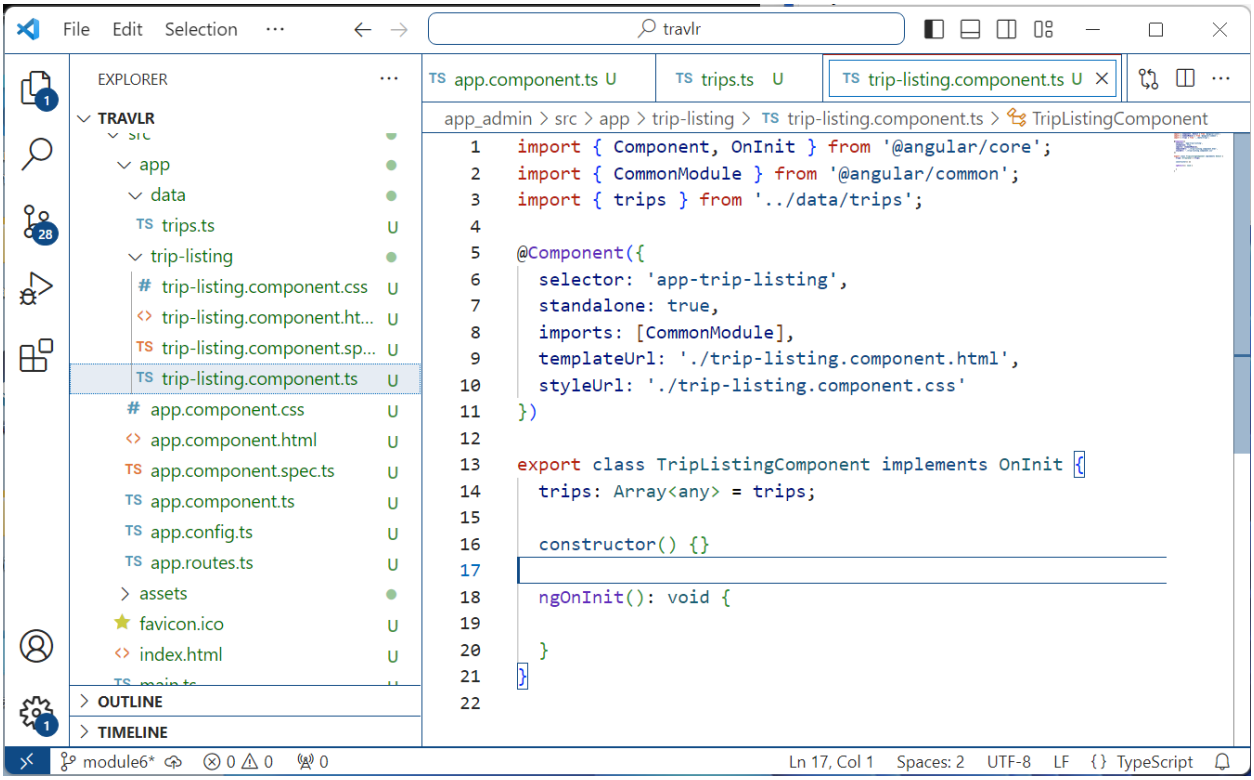


3. Edit the ***trip-listing.component.ts*** file to add an import of the new ***trips.ts*** file, and define an appropriate class variable within the *TripListingComponent* class to contain the data. We will also need to add *OnInit* to the imports for the class and setup the basic structure for our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

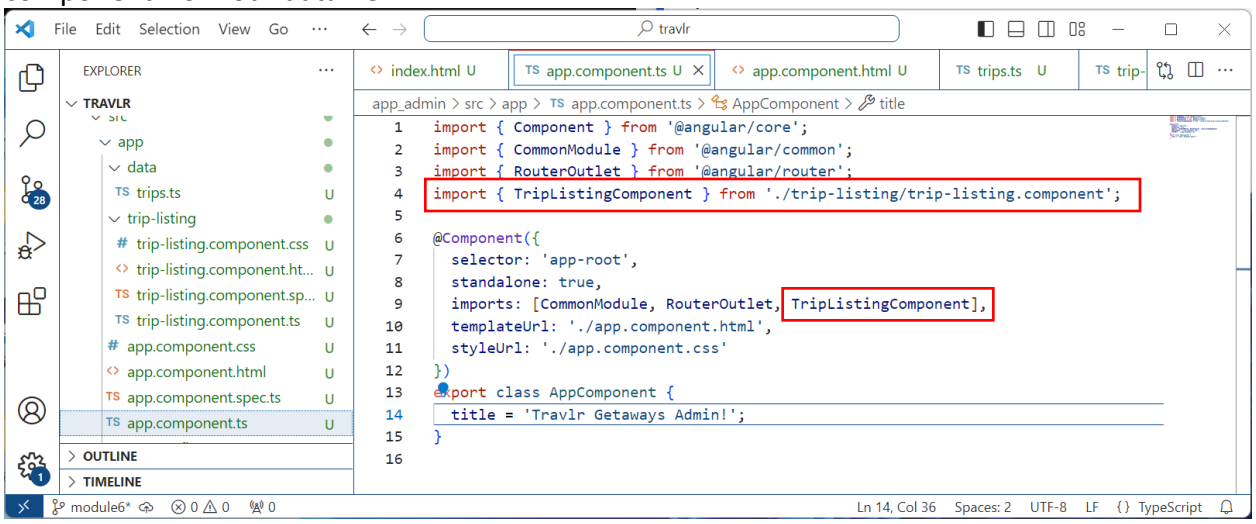


The next step will require editing the ***trip-listing.component.html*** file and replace the initial contents: trip-listing works! with <pre>{ { trips | json } }</pre>

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.

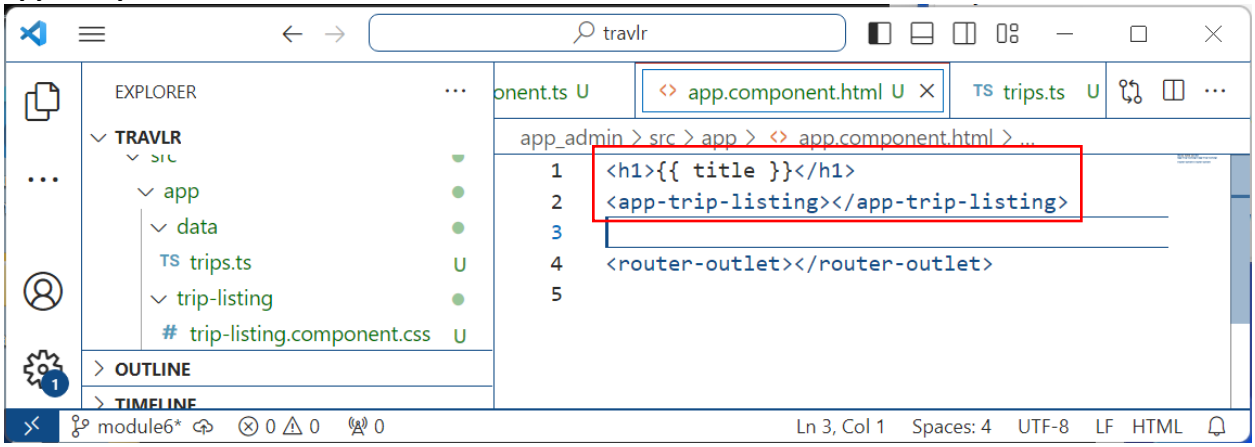


4. In the next step, we need to make one more edit to the ***app.component.ts*** file and pull in our *TripListingComponent*. This will allow us to access the data that we have pulled into that component from our datafile.

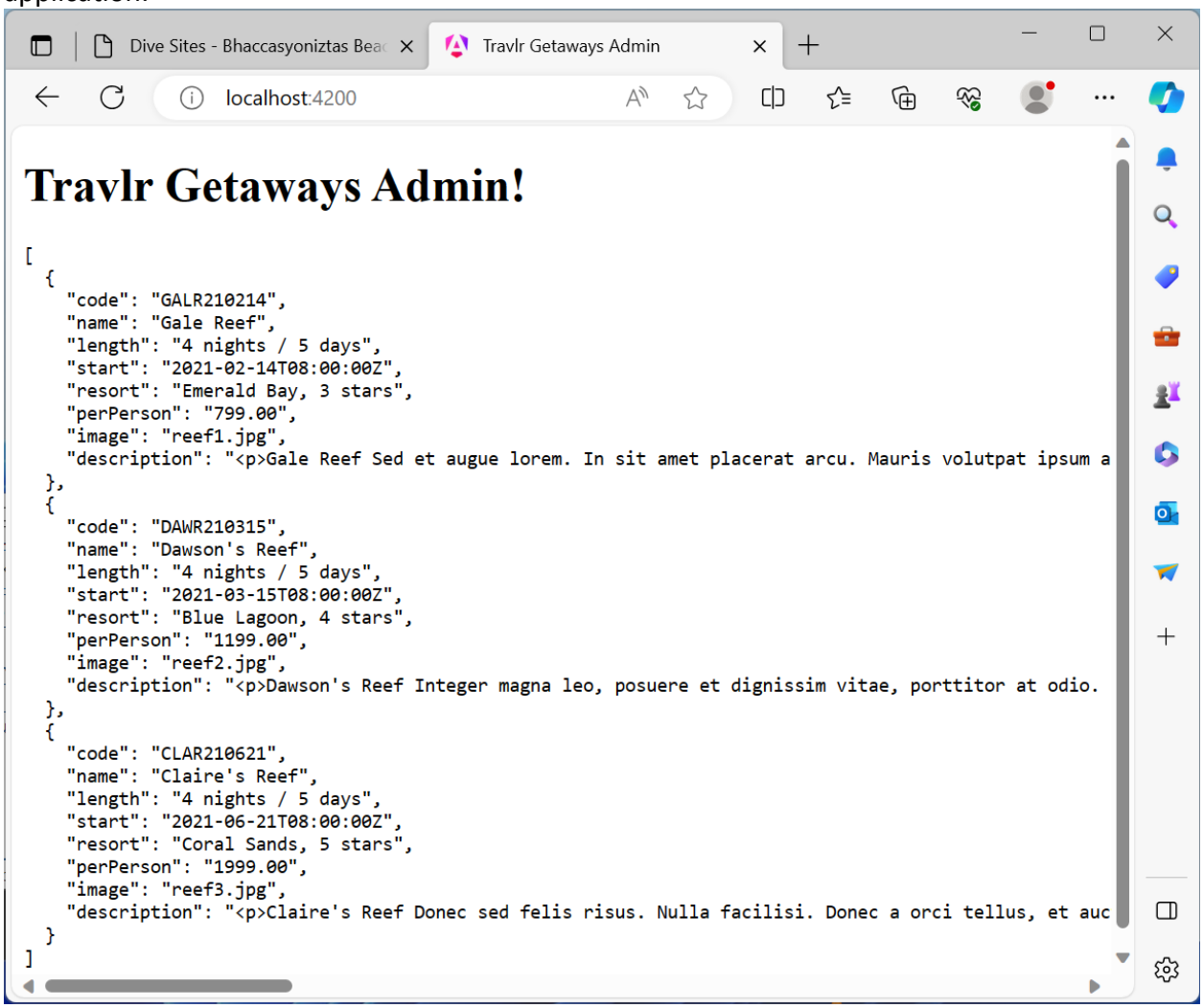


5. And with one more small edit, we can dynamically pull both the *title* variable from the *AppComponent* class and the *trips* array via the *app-trip-listing* selector from the *TripListingComponent* class and render them in our application. This edit will be made in the

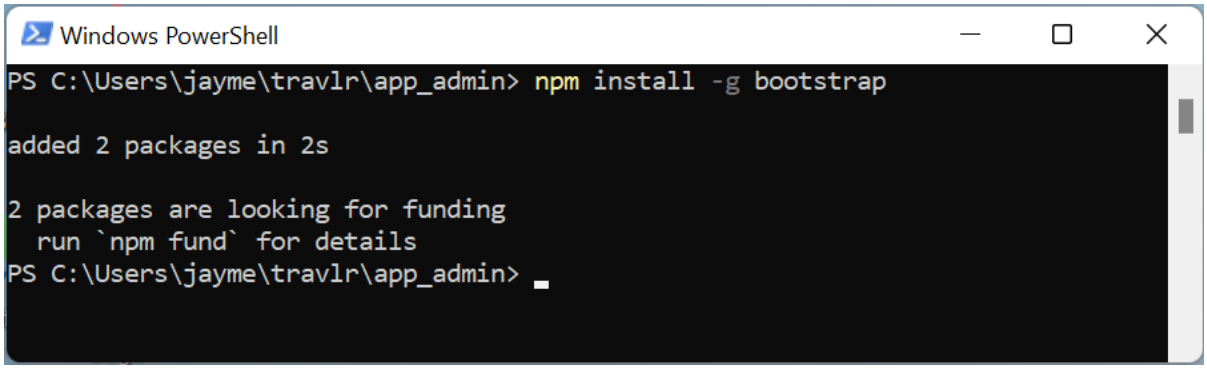
app.component.html file.



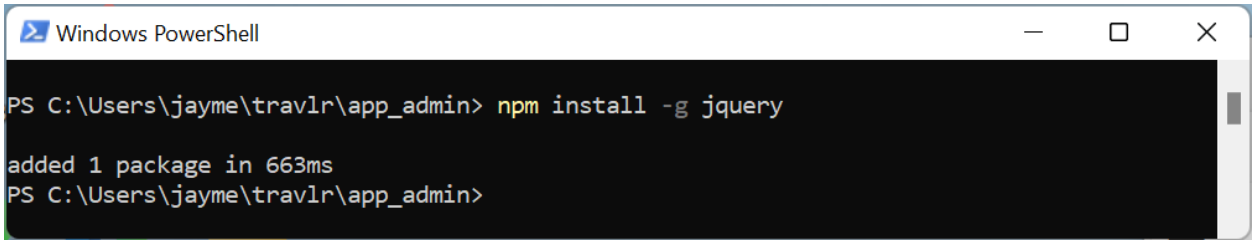
6. Once this file is saved, the application should automatically refresh and show the JSON in application.



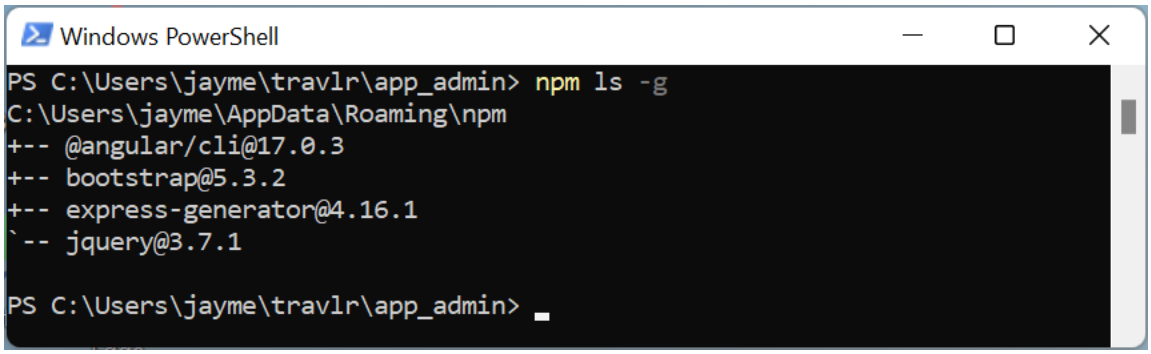
7. Now that we can see that we have access to the data, we need to determine how we can format the data so that it is usable for a front-end application. To handle this, we are going to involve the Bootstrap CSS framework. This requires an additional install in our Node.JS environment.



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):

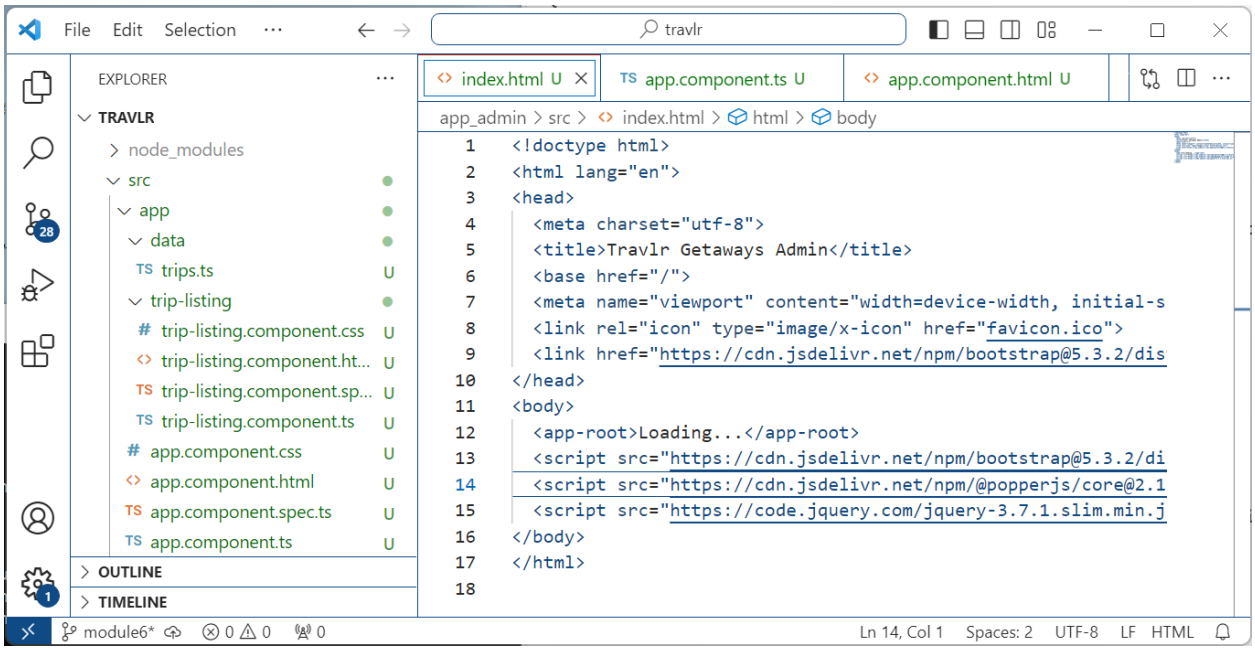
```
<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/boot
strap.bundle.min.js" integrity="sha384-
C6RzsynM9kWDrMNeT87bh95OGNyZPhcTNXj1NW7RuBCsyN/o0jlpcV8Qyq46cD
fL" crossorigin="anonymous"></script>
```

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:

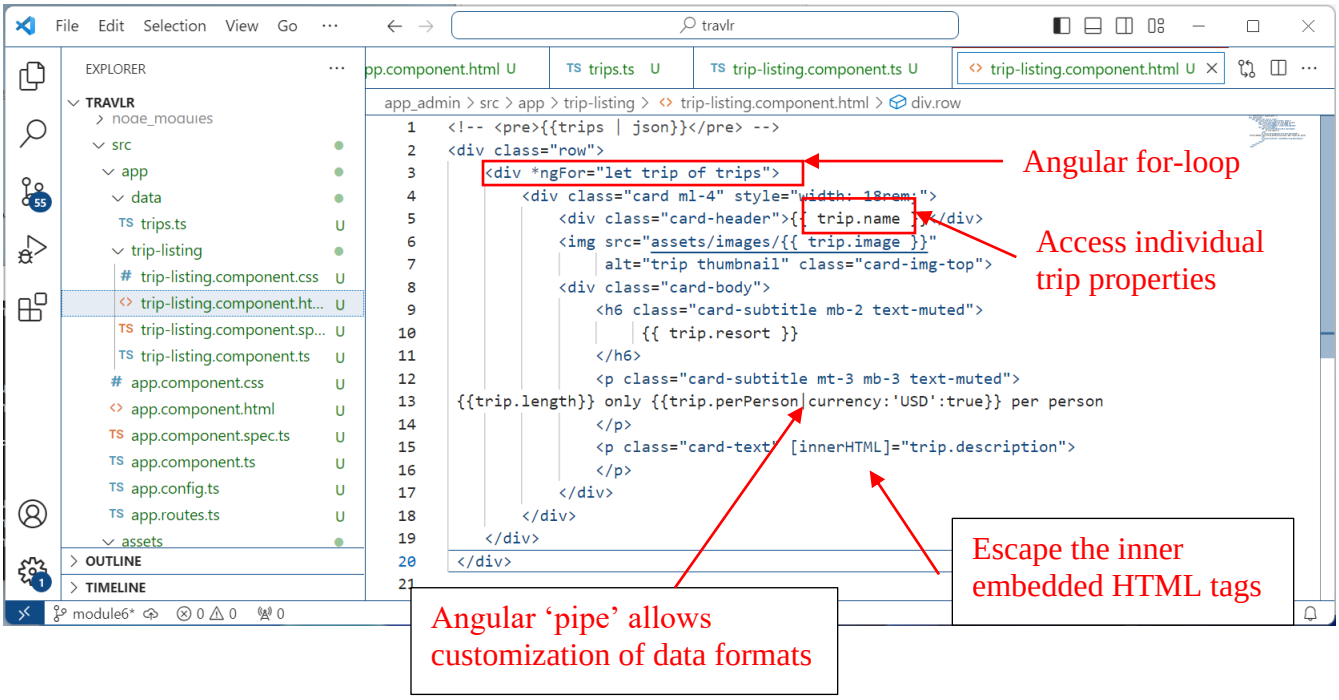
```
"assets": ["src/favicon.ico", "src/assets"],
```

    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***.

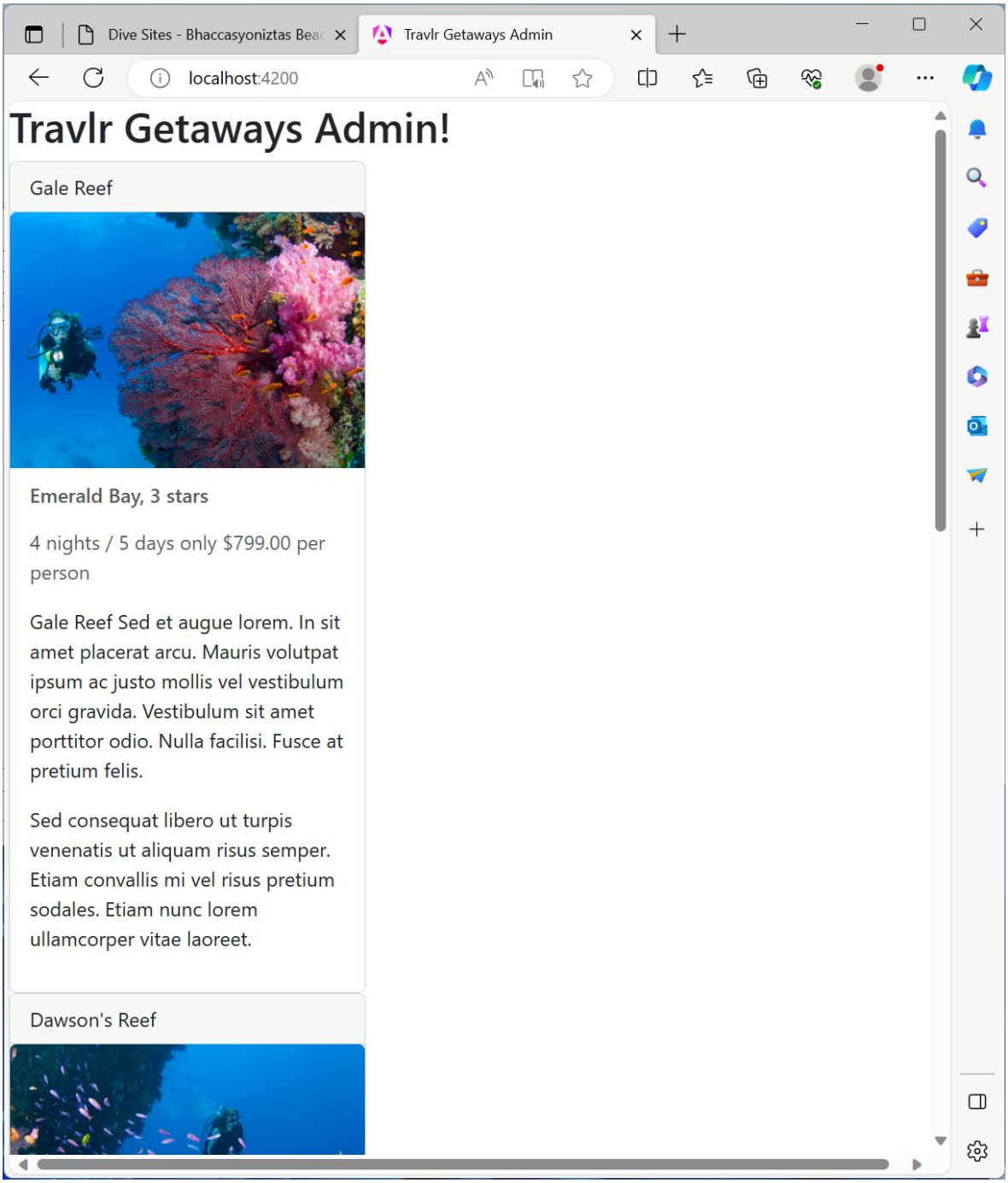
    The contents of the ***trip-listing.component.html*** file will be replaced with the following code:

```
<!-- <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>
```

