Southern New Hampshire University

CS 465 Full Stack Guide

Table of Contents

Module 1: Intro, Setup, and Static HTML Site

Create Website

1. Open a Windows PowerShell command prompt and ensure you are in the top folder of your **travlr** local Git repository directory. Use the following:

```
cd ~/travlr
```

2. 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
```

```
> npm install -g express-generator
C:\Users\     \AppData\Roaming\npm\express -> C:\Users\     \AppData\
s-cli.js
+ express-generator@4.16.1
added 10 packages from 13 contributors in 0.734s
```

   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
```

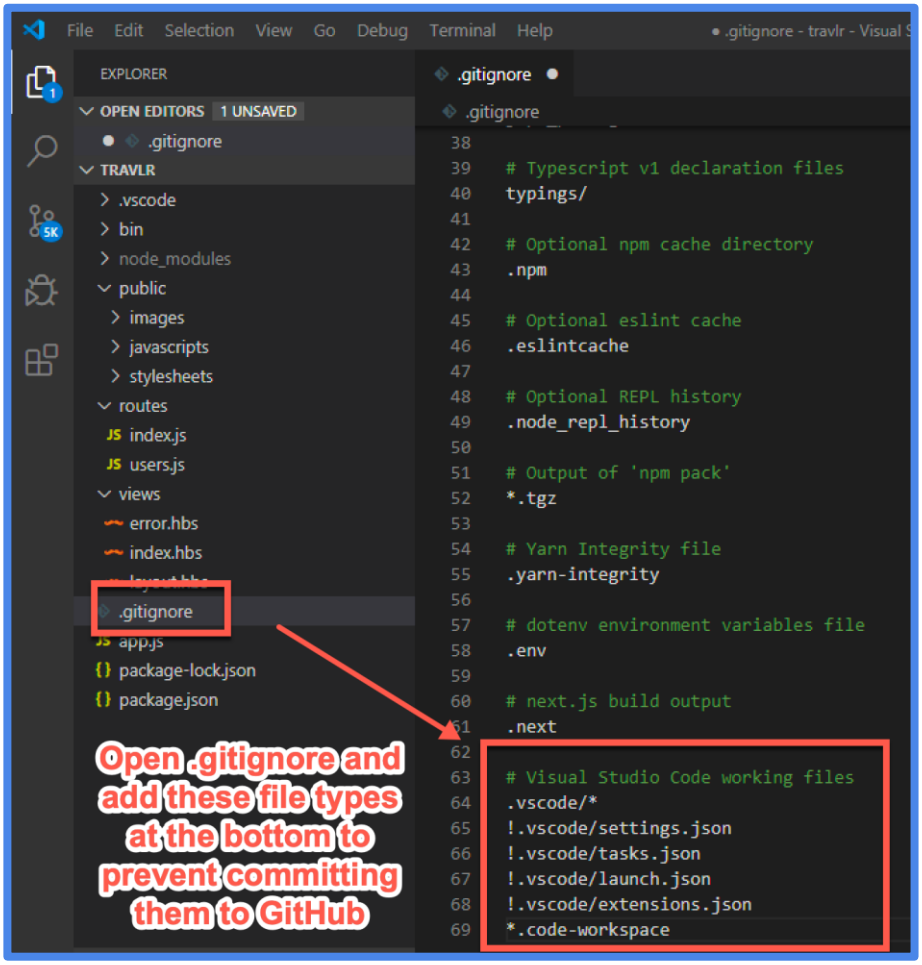
```
> express --view=hbs --git --force

   create : public\
   create : public\javascripts\
   create : public\images\
   create : public\stylesheets\
   create : public\stylesheets\style.css
   create : routes\
   create : routes\index.js
   create : routes\users.js
   create : views\
   create : views\error.hbs
   create : views\index.hbs
   create : views\layout.hbs
   create : .gitignore
   create : app.js
   create : package.json
   create : bin\
   create : bin\www

   install dependencies:
     > npm install

   run the app:
     > SET DEBUG=travlr:* & npm start
```

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:

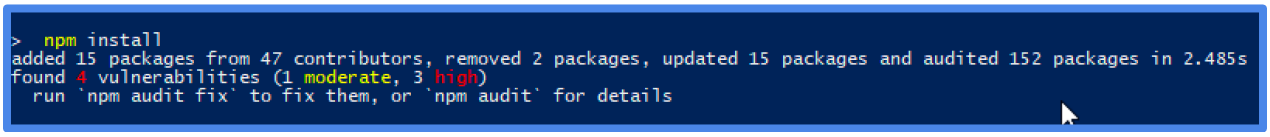
```
.vscode/*
!.vscode/settings.json
!.vscode/tasks.json
!.vscode/launch.json
!.vscode/extensions.json
*.code-workspace
```



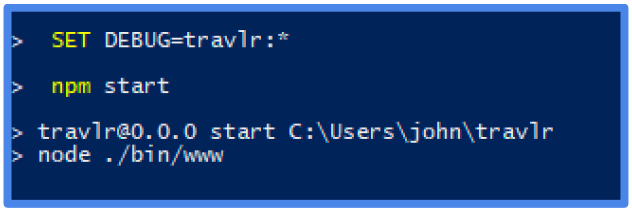
Remember to save this file after inserting the code lines!

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:

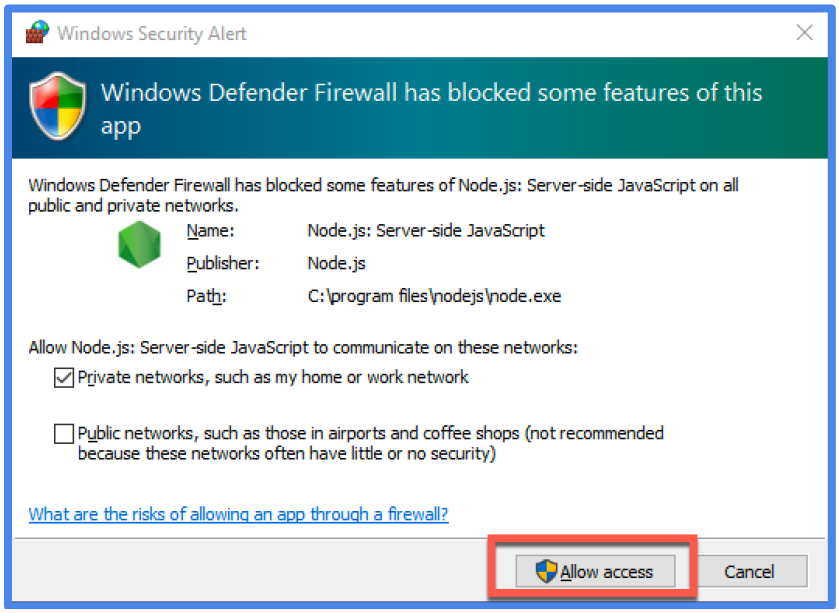
```
npm install
```



6. Start the webserver using the instructions displayed earlier when Express generated the website:
   a. `SET DEBUG=travlr:*`
   b. `npm start`