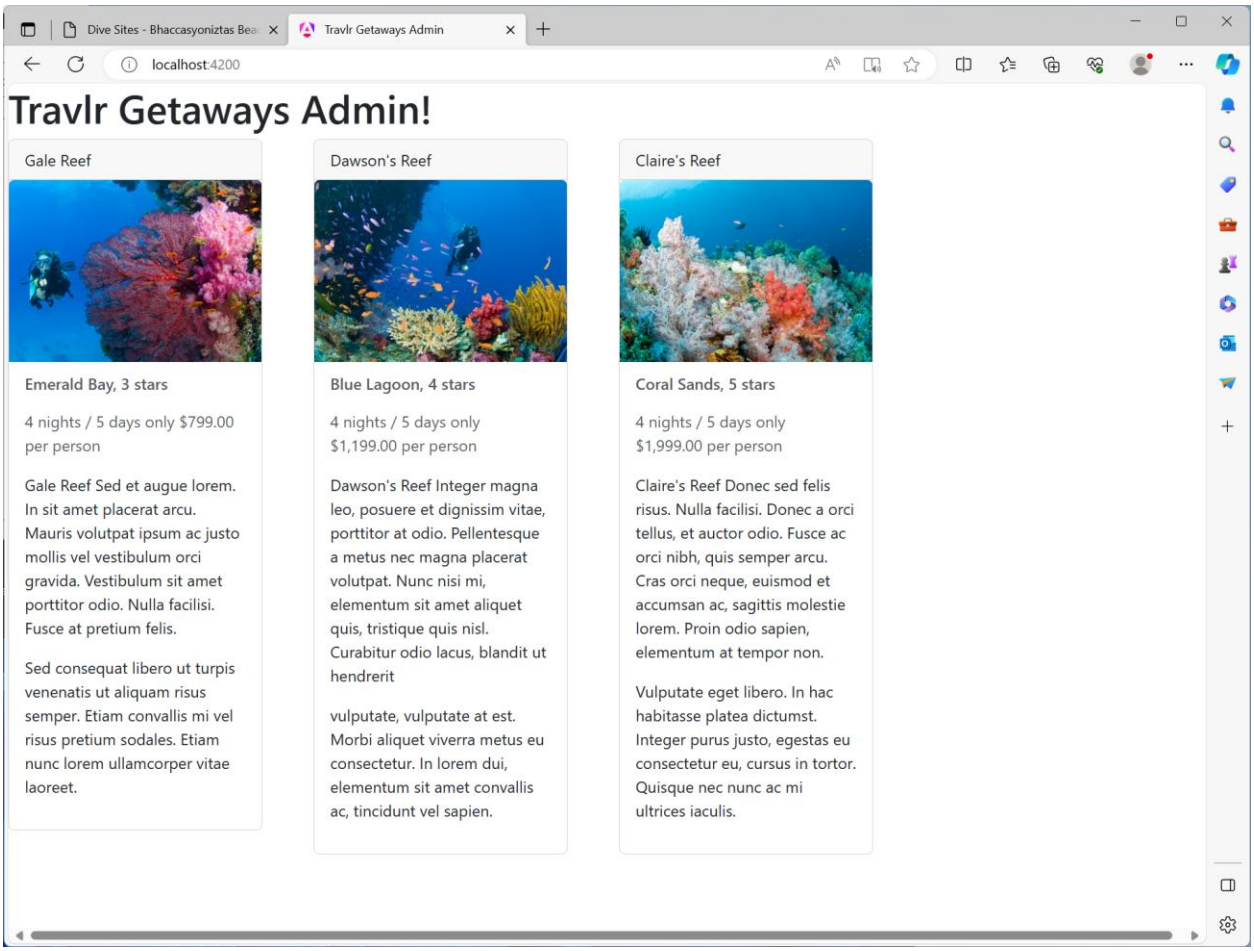


This code relies on the Bootstrap CSS framework to generate a pleasing view of the data. Please note: The cards will, by default, appear vertically in the window. To get them to appear horizontally, you need to make sure that you have enough horizontal space. Each row (in the

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

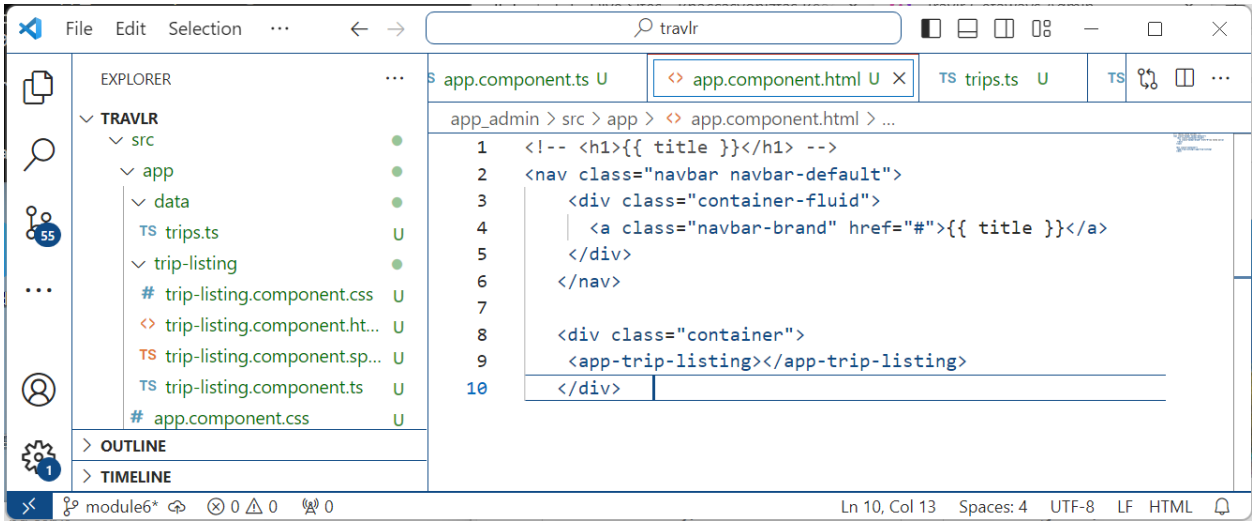


If you edit the '*ngFor' tag-set in the trip-listing.component.html file you can add a 'class' attribute that will set the number of columns that each card should consume. After 12 columns have been consumed the next cards will show beneath the first set of cards. As an example, editing the tag-set to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

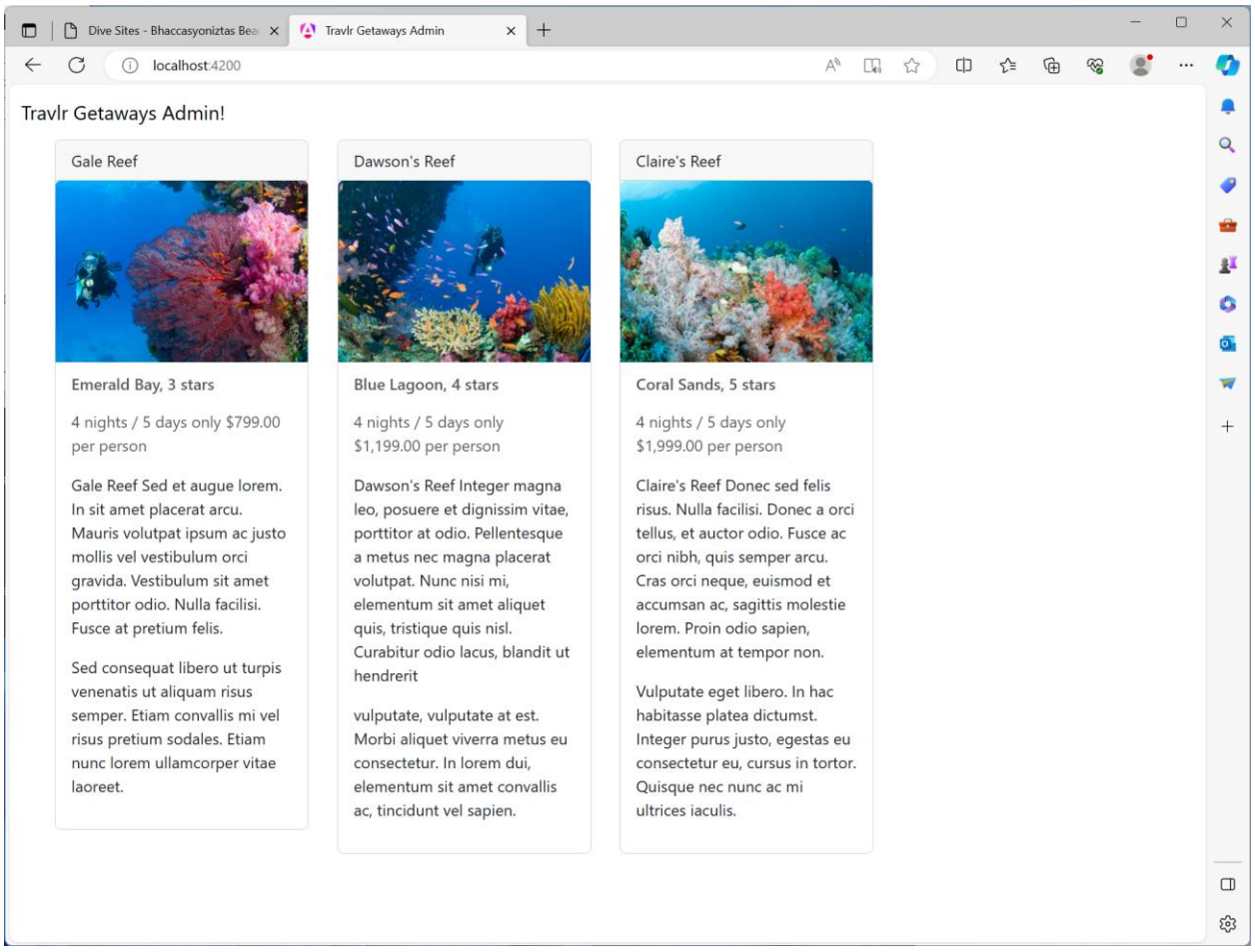


12. The ***app.component.html*** page can be cleaned up to render the listing with a navigation bar and some whitespace. To do this, we will replace the contents with:

```
<!-- <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.



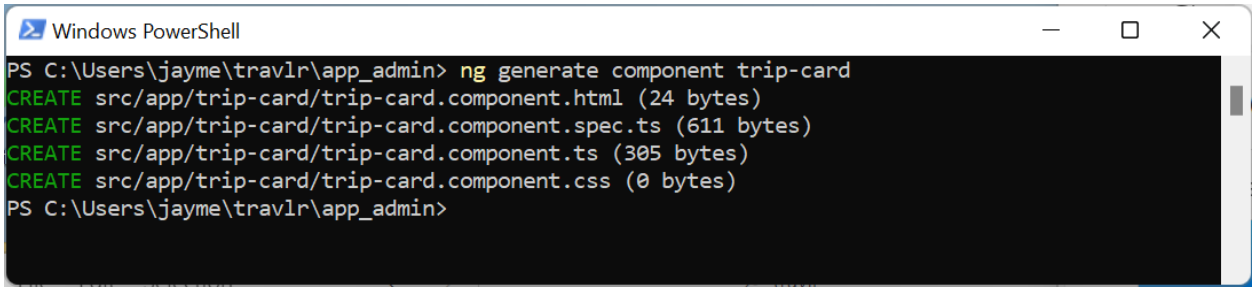
Refactor Trip Rendering Logic into an Angular Component

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:

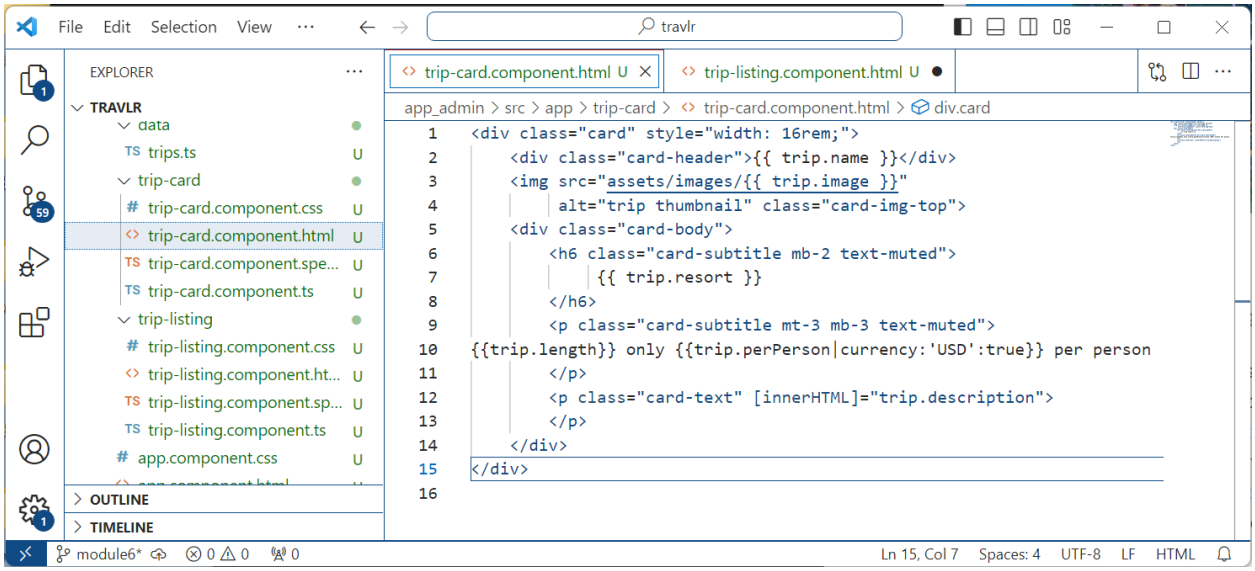
```
ng generate component trip-card
```



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:

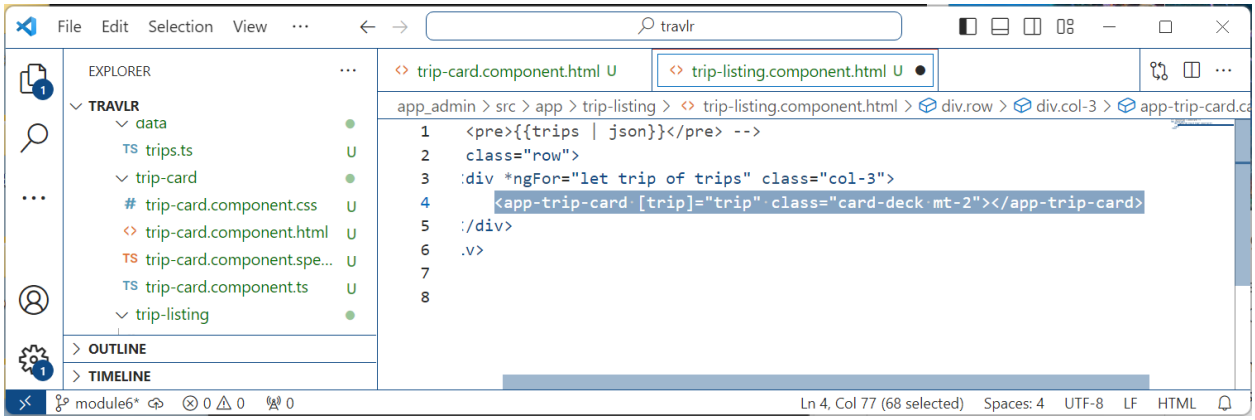
```
<div class="card" style="width: 16rem;">
    <div class="card-header">{{ trip.name }}</div>
    <img src="assets/images/{{ trip.image }}"
         alt="trip thumbnail" class="card-img-top">
    <div class="card-body">
        <h6 class="card-subtitle mb-2 text-muted">
            {{ trip.resort }}
        </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>
```

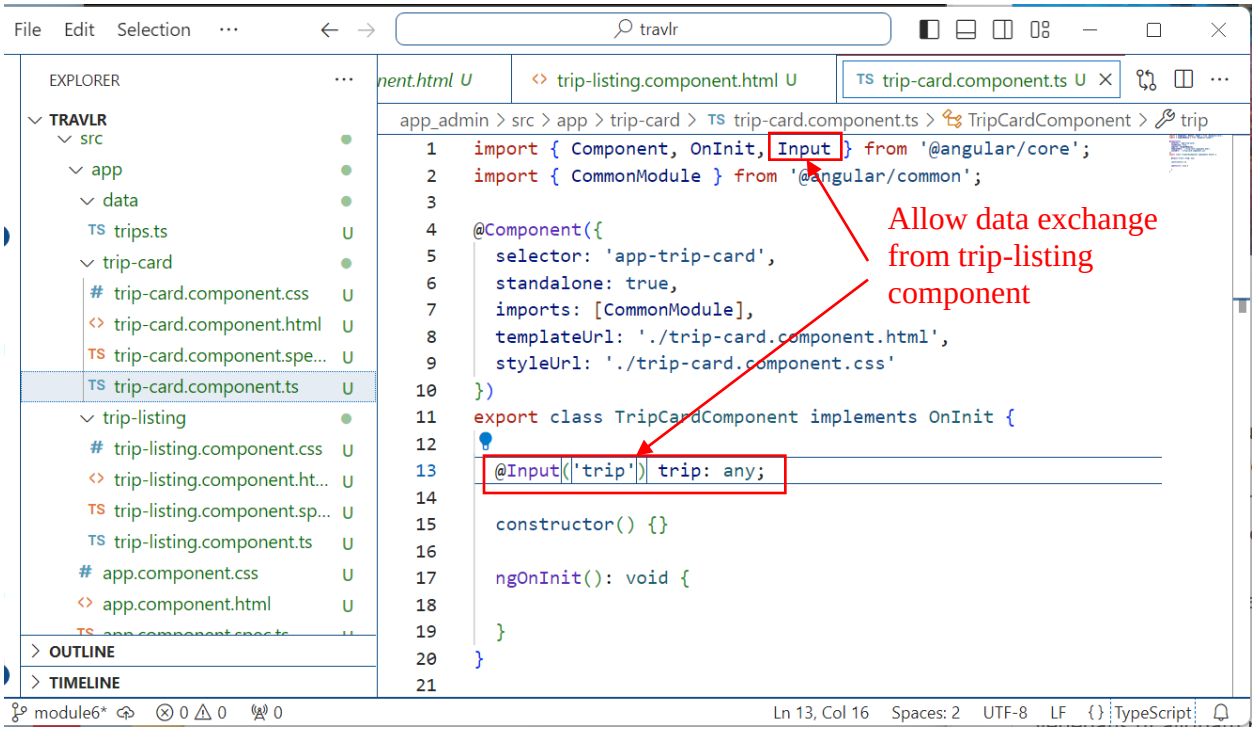


This section is replaced in the ***trip-listing.component.html*** file with:

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



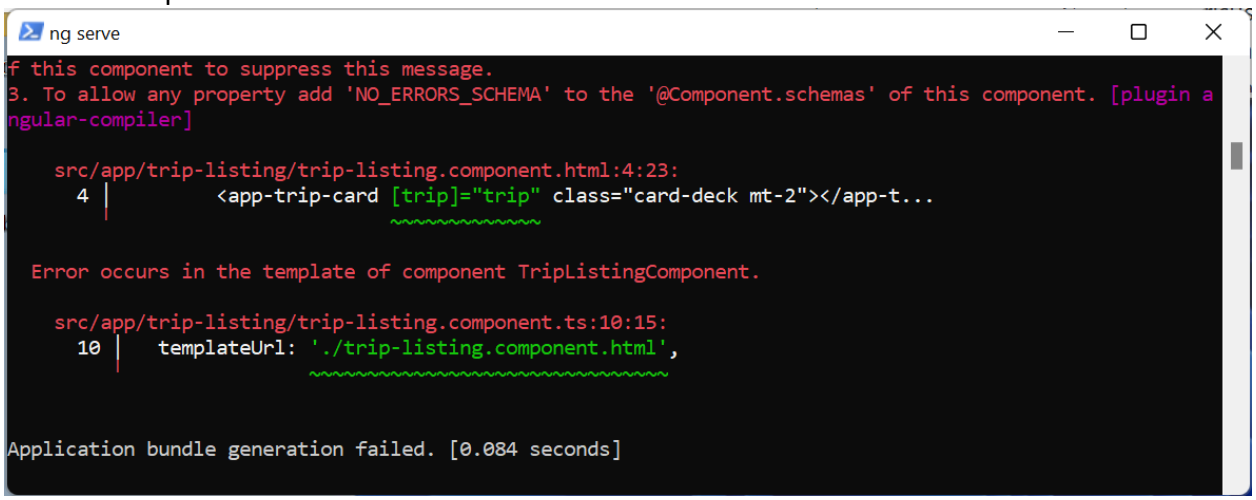
3. In order for this to get pulled into our application, we now have to add some code to the ***trip-card.component.ts*** file to add the *Input* directive in order that the *trip-listing* component can pass the *trip* data so that it can render an instance of a trip.



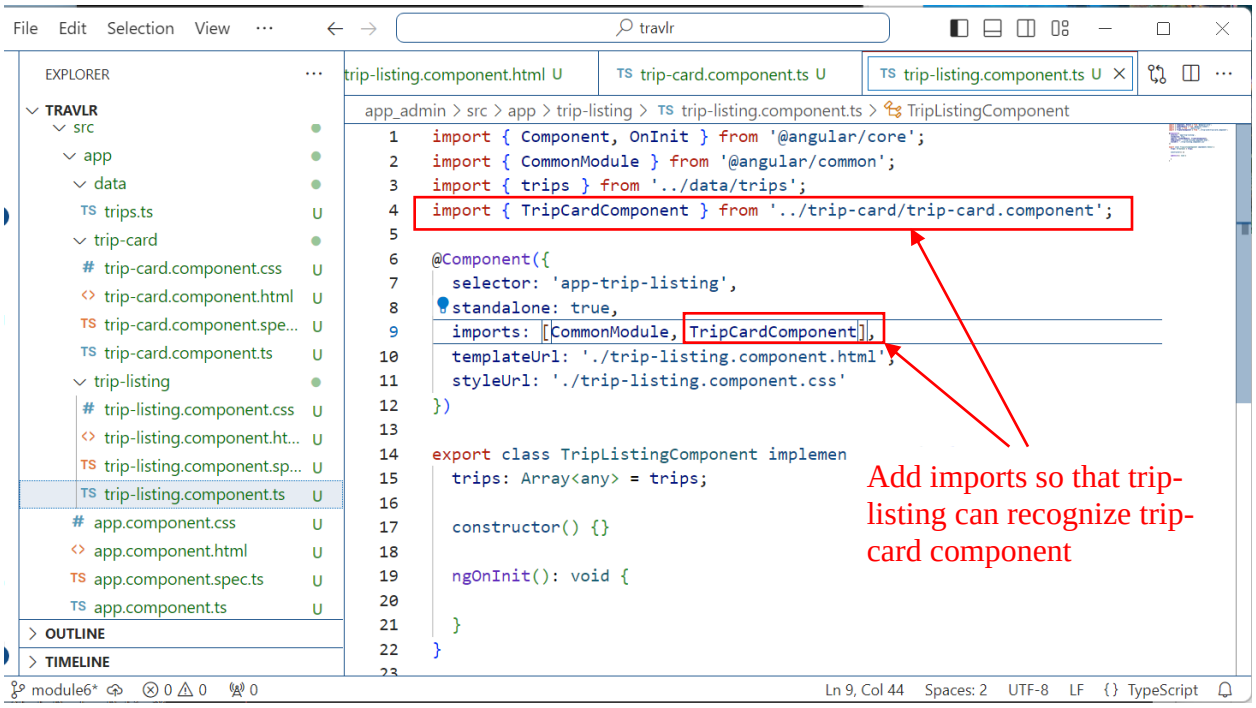
Your ***trip-card.component.ts*** file should contain this code:

```
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.

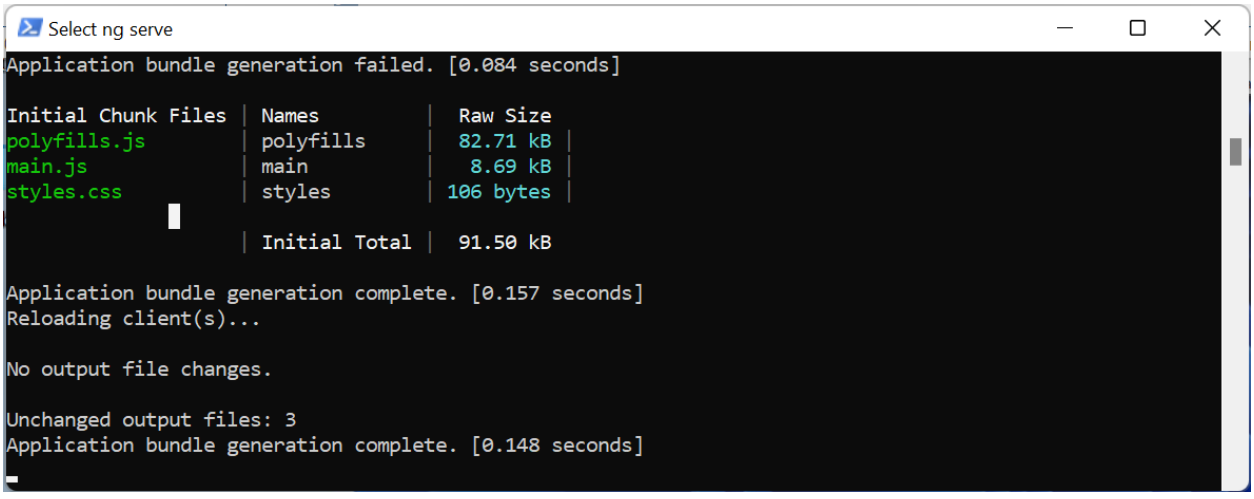


Your updated ***trip-listing.component.ts*** file should look like this:

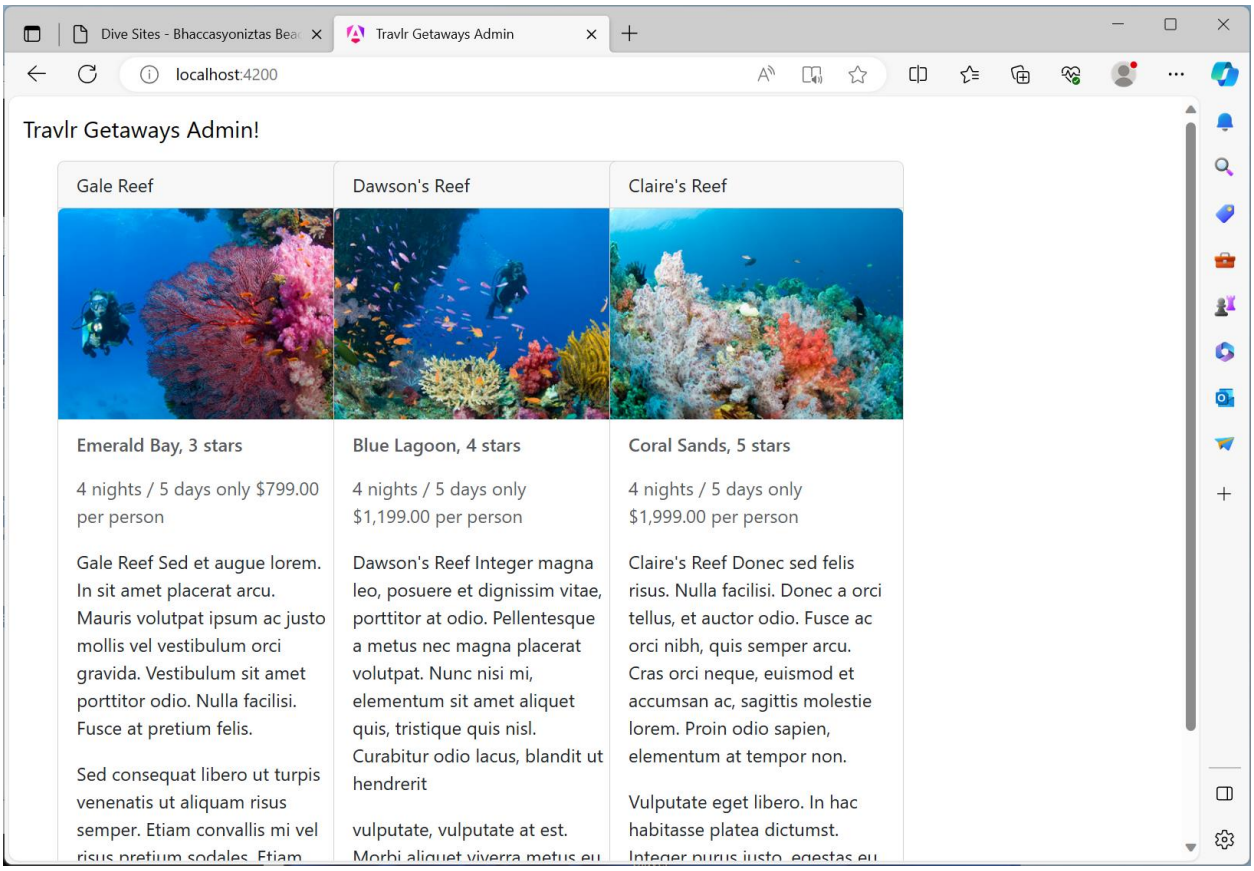
```
import { Component, OnInit } from '@angular/core';
import { CommonModule } from '@angular/common';
import { trips } from '../data/trips';
```

```
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.

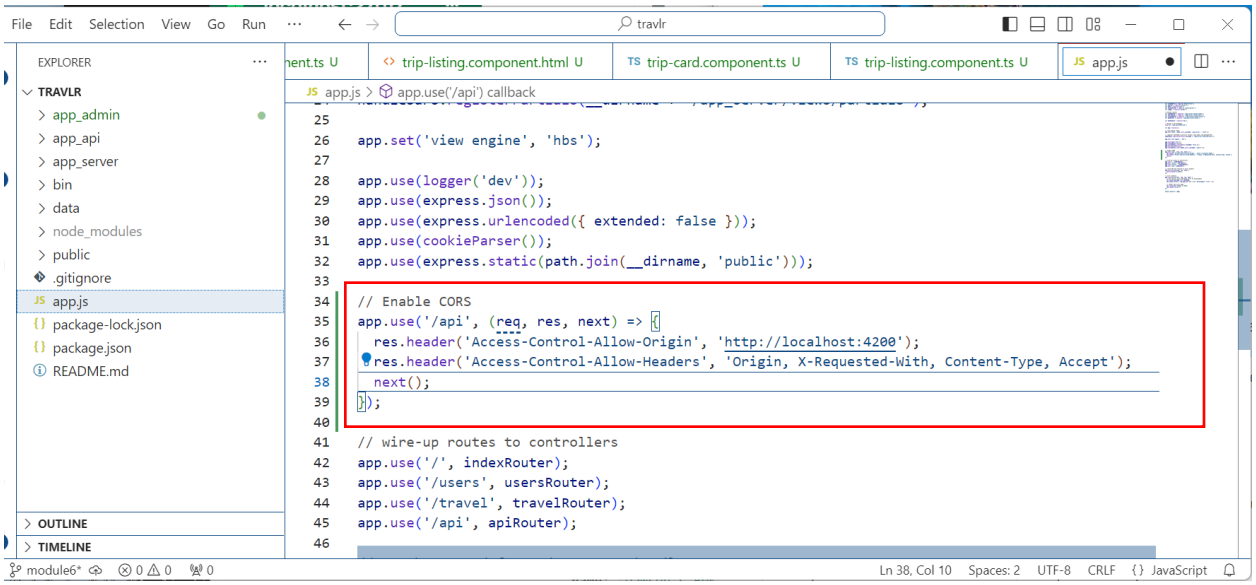
Create Trip Data Service

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

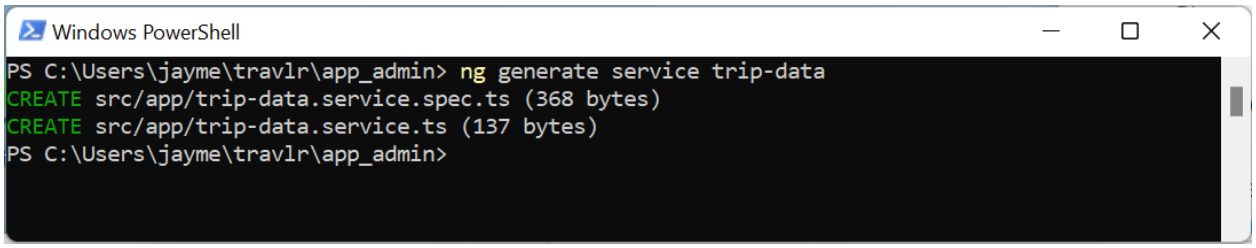
the ***app.js*** file at the root of our application tree and it enables what is generally referred to as CORS (Cross Origin Resource Sharing).



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

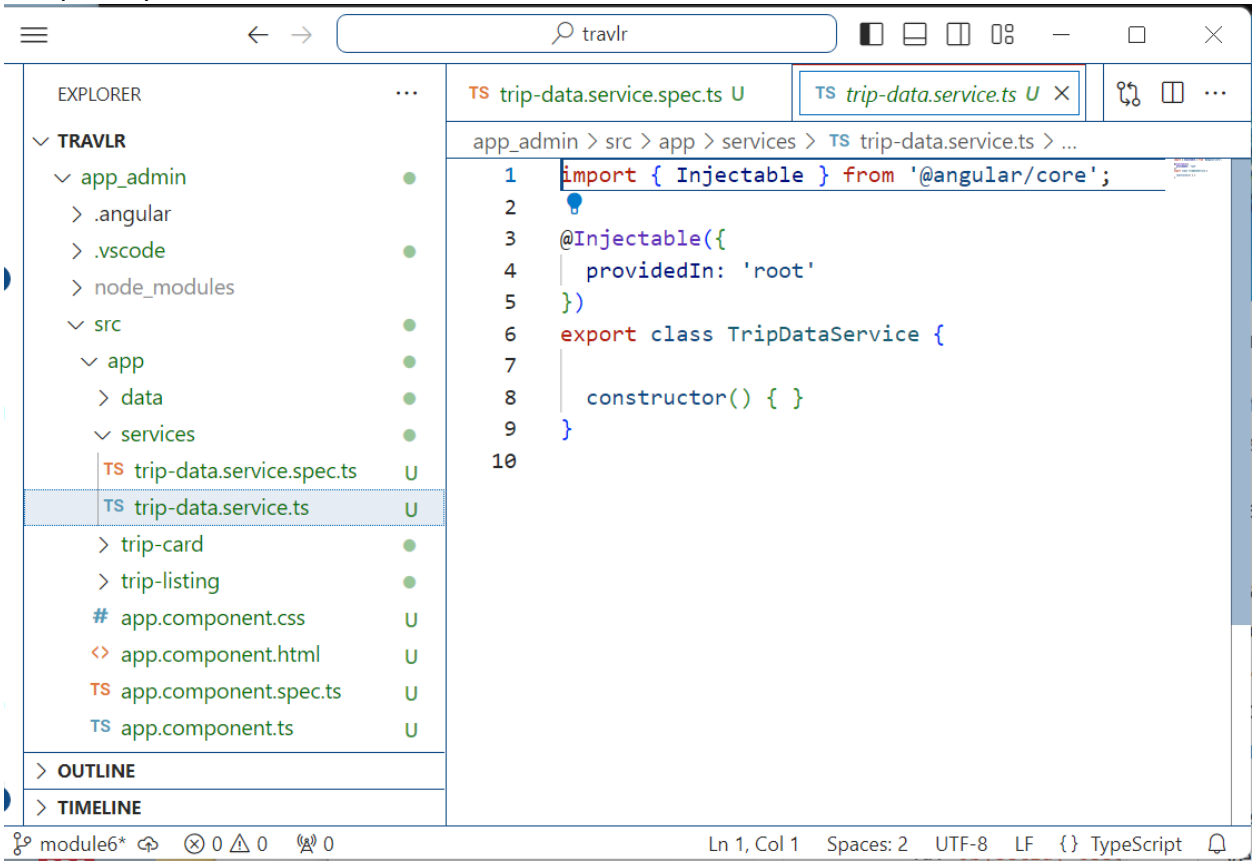
2. Now we will need to create a new Angular service called trip-data to allow us to configure the appropriate data paths. The command we need to use to accomplish this is:

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

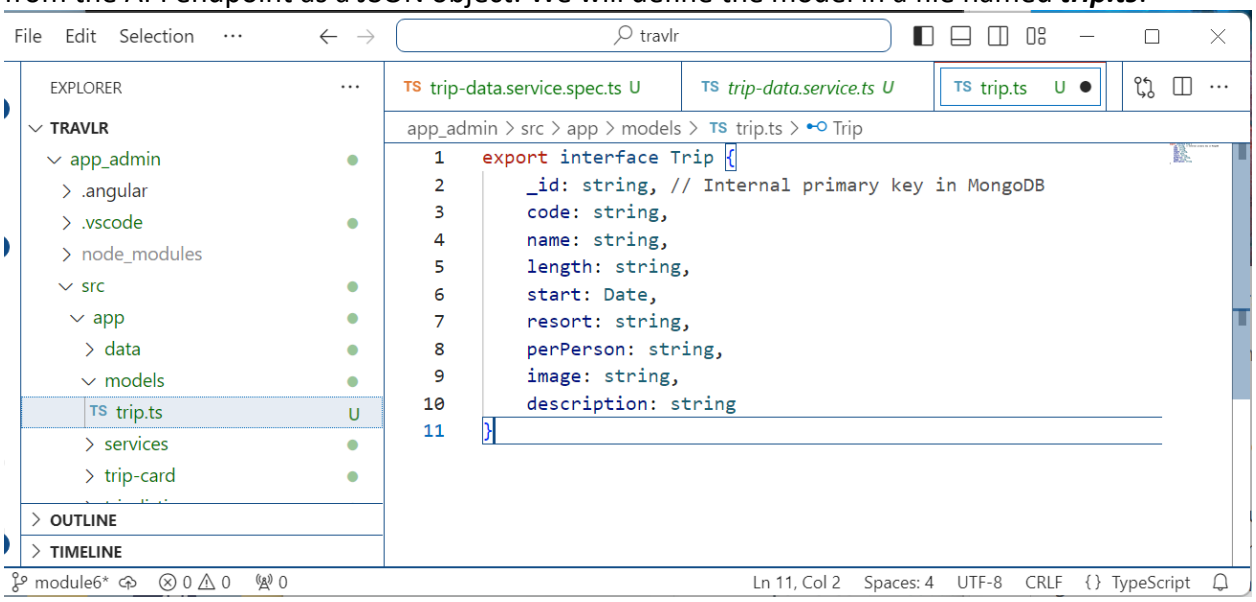


3. At this point we need to do some house-keeping in order to prepare the application structure for additional services later on. We will want to create a new folder under ***app_admin/src/app*** called ‘***services’*** and move the ***trip-data.service.**** files from the ***app_admin/src/app*** folder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

complete, your file structure should resemble this:



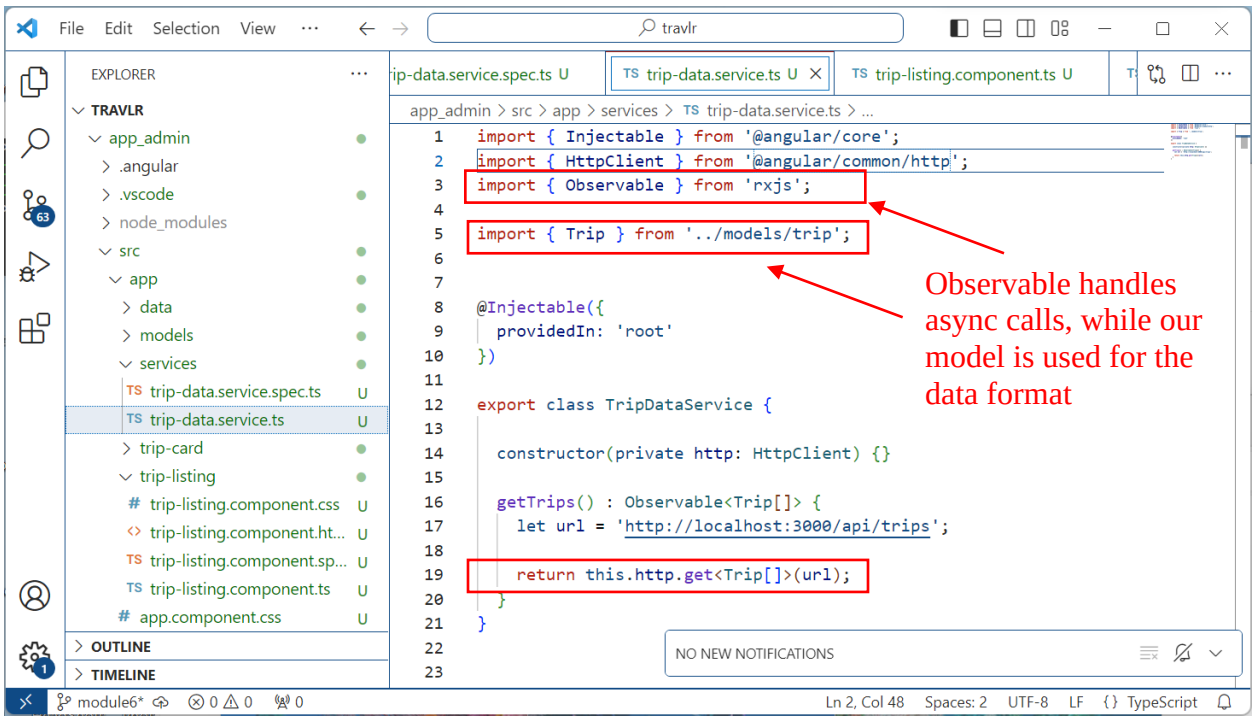
4. In order for our new basic service to function properly, we need some way to communicate what type of data it is going to handle. For this we need to create a ***models*** folder. We are going to put this parallel to the ***data*** and ***service*** directories under ***app_admin/src/app***. This 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.

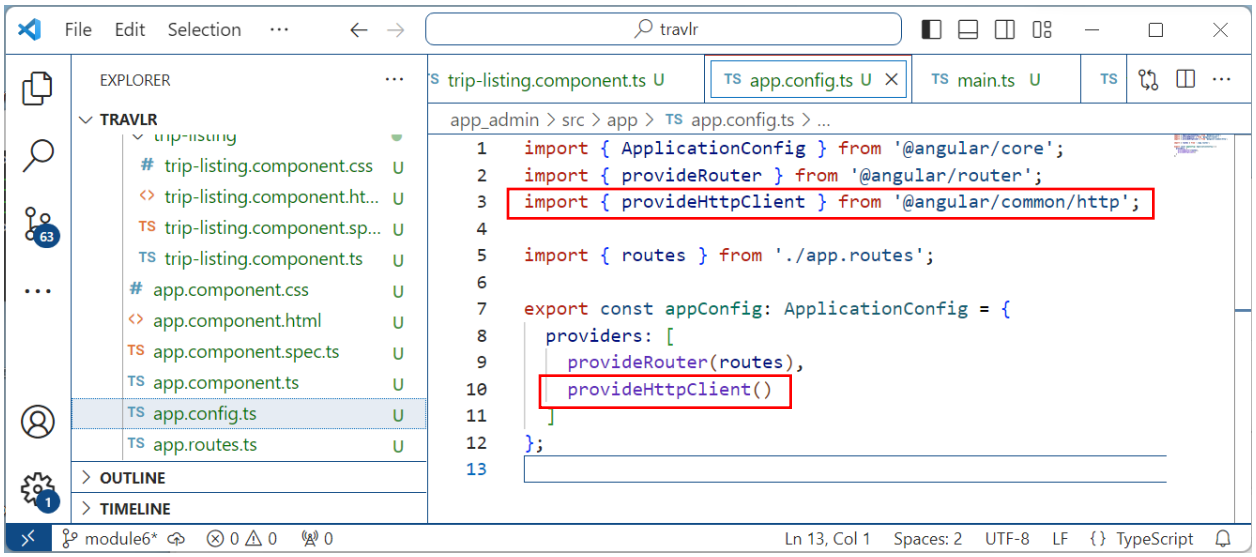
5. Now that we have the interface defined, we need to introduce it to the service. In order to accomplish this, we need to edit the ***trip-data.service.ts*** file and add some code. This is going to be very straight-forward because we aren't going to introduce error handling on the connection at this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

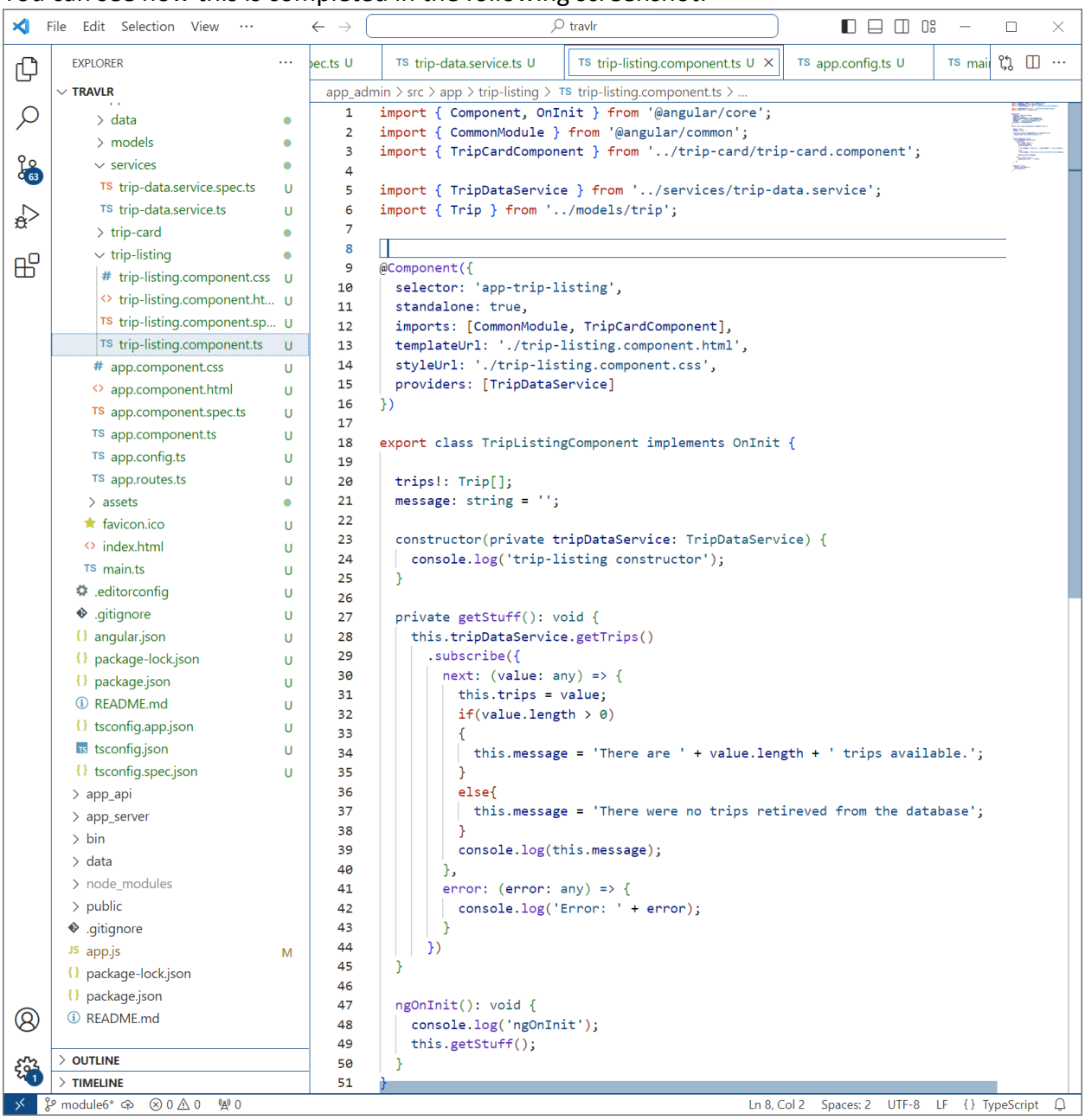
```
private getStuff(): void {
  this.tripDataService.getTrips()
    .subscribe({
```

```
        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.

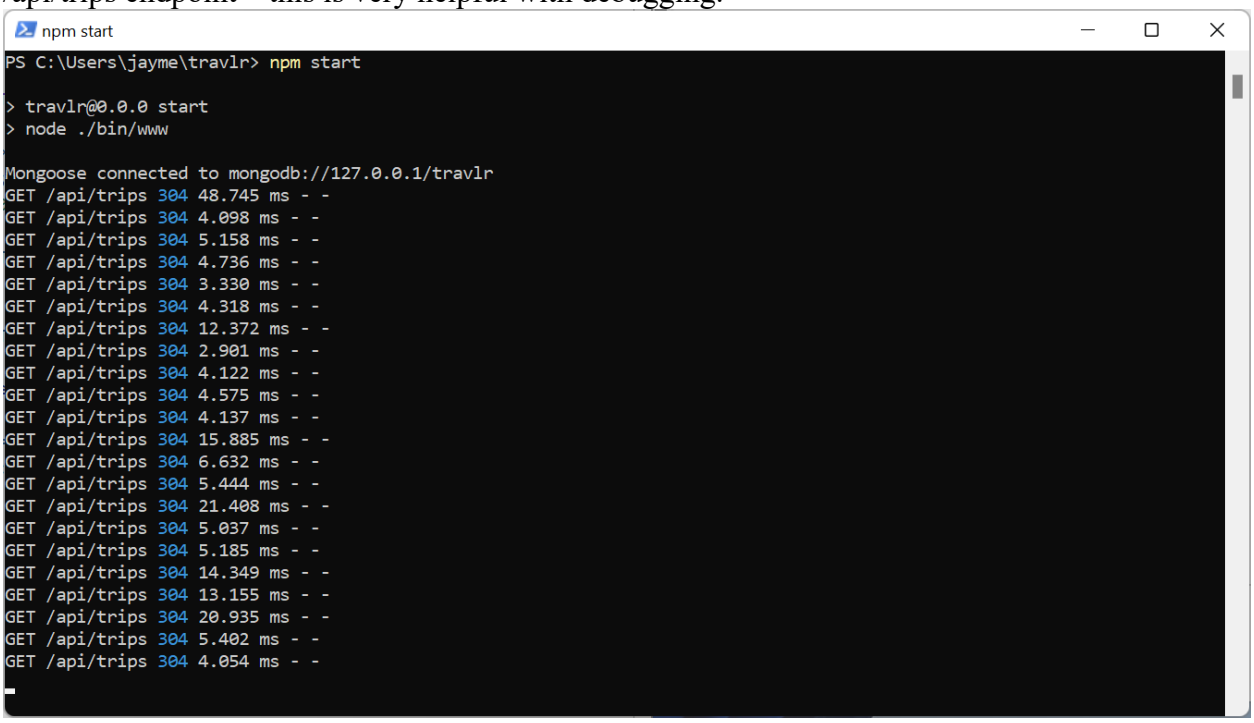
```
  ngOnInit(): void {
    console.log('ngOnInit');
    this.getStuff();
  }
```

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

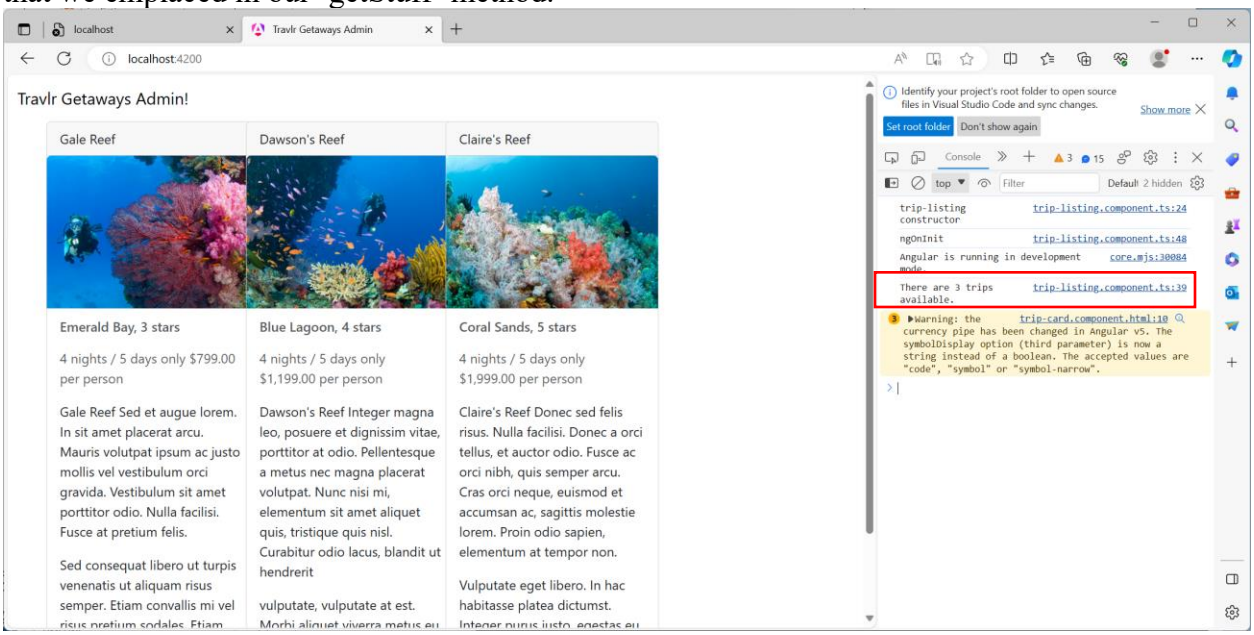


If you haven't already done so, make sure that you have your Express app running in a PowerShell Window. From this console, you can see each time an API call is made to the

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

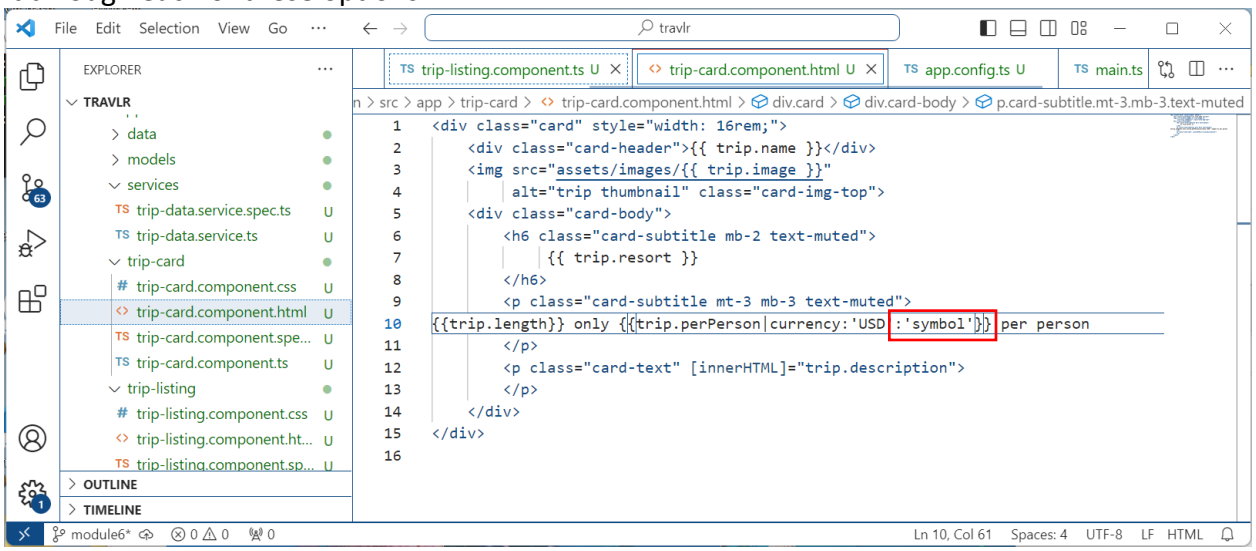


Additionally, if you haven't already, you should open the developer tools panel for your web-browser. In the Microsoft-Edge Browser, you can achieve this with CTRL-Shift-I. This is a very, very useful method to assist with debugging Angular applications – as you can watch the local console and output *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.

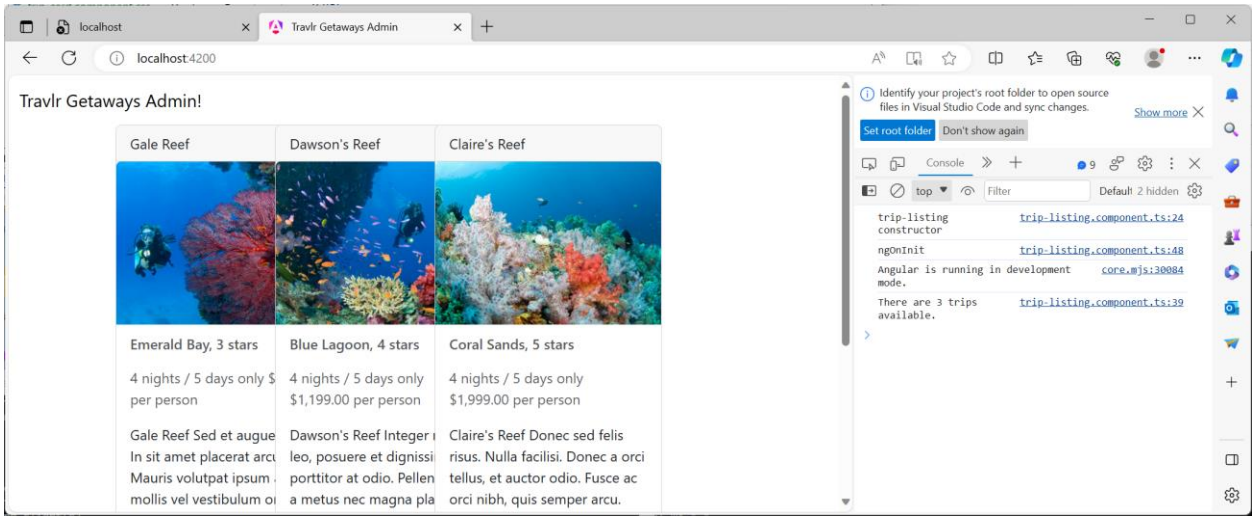


You can also see warnings in this window. In this case, we find that there is an issue with the currency pipe that we are using in the ***trip-card.component.html*** file. It is not uncommon for newer versions of Angular to deprecate (make obsolete) items from previous versions. When we find these items,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

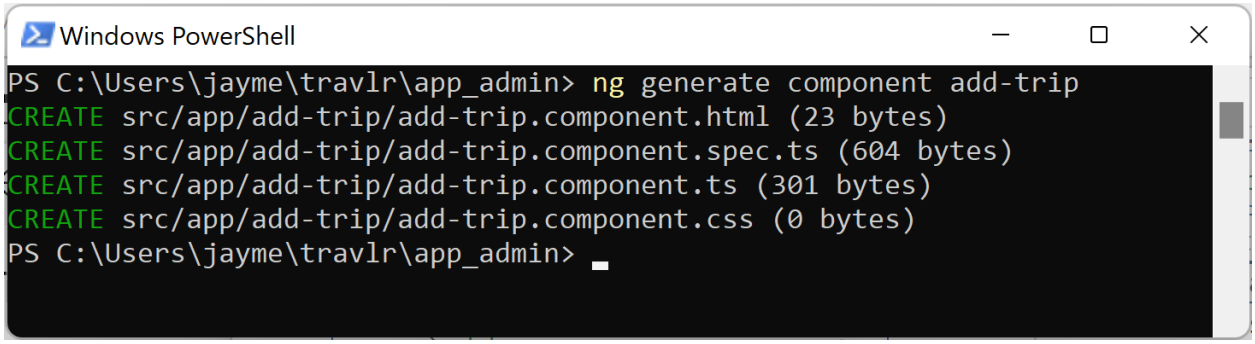


Add Trips

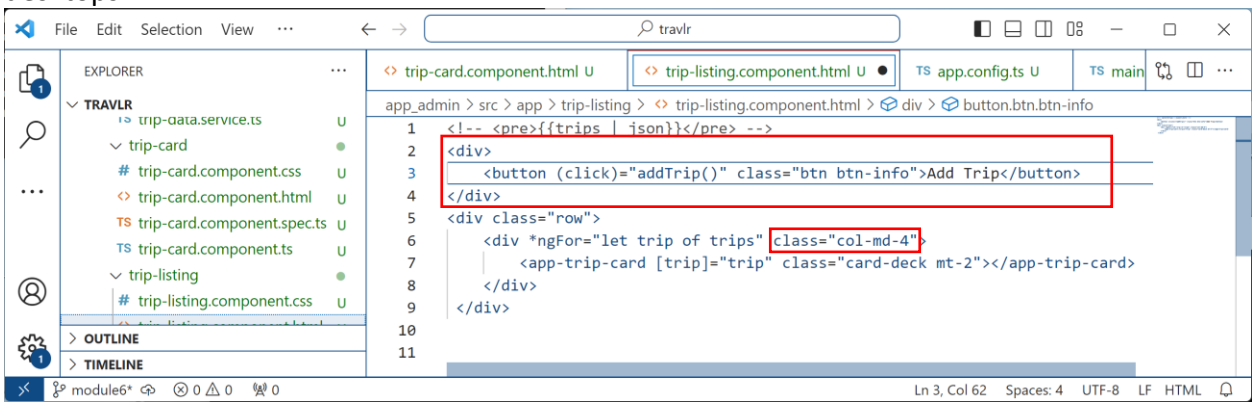
To this point, we have created an application that will display our trips, but if we wanted to make any changes to a trip definition or add a new trip, we would need to go and manually edit our Mongo Database. That really isn't practical in a production application, so we are going to add some code that 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
```

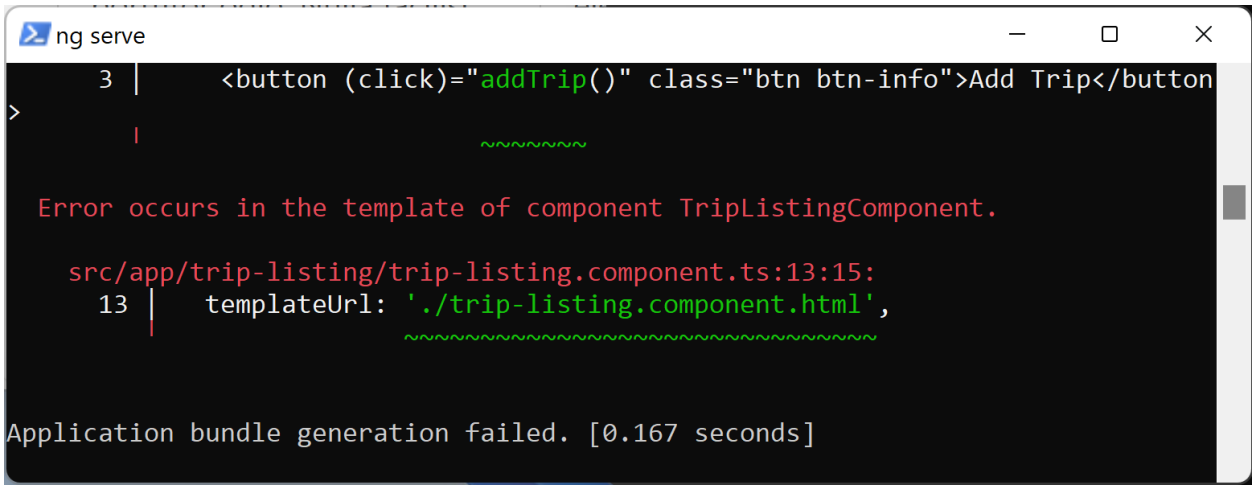


2. The next thing we are going to do is to make a change to our ***trip-listing.component.html*** file to add a button that we will utilize to select the add-trip functionality for our application. At the same time, we are going to make a small change to the column definition for our application to make 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.



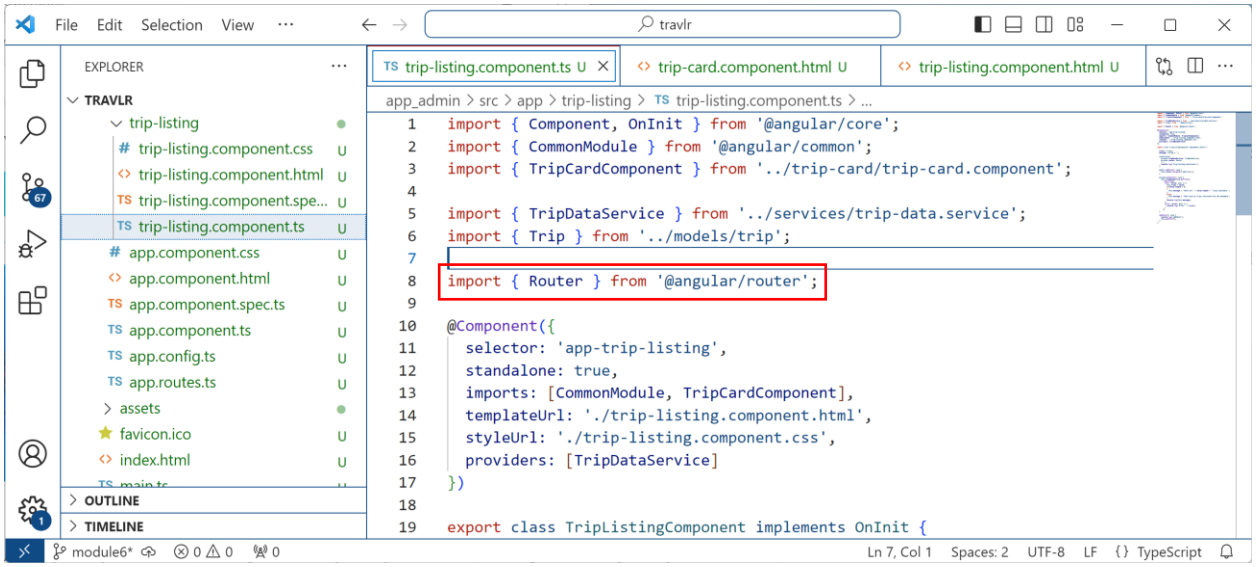
You will notice that the button is set to execute an addTrip() method when pressed. We will be wiring this up in the coming steps. The change to the class definition for our trip selector specifies to Bootstrap that each card will consume 4 of 12 columns on a medium or larger screen (768 pixels or greater)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



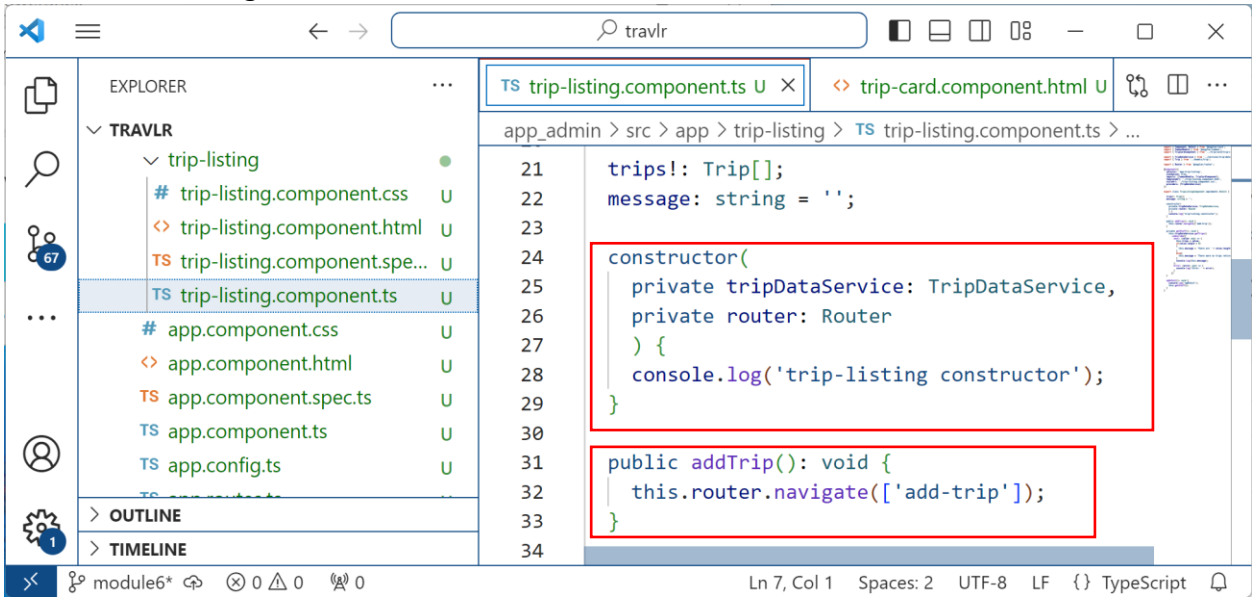
So the next step becomes modifying the ***trip-listing.component.ts*** file to add our new capability.

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:

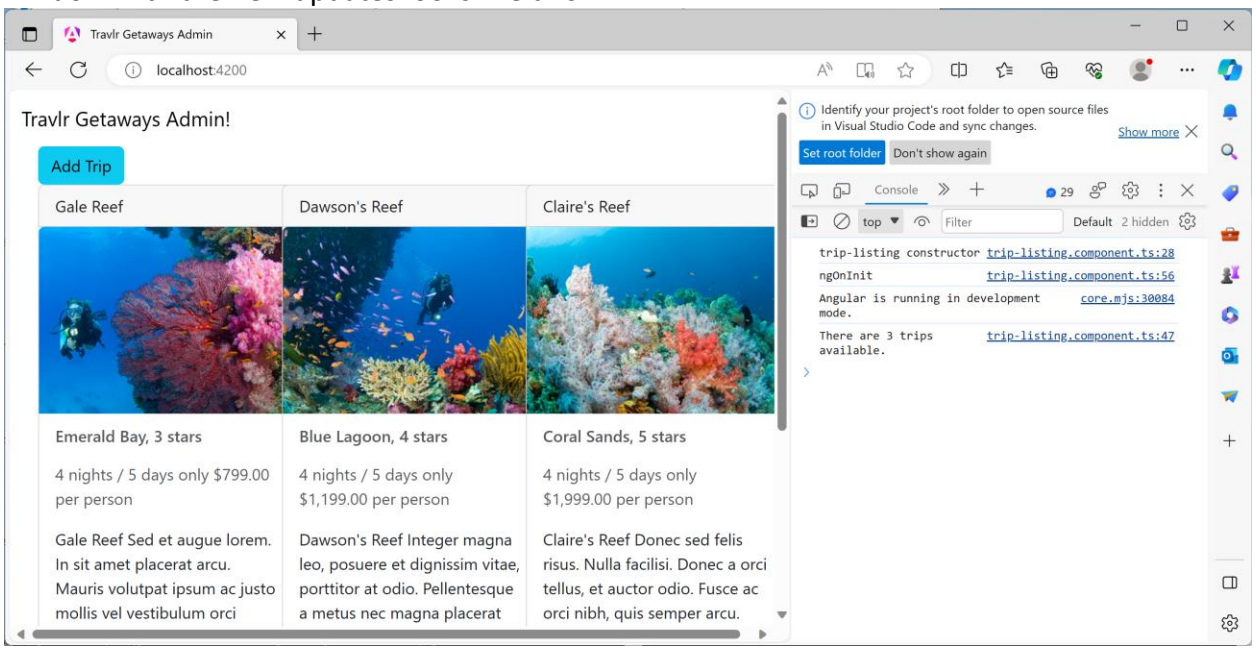


This pulls in the Angular routing capability.

The final two changes are shown here:

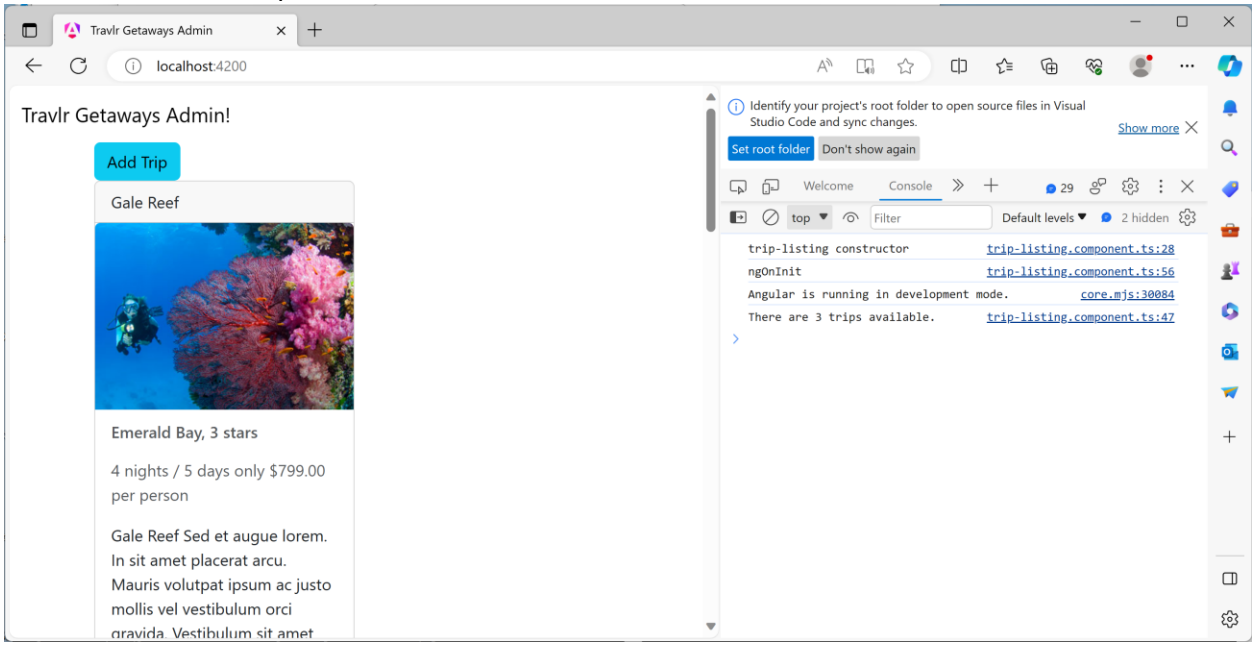


Please note that we need to make the addTrip() method public because it is going to need to be accessed externally from the perspective of the trip-listing component. At this point, our window with the new updates looks like this:

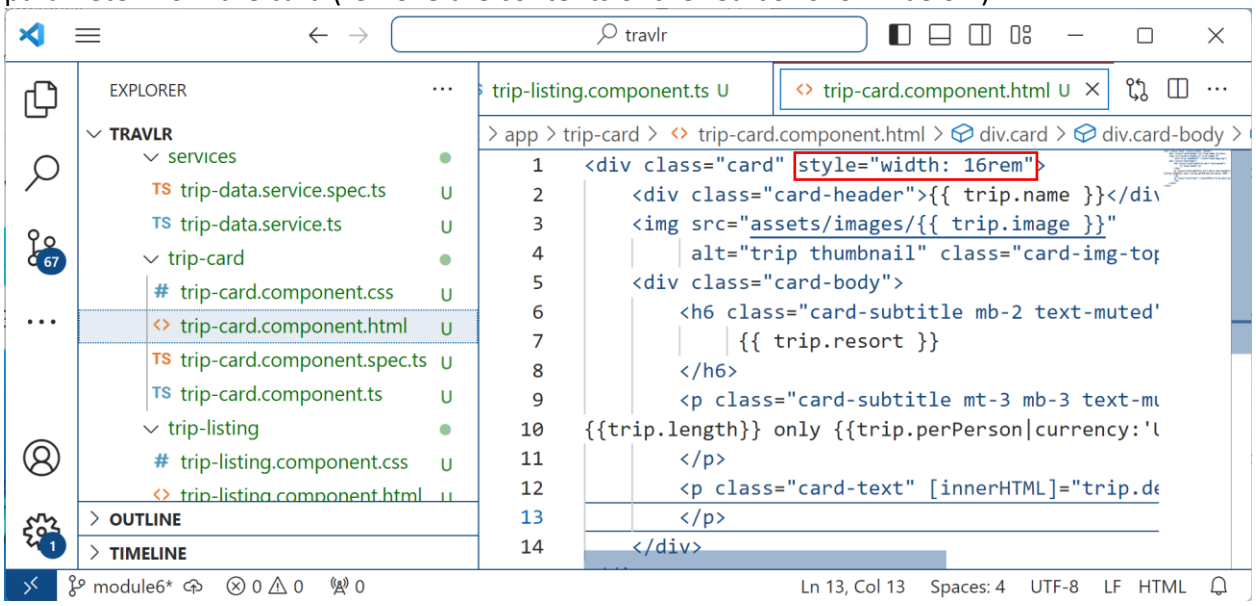


This is visually pleasing, but if you manipulate the screen size and shrink it, you will notice that the on a smaller screen with vertically stacked cards, the cards have the same width and the scale doesn't look quite right. The following is what we see with the viewport

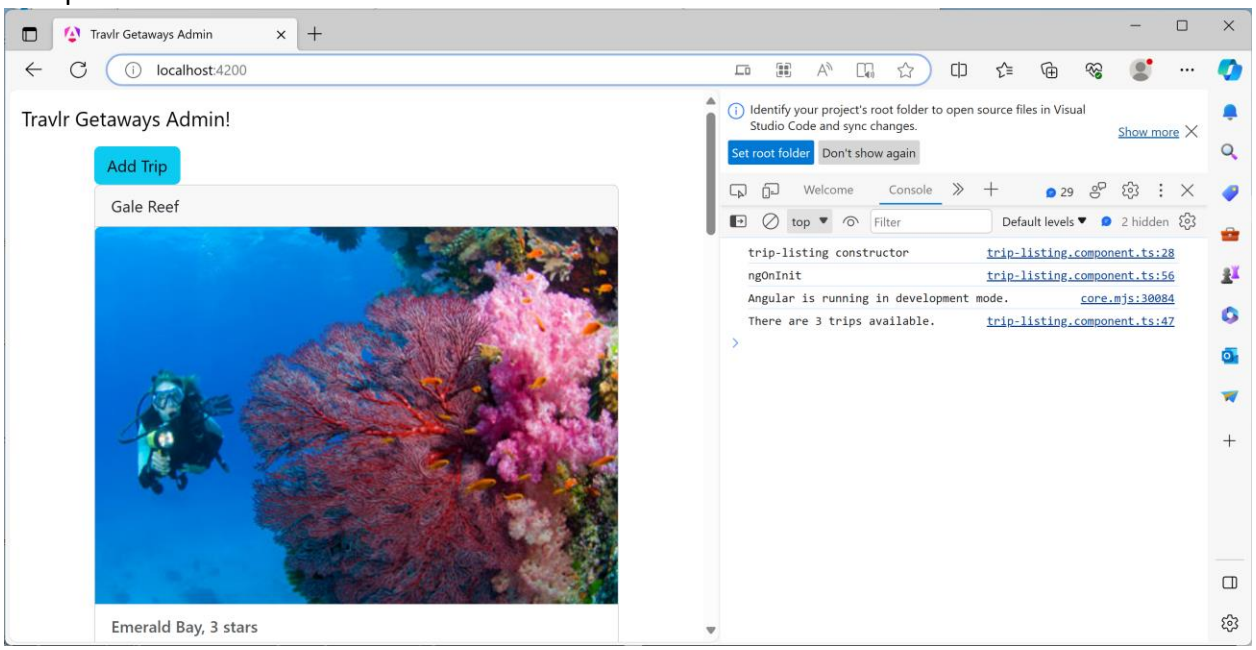
shrunk to below 768 pixels in width:



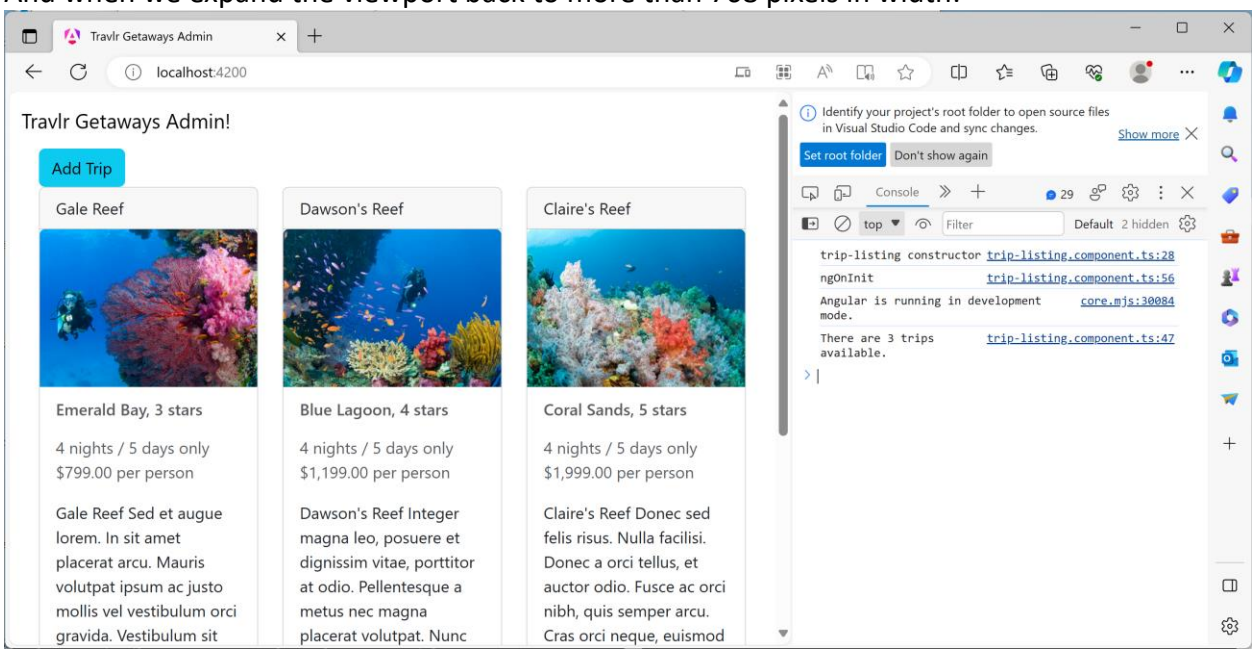
We can make one more small modification to improve things and make them scale dynamically. So let's edit the ***trip-card.component.html*** file and remove the width parameter from the card (remove the contents of the red box shown below):



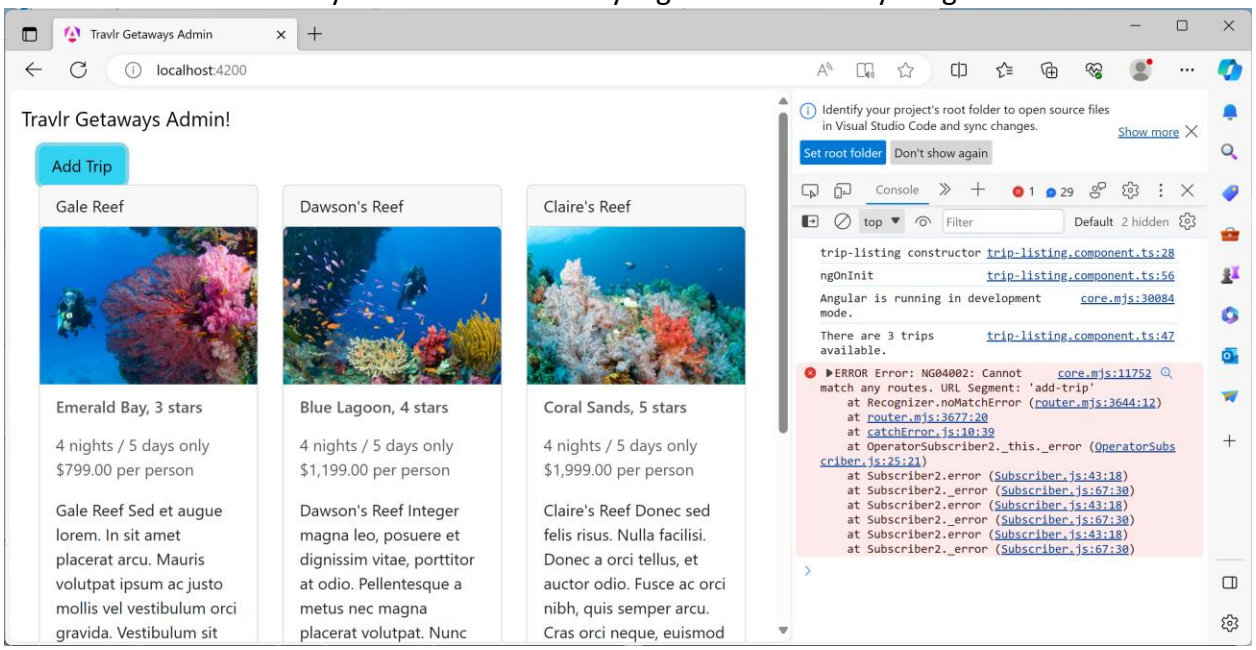
With this change, you can see that the card will now automatically scale with the size of the viewport:



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

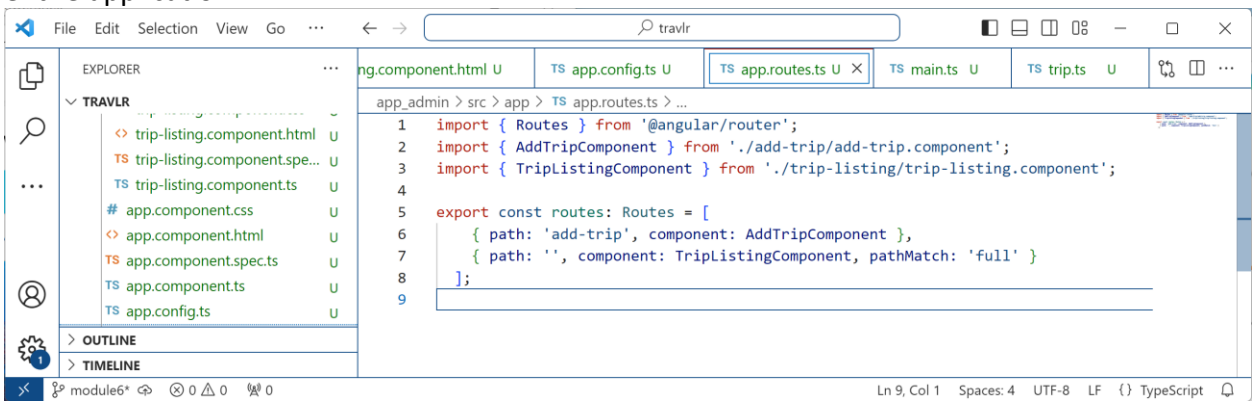


5. If we were to test our application now by pressing the *add-trip* button we would generate an error because we do not yet have the necessary logic for it to do anything:



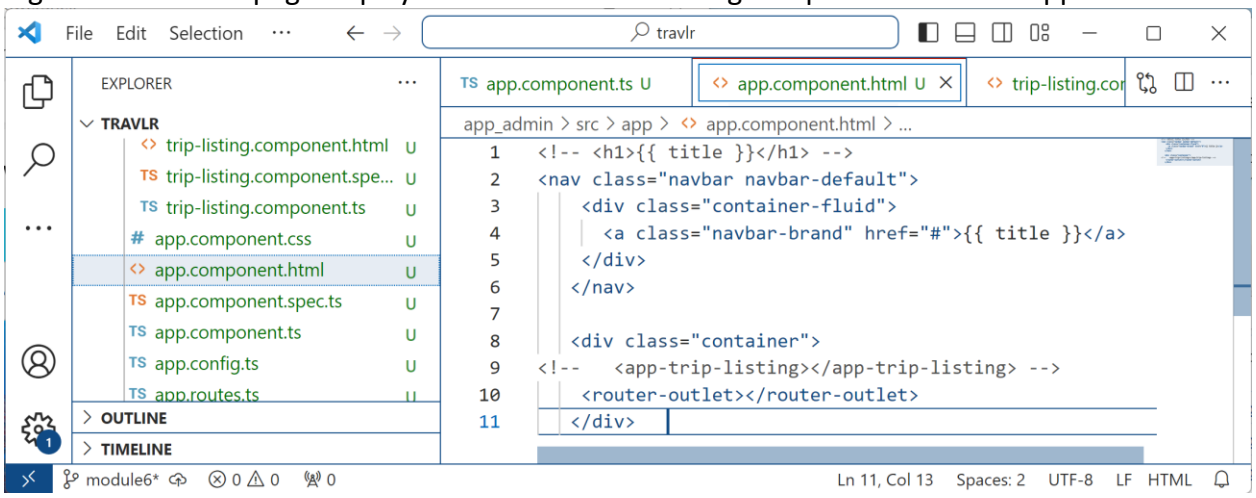
Which leads us into the next step for the application which is to wire-up the routing capability for our application. The built-in routing capability in Angular is very powerful, and will allow us to update our application to become more dynamic with its display capabilities.

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.

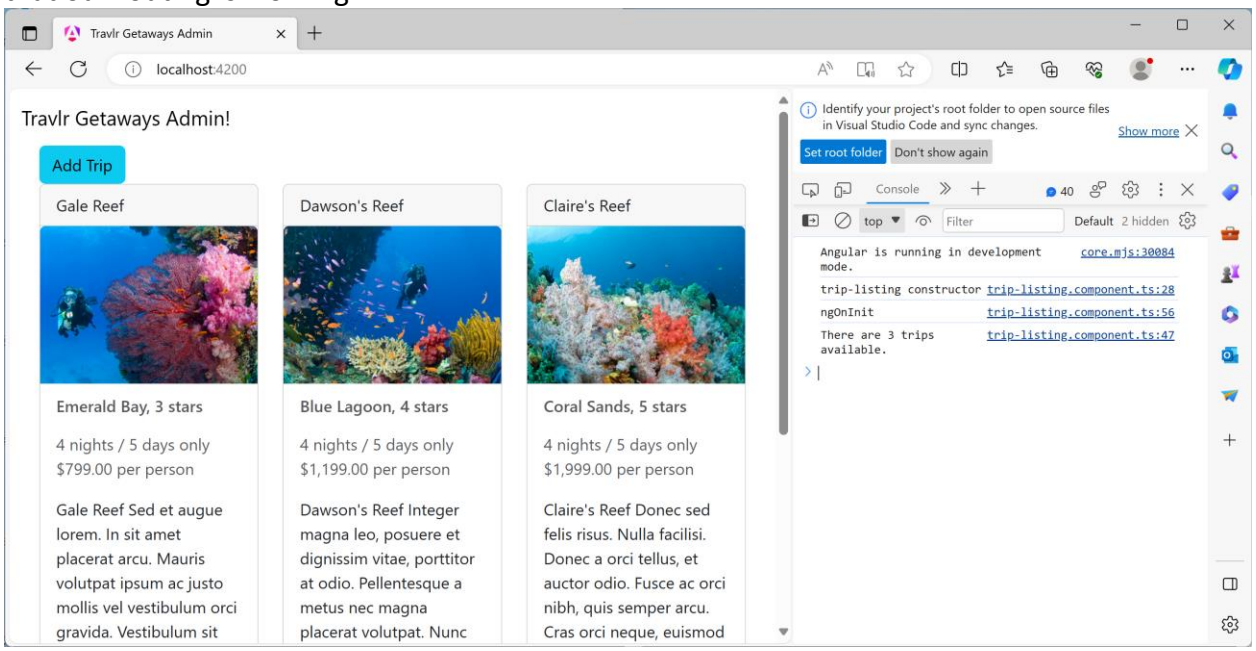


As you can see we created two paths in our definition – this is so that we can activate each of these components separately with our application. Please Note: You must also import each component for which you are providing a route.

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:

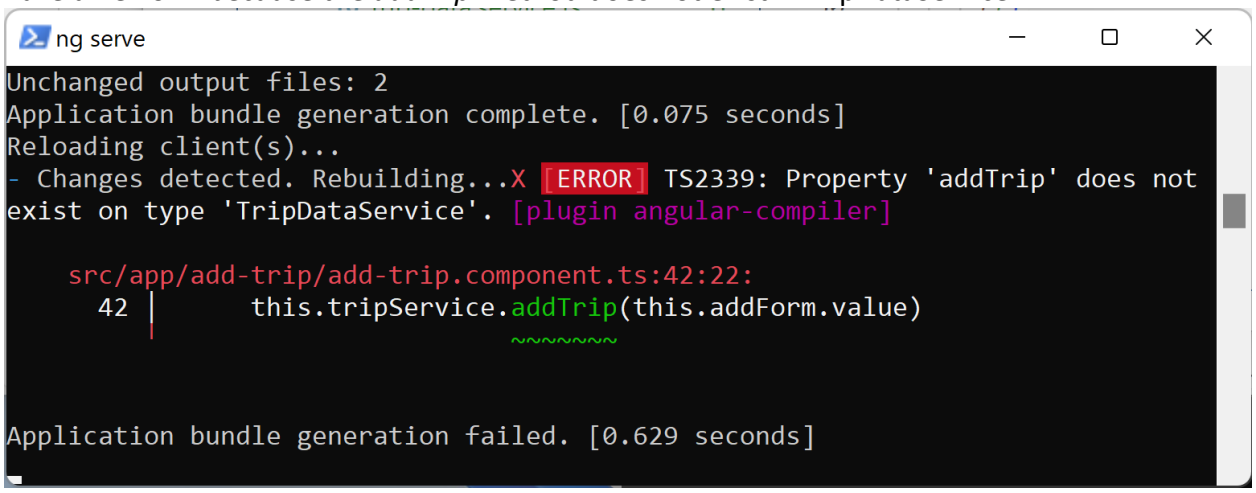
```
import { Component, OnInit } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormBuilder, FormGroup, Validators } from "@angular/forms";
```

```
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.

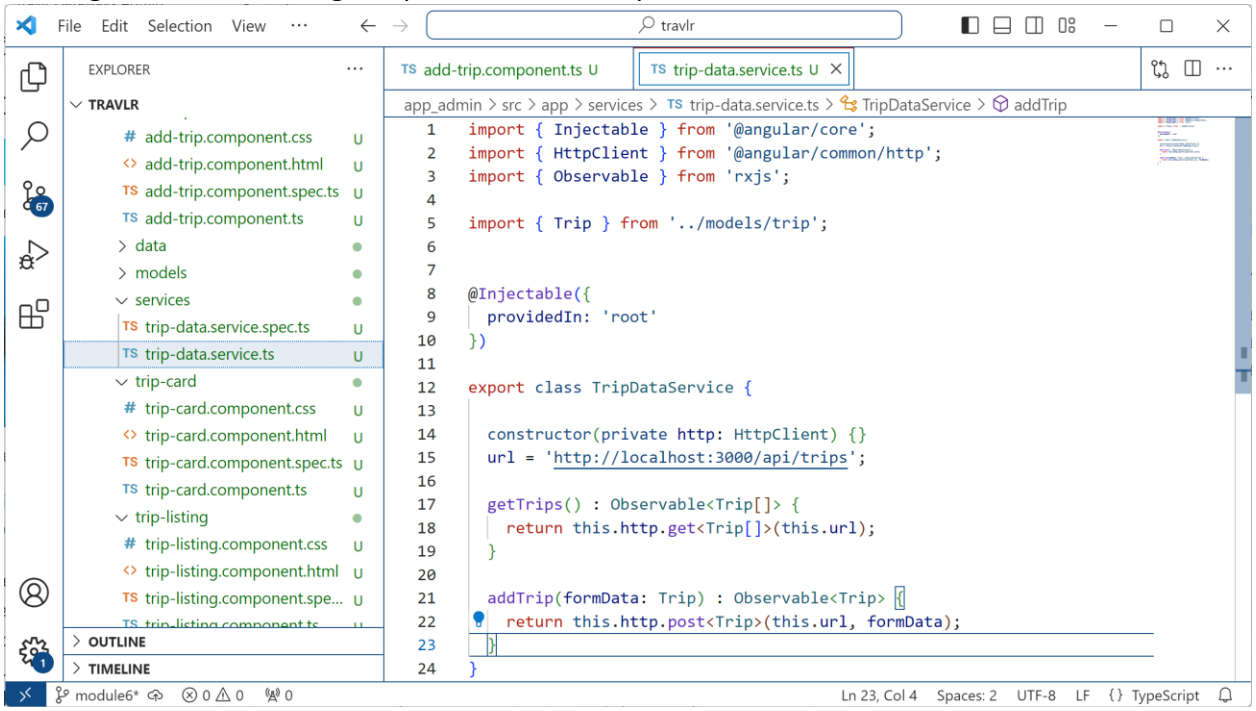
The onSubmit() method that we create is declared as public so that it can be called on the button-press, and it utilizes the TripDataService that to pass the data back to the Express application. You can see the similarity in how we handle the subscription to the TripDataService method – it is the same 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:

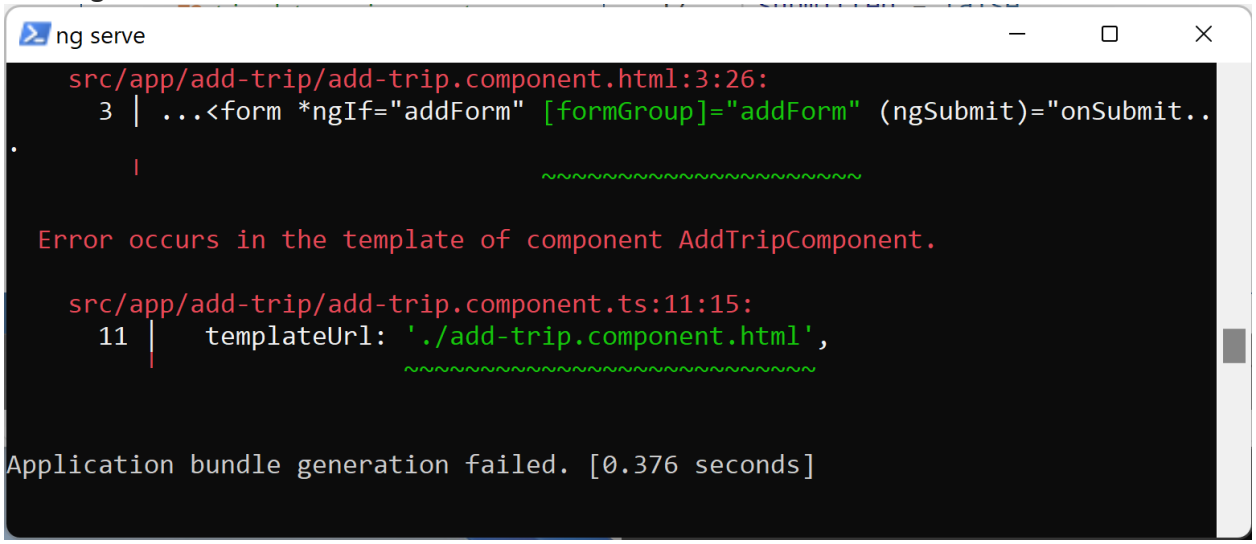
```
<div class="col-md-4">
    <h2 class="text-center">Add Trip</h2>
    <form *ngIf="addForm" [formGroup]="addForm"
(ngSubmit)="onSubmit()">

      <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']">
```

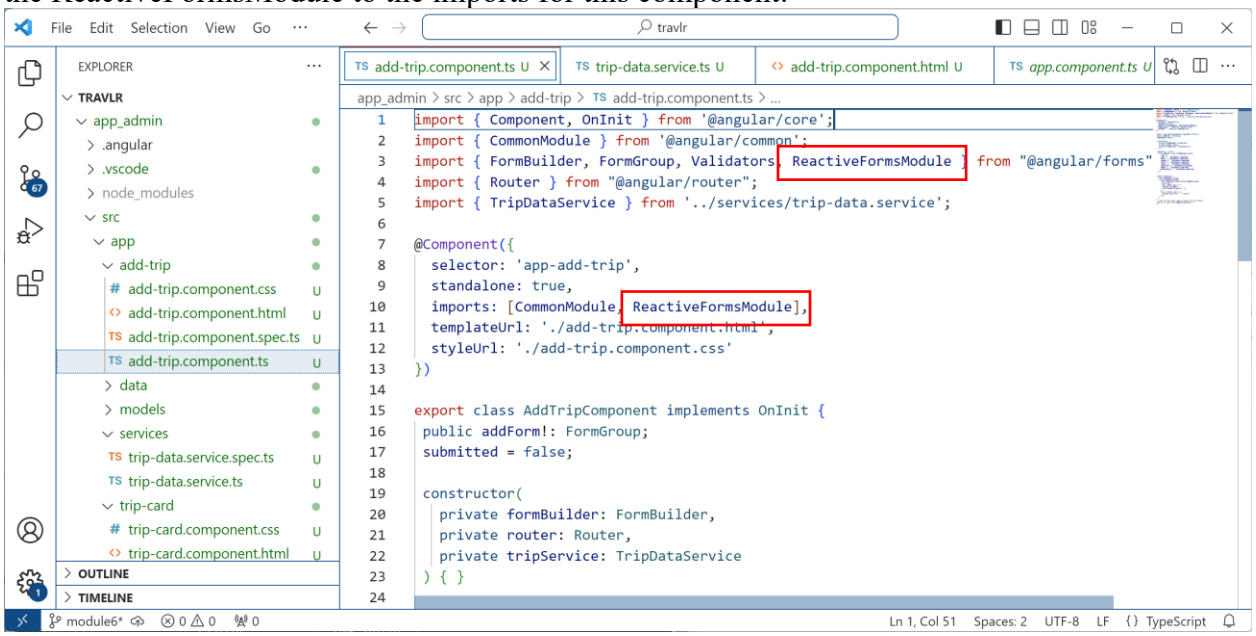
```
            Description is required
          </div>
        </div>
      </div>

      <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:

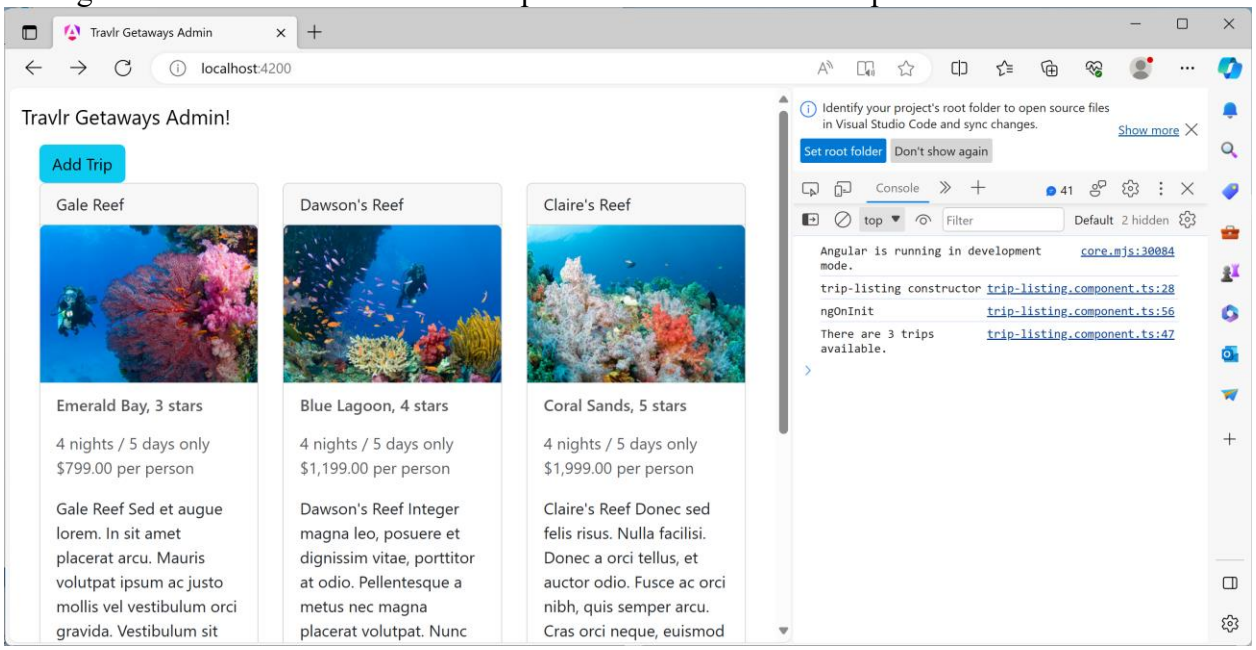


When we double-check our add-trip.component.ts file, we find that we have the appropriate definitions for addFrom, and we imported FormGroup, but it still isn't working. We need to make two more changes. We need to make addForm a public variable, and we need to add

the ReactiveFormsModule to the imports for this component:



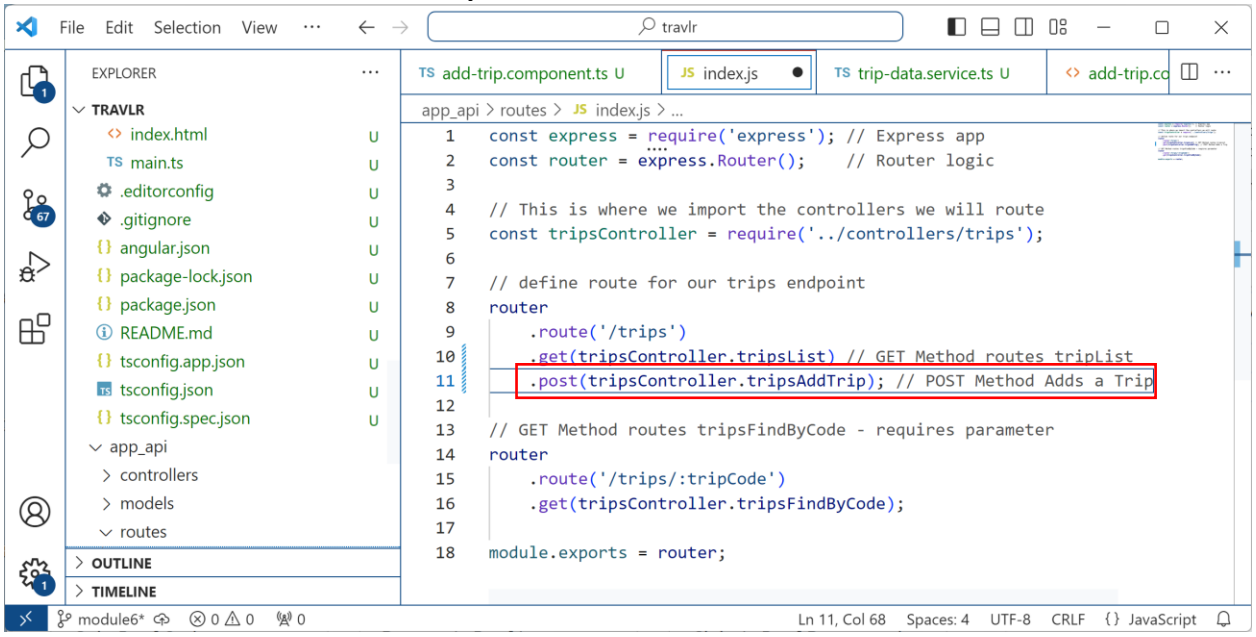
Saving the file will result in a clean compile and no errors in the inspection window:



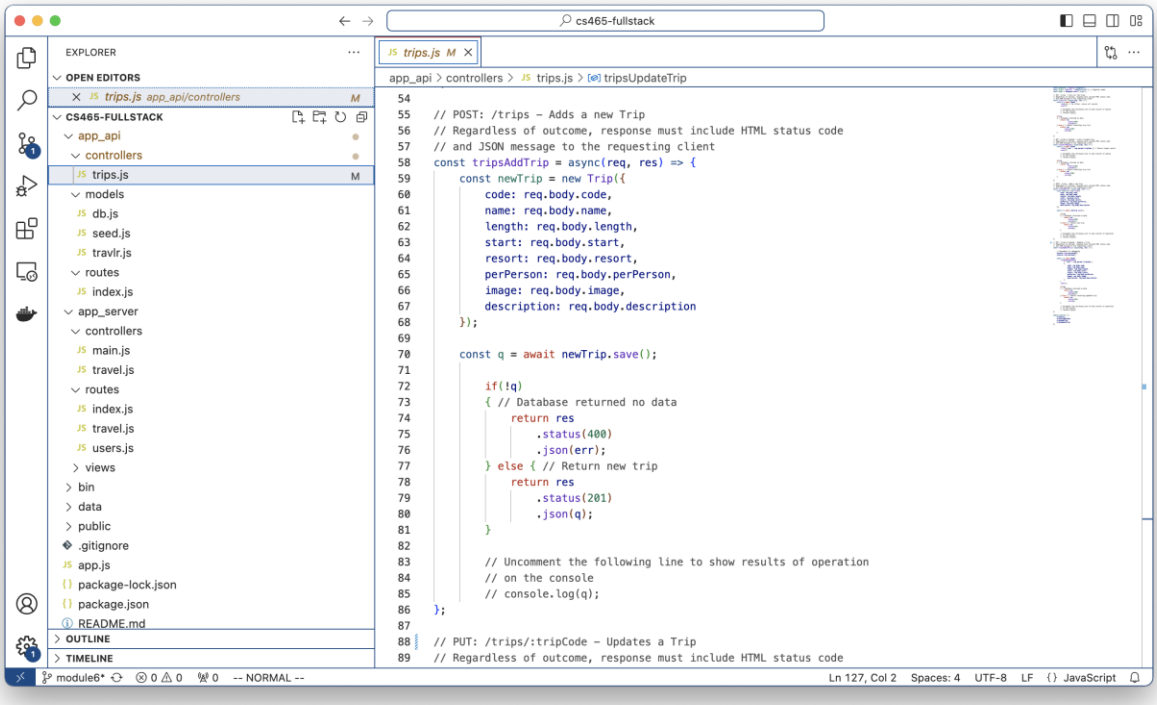
However, selecting the Add-Trip button and completing the form will generate a 404 error because the backend does not know how to treat the post command.

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***.

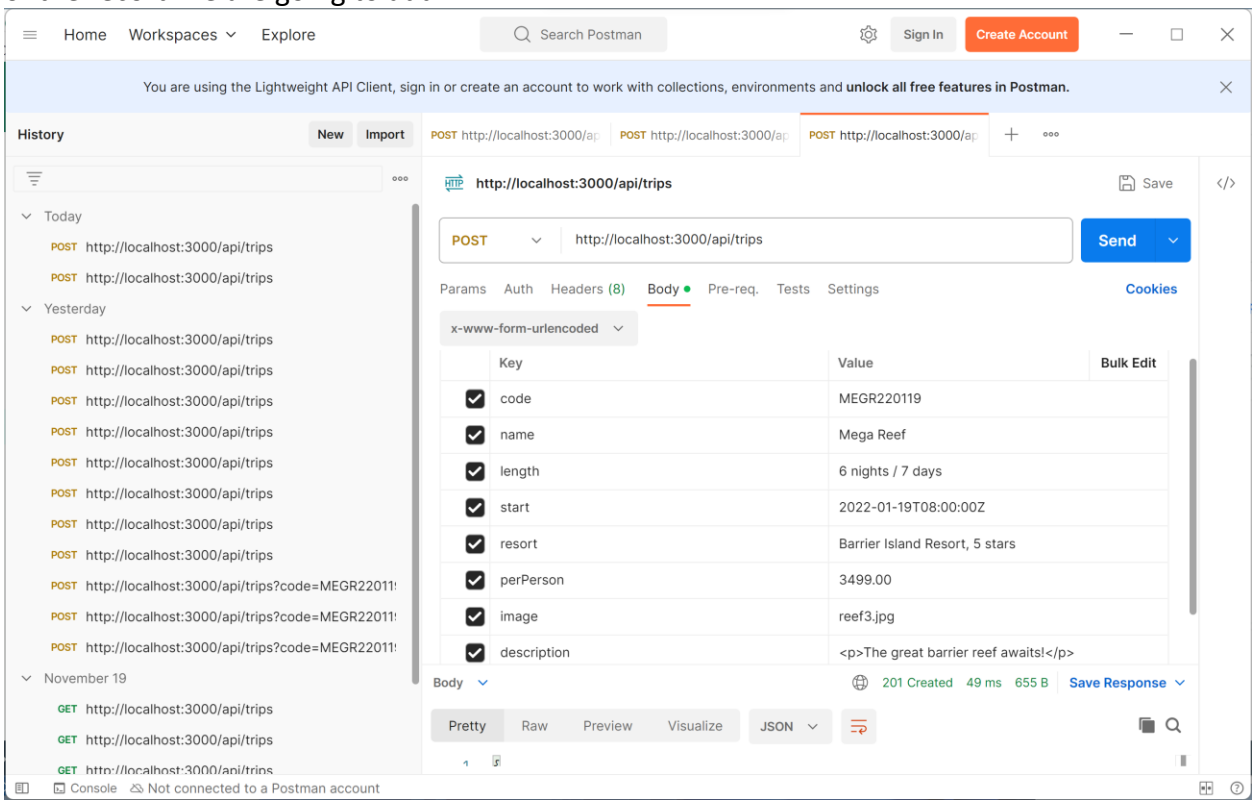


12. Now we need to modify the tripsController to handle the data that we are passing back within the request body.

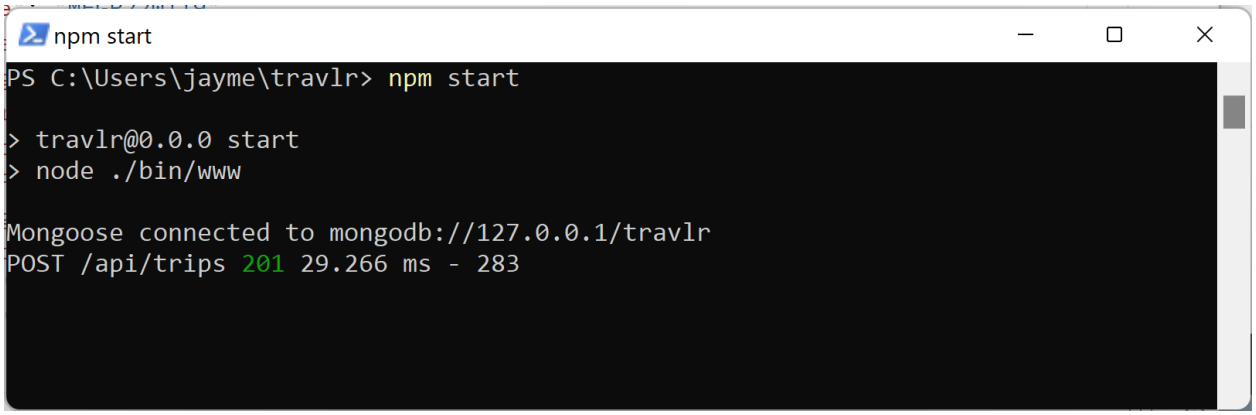


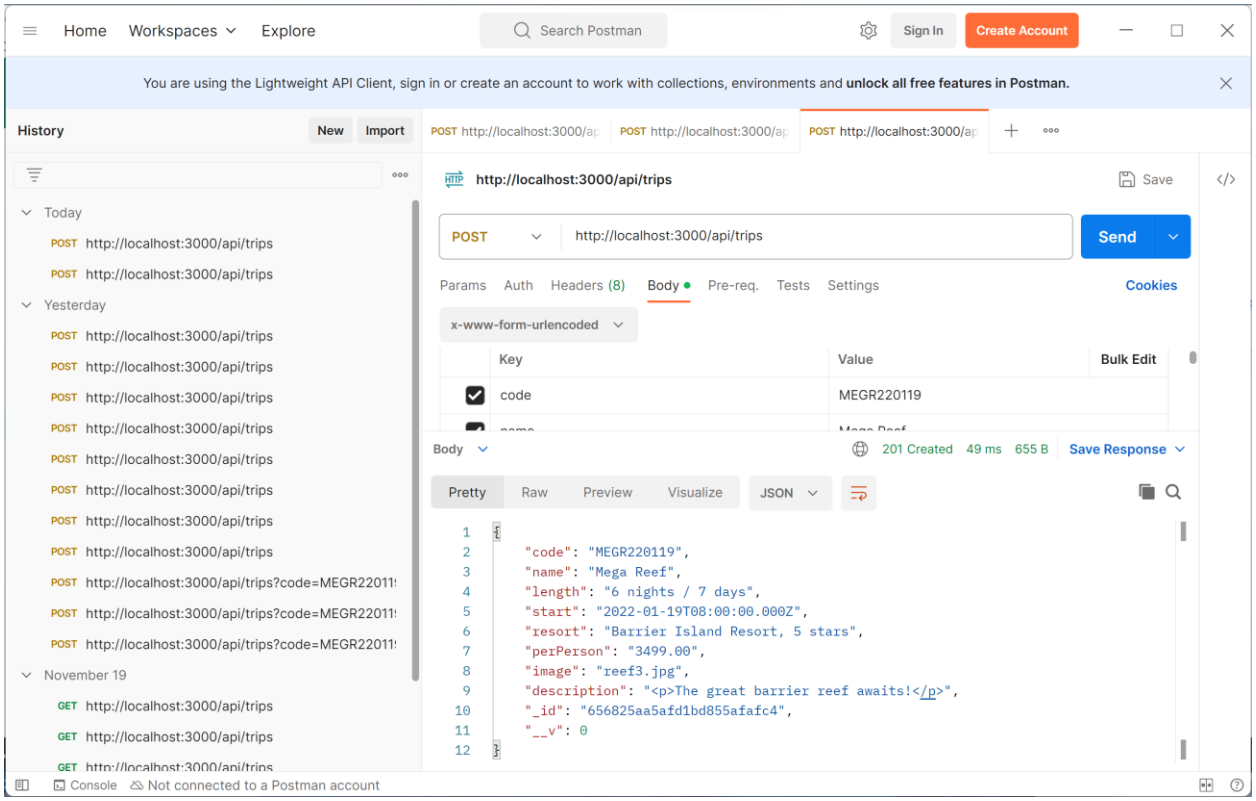
Make sure you add your new method to the modules.exports list!

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.

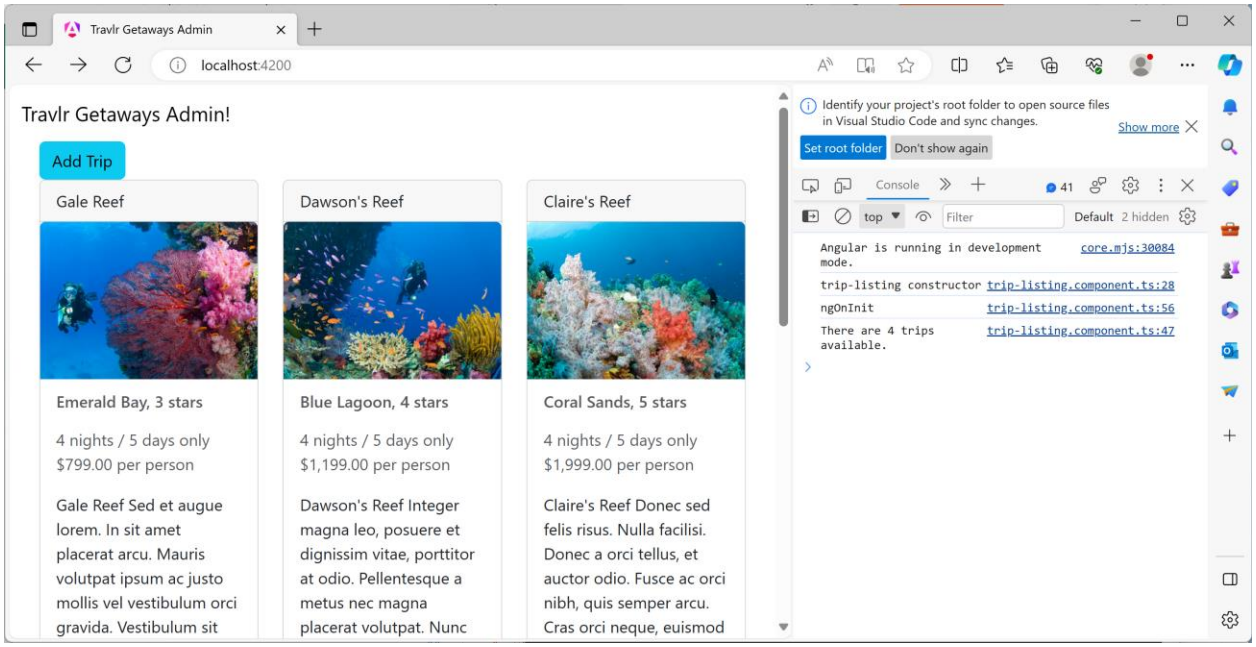


Once you have verified that you have added each field (and that they are spelled correctly) you can press the 'send' button to bounce your query against the backend API endpoint. If everything is successful, the call will return a JSON object with the record that you just entered into the MongoDB, and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





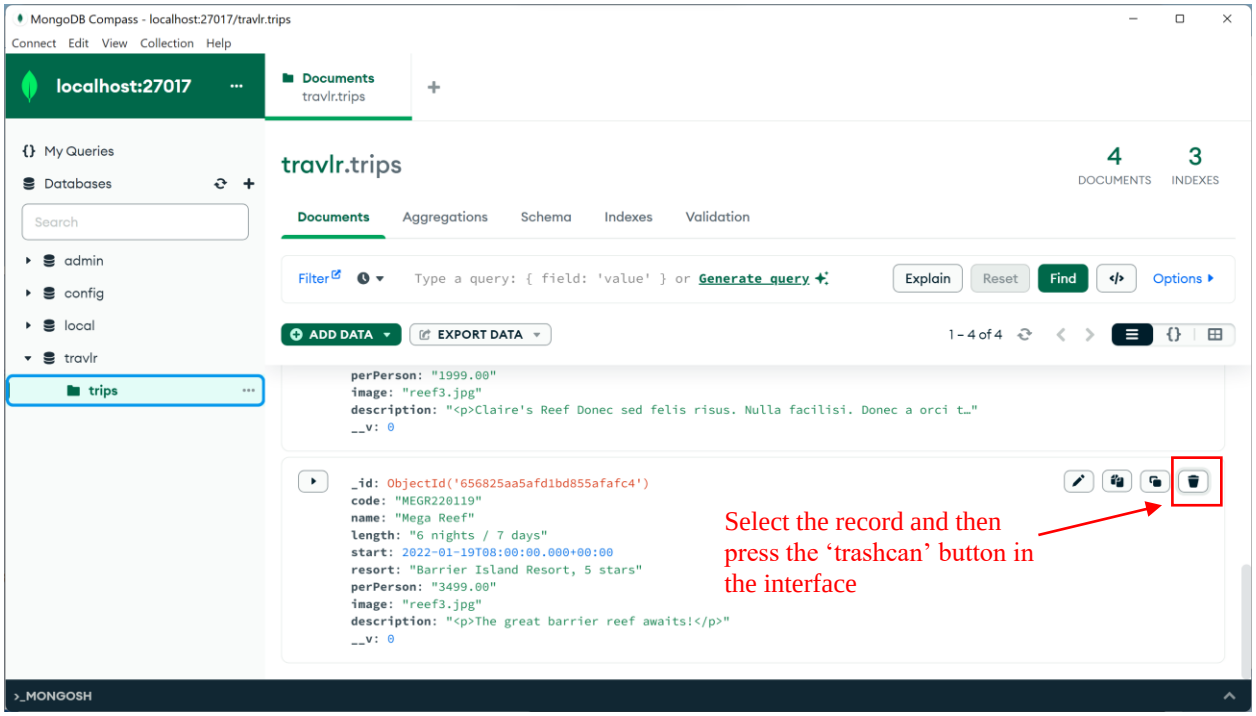
You should also be able to look at your front-end and see that there is now a fourth record available:

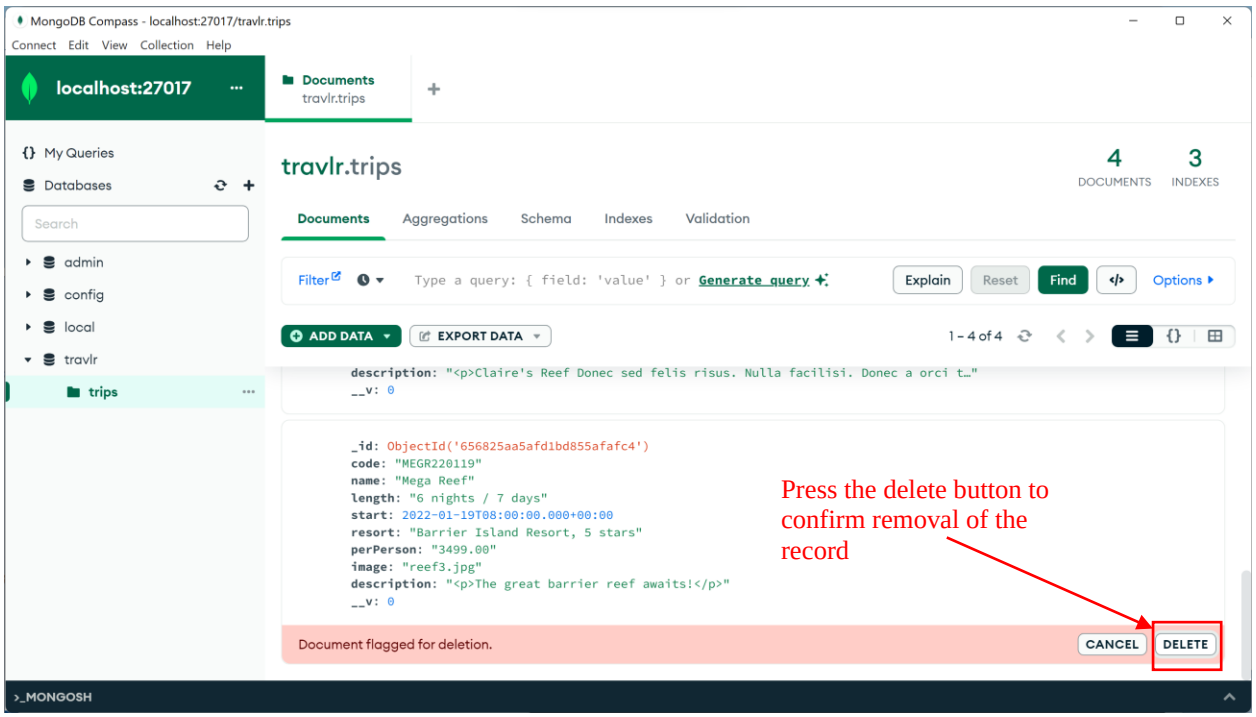


14. Now that we have shown that our test works directly with the back-end, we need to test to make certain that it also works when the request is submitted from our Angular JS front-end

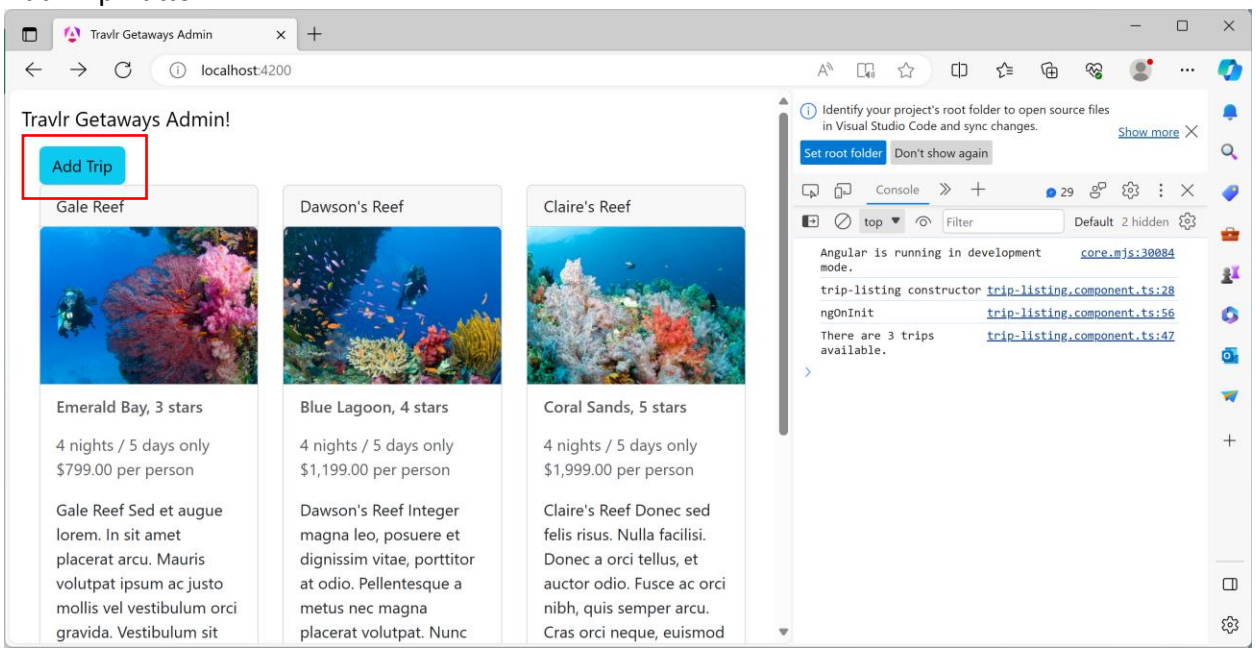
application. We have two choices here. We can remove the record we just added from the Mongo Database, or we can add a duplicate record. I am going to go through the process of removing the record from Mongo so that I can definitely determine the success of my test with the Angular application.

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

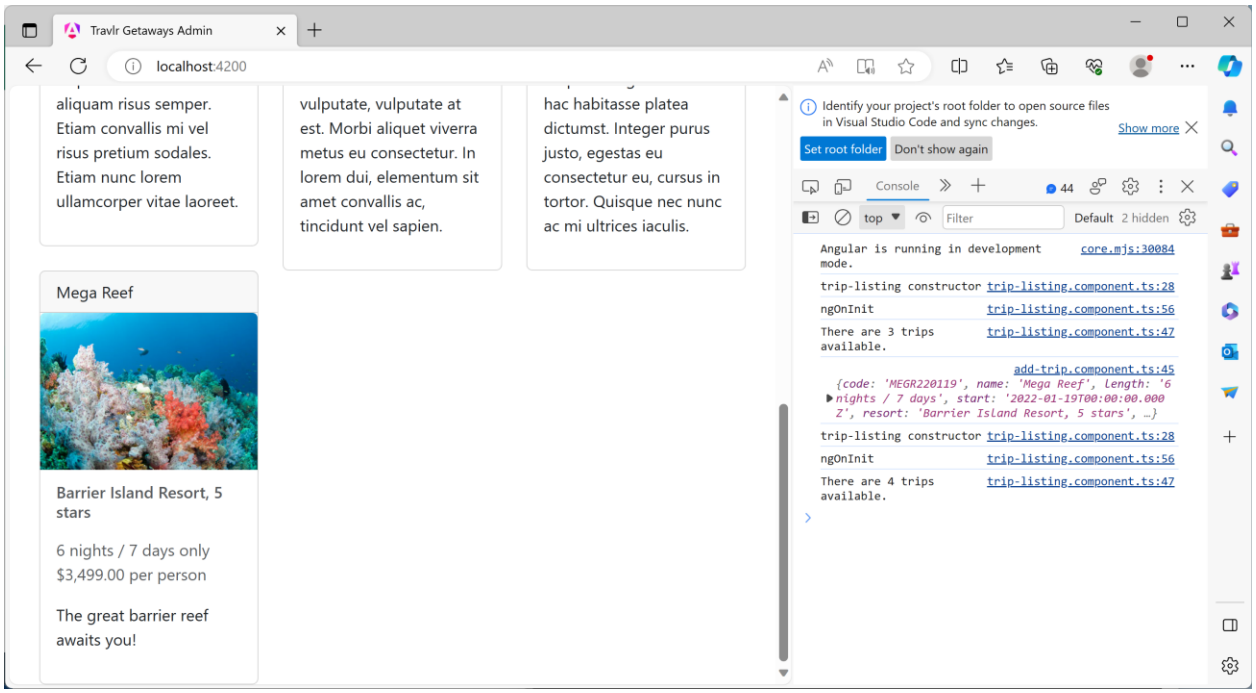




15. Now that we are back to having only three records in the database, let's go back to our Angular application and test our data entry form. From our Angular application, press the Add-Trip Button:



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.

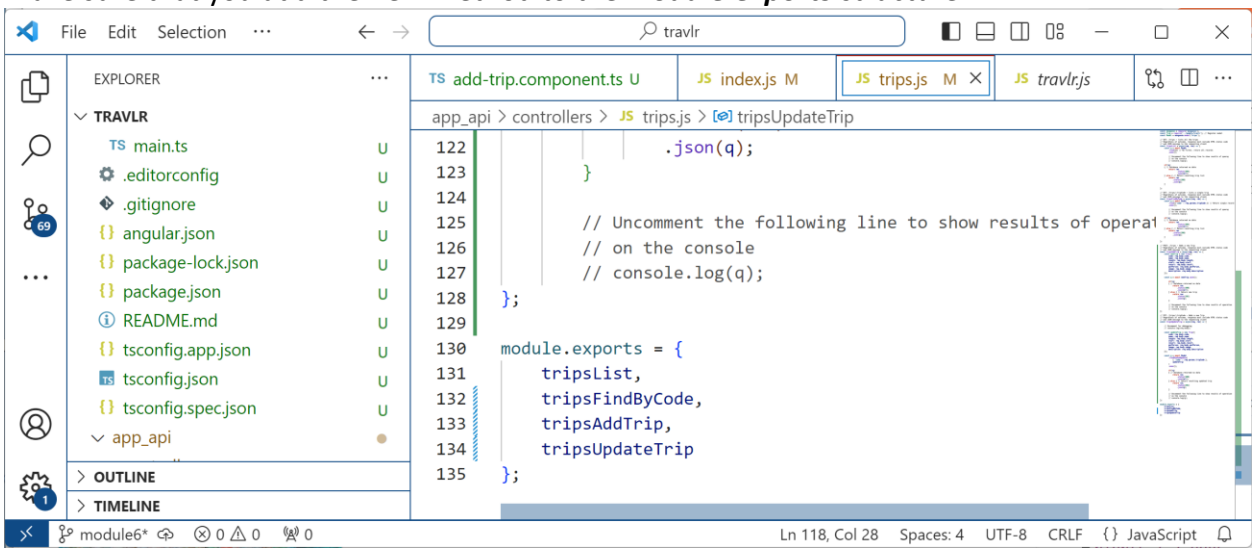
   We will add the following code to the file:

```
// 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);
};
```

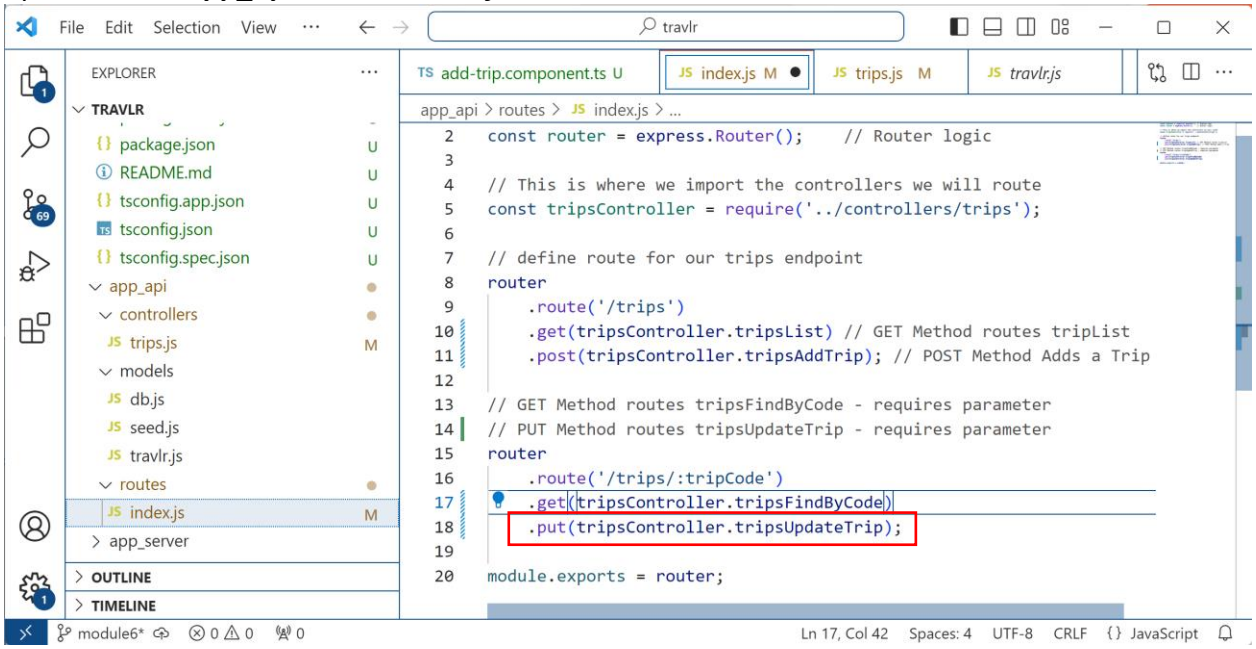
The biggest difference between the update method and the add method is that we are using the findOneAndUpdate method from Mongoose. This directs the database to locate the specified record (which we are selecting with a parameter for our API endpoint) and update the document based on the fields that 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:

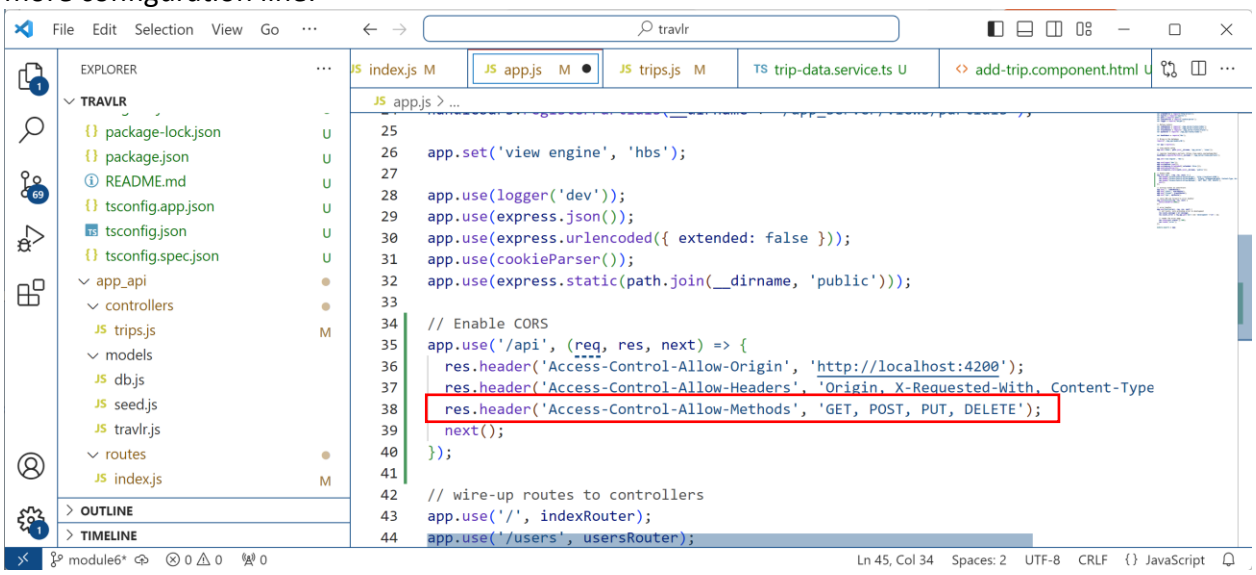


2. Now that we have our new method constructed, we need to add a route so that the Express application can find the method to call when it recieves a PUT request. This requires an

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:

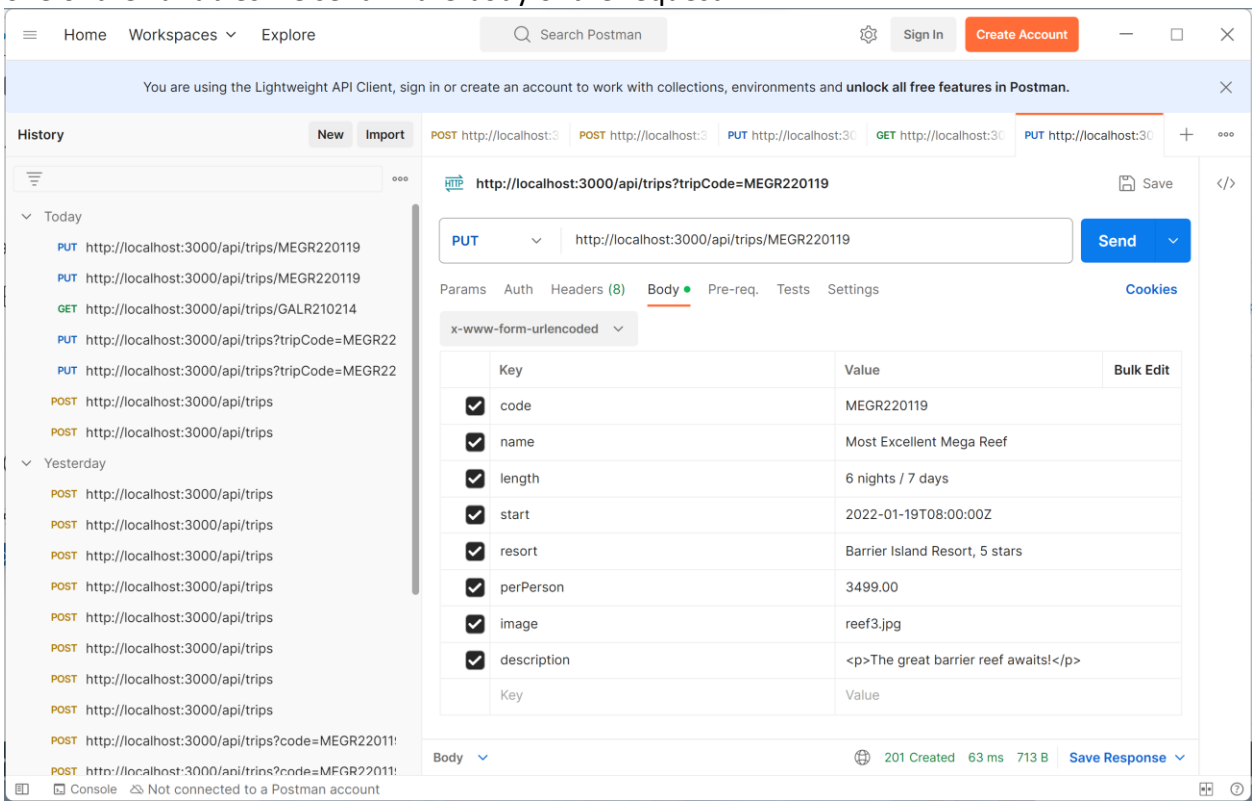


We added one line to our definition of our /api endpoint to enable access to the additional HTTP verbs.

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

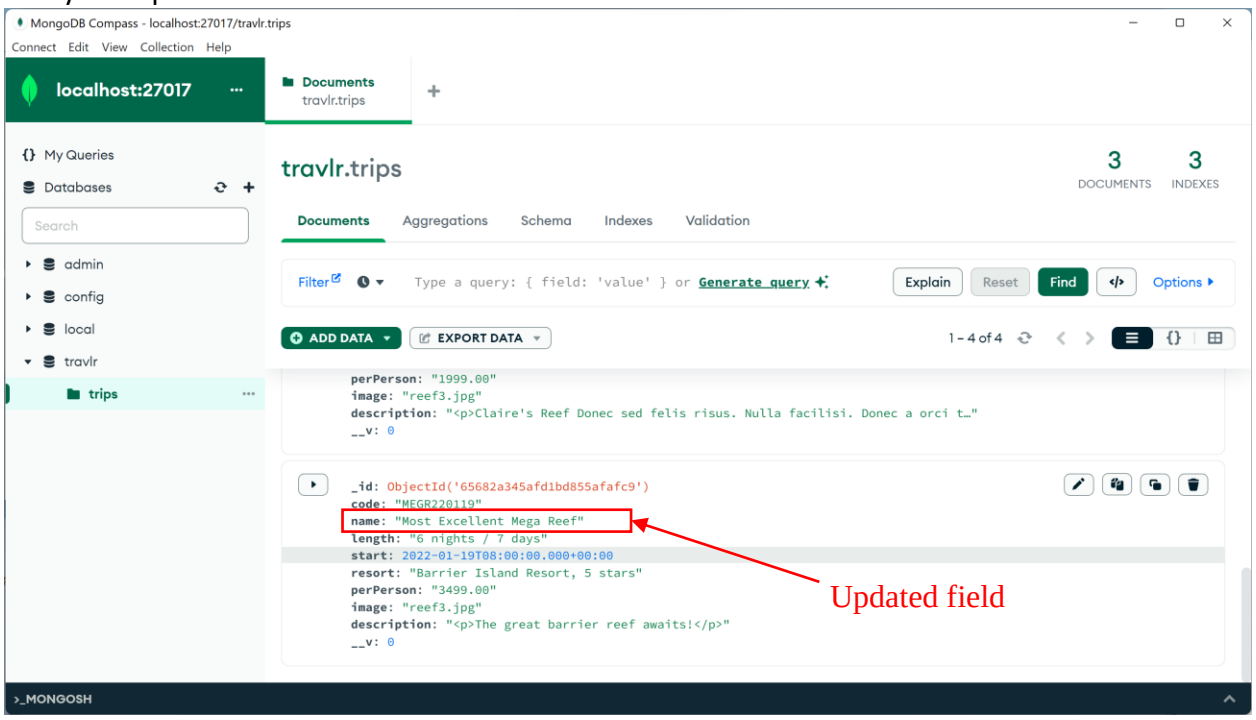
Once you have completed this step, please restart your Express backend server process.

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.



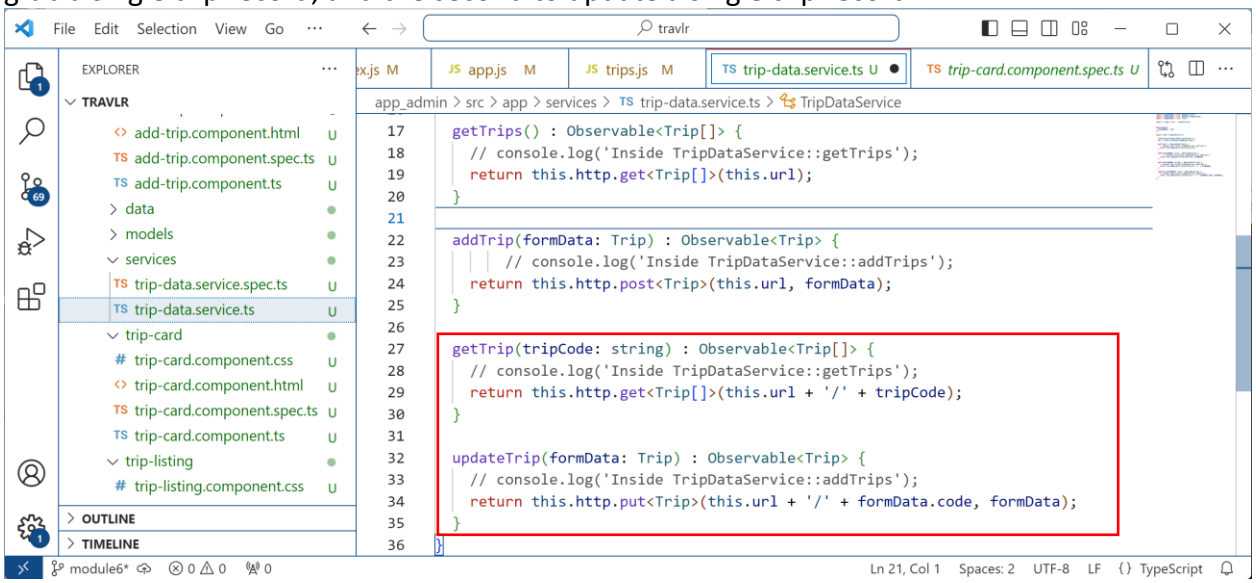
Once you have prepared the query, press the 'send' button and test your API call. You should get a '201 Created' message at the bottom of the window, and if you expand that section you will see the original record that you just updated.

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!

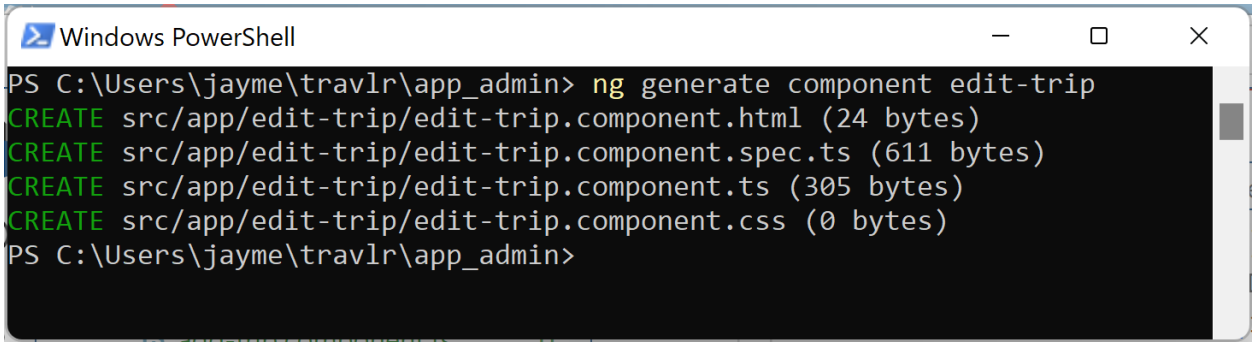
5. Now that we have the backend for the update functionality built, we need to go back and make the adjustment for the Angular application to provide a front-end interface to support updates. We will start by adding two additional methods to our ***TripDataService*** the first to grab a single trip record,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



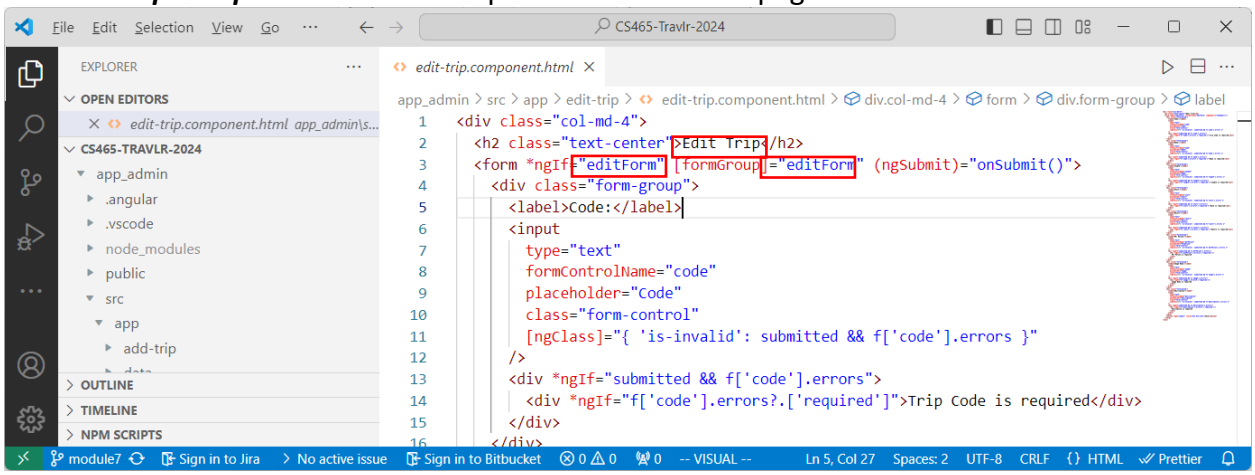
Please Note: For both of these methods we had to make sure we extended the URL to support the addition of the parameter for the tripCode that will specify an individual record.

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***.

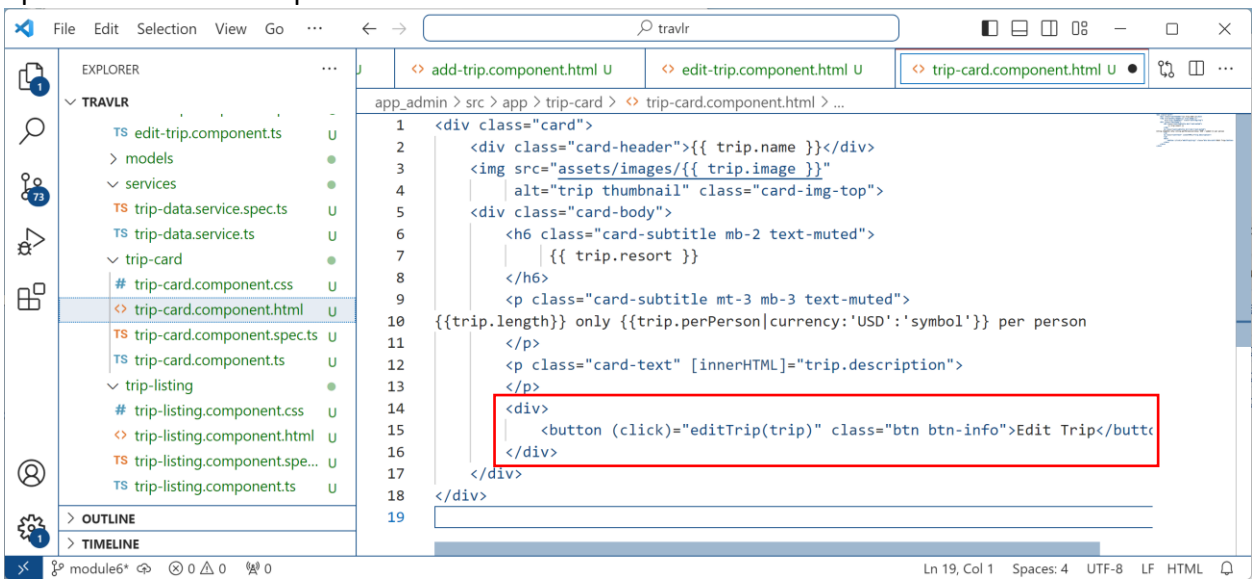
```
ng generate component edit-trip
```



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.

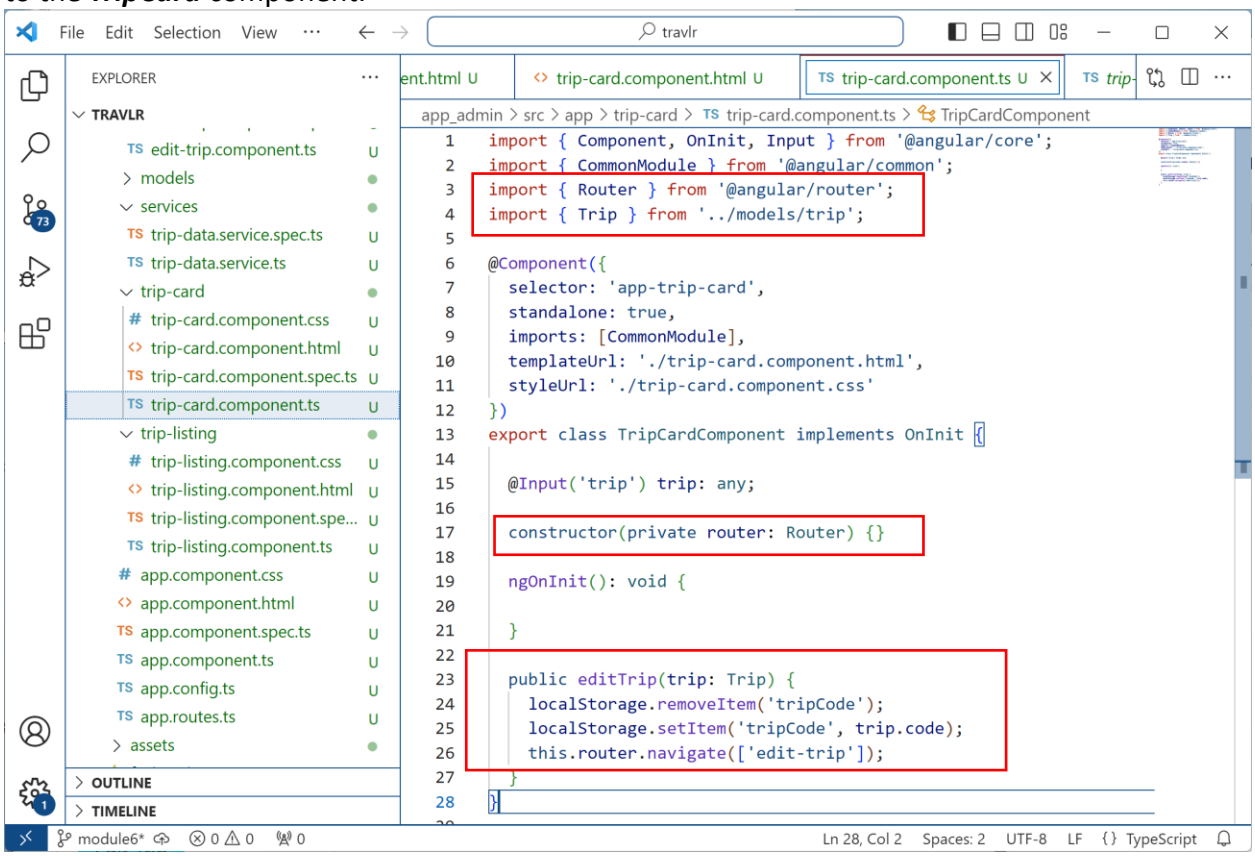


8. Now we will add an edit button to the bottom of the trip-card so that it renders the 'edit' option with each card presented.



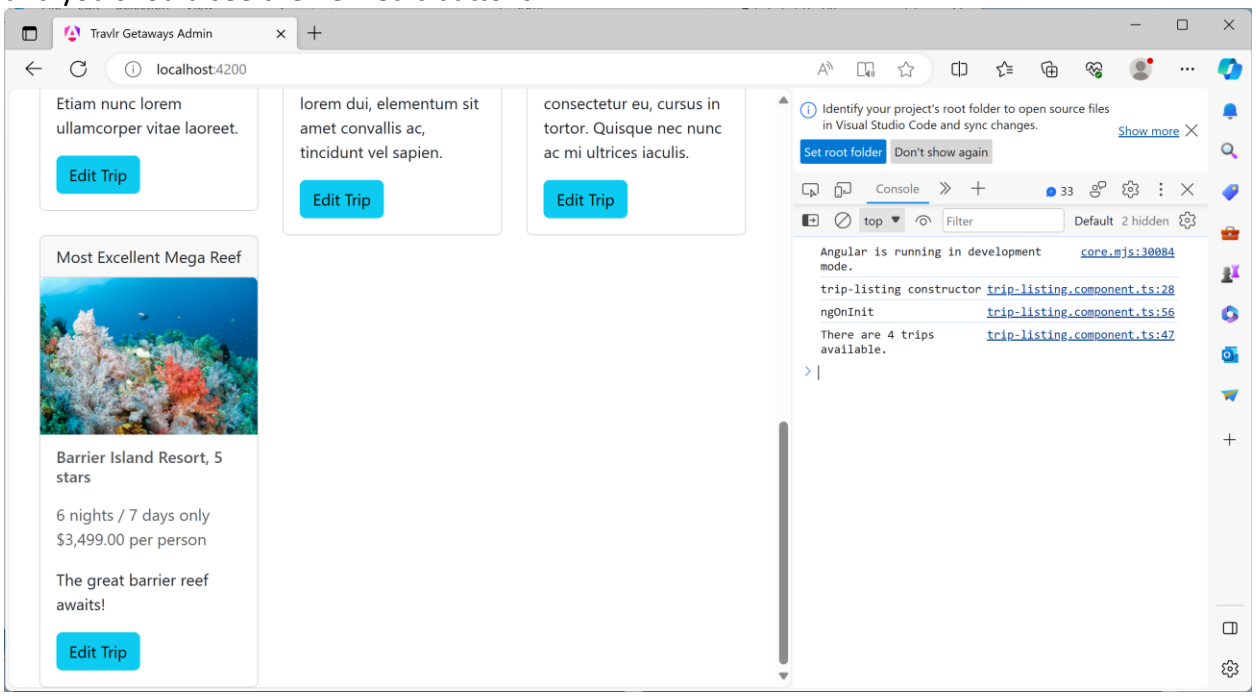
When the Edit Trip button is pressed the editTrip method will be called with a parameter of the trip displayed in the current trip-card.

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



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.

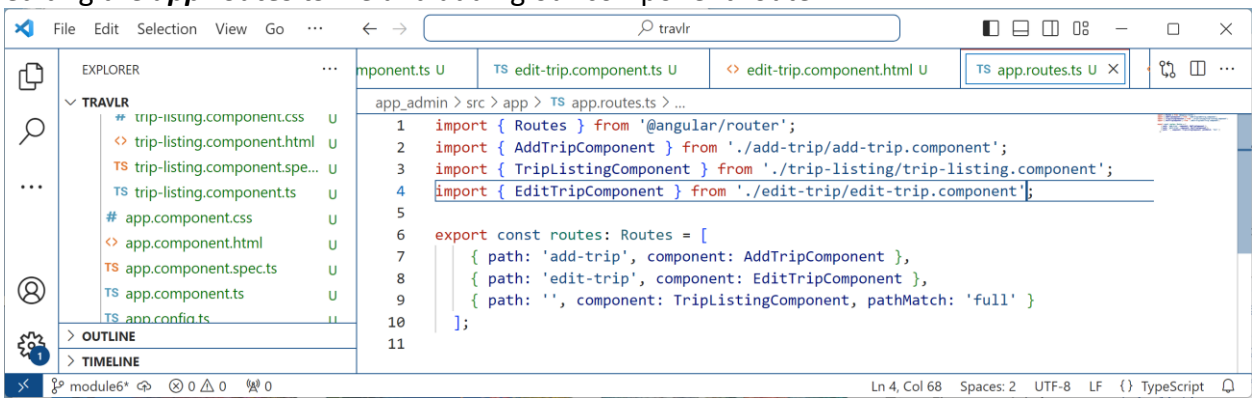
```
  public onSubmit()
  {
    this.submitted = true;

    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);
        }
      })
    }
}
```

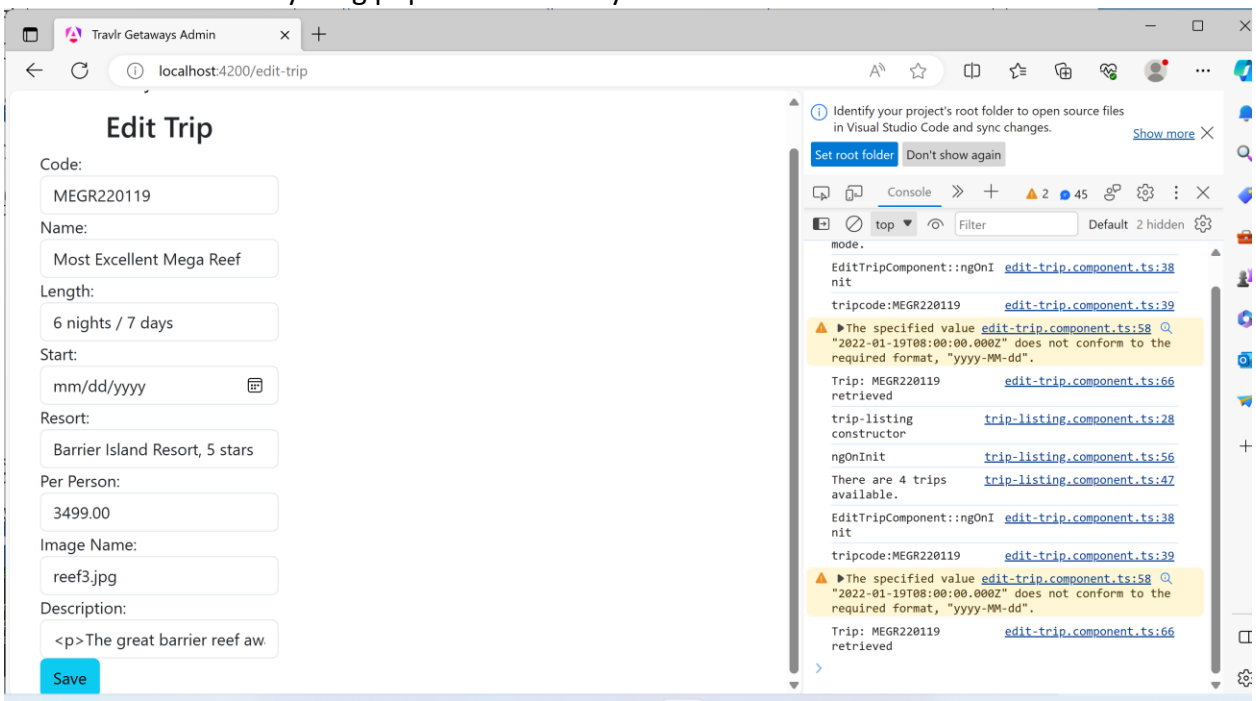
h. We need to add a quick access method to get at the form fields. This will be identical to the method we built in the ***add-trip*** component.

```
// 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.



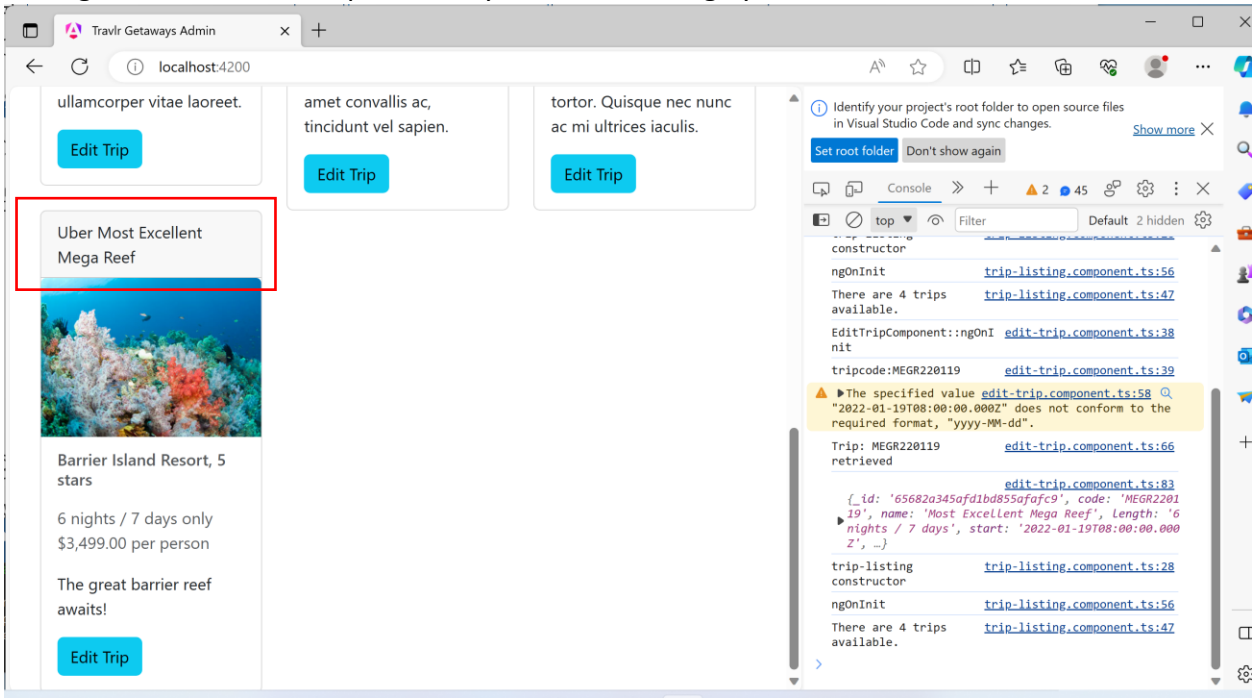
12. Now we can go back to our application and see what we have. We will select one of our trips to edit and see if everything populates correctly:



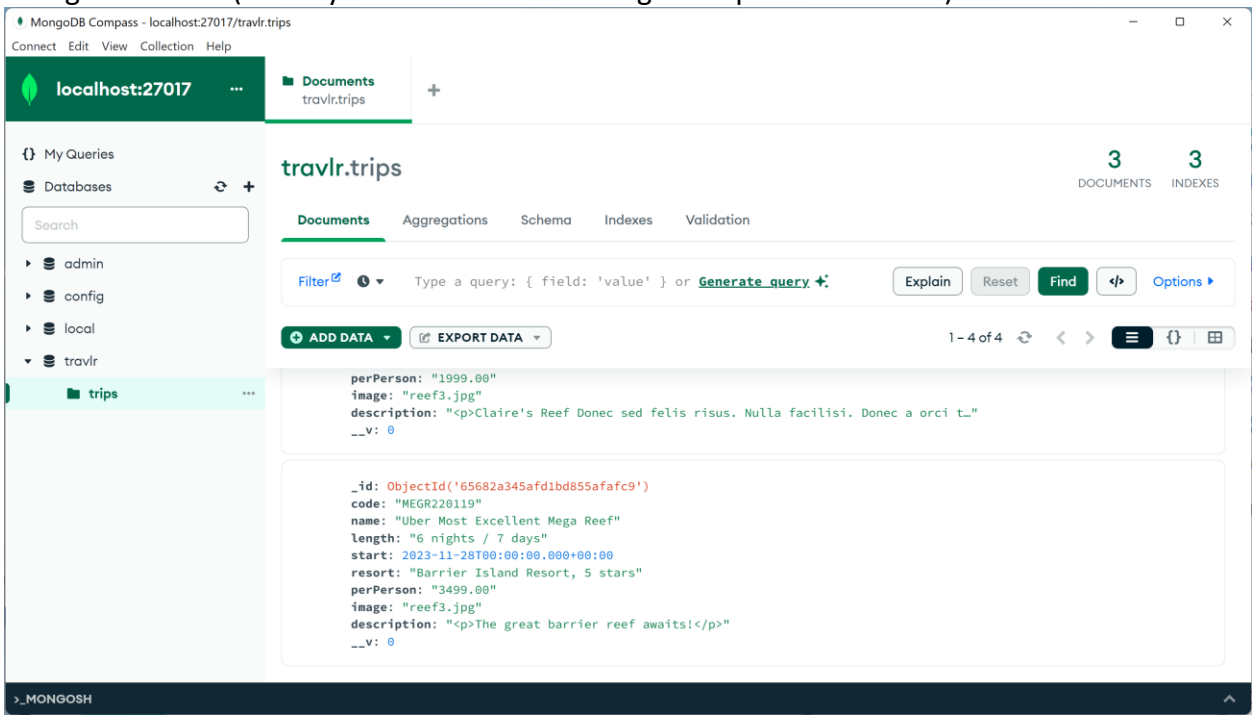
You will notice an error in the console because the value that we receive from the database is in a different format than the simple date widget we selected for our panel. It is a useful exercise to think about how you would address the difference between the data and the form. As a challenge activity – 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?

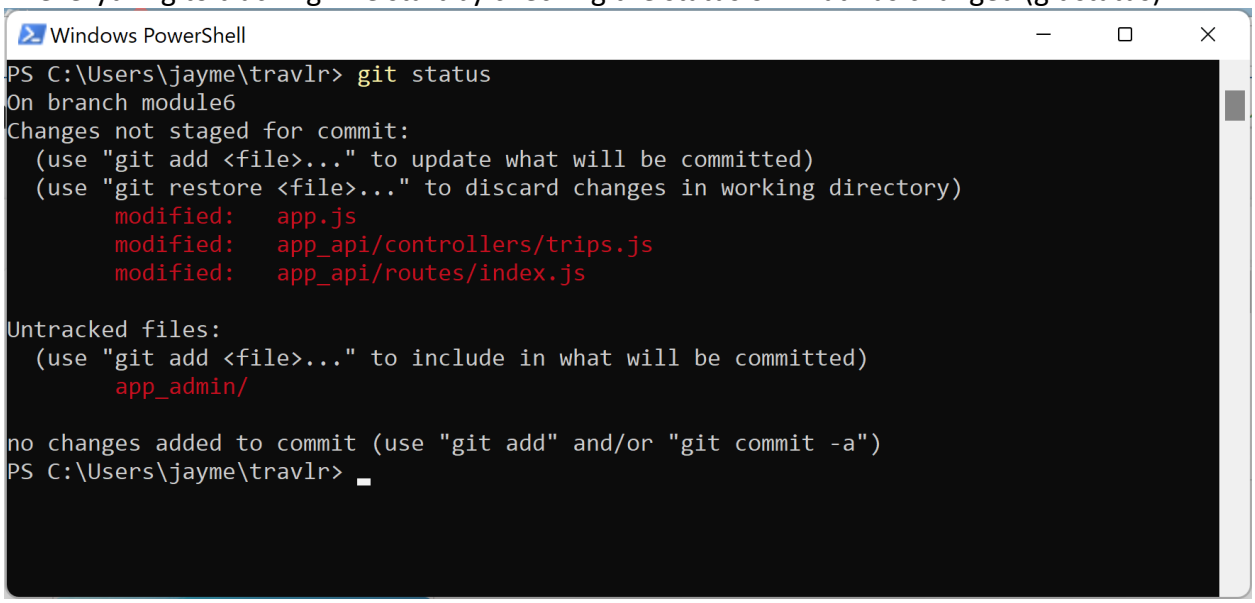


You can clearly see the edit that we made reflected in the page. If you close your browser and re-connect, you can see that the change is persistent and it has been saved in the Mongo Database (which you can also see in Mongo Compass or DBeaver).

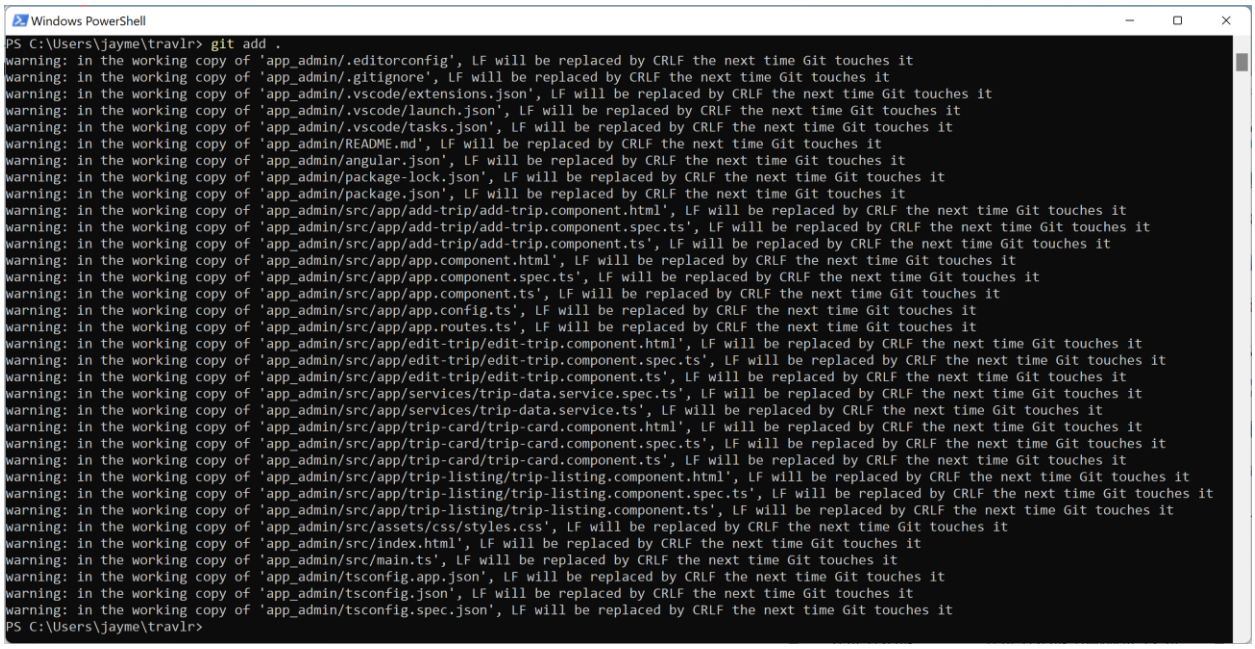


Finalizing Module 6

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'`):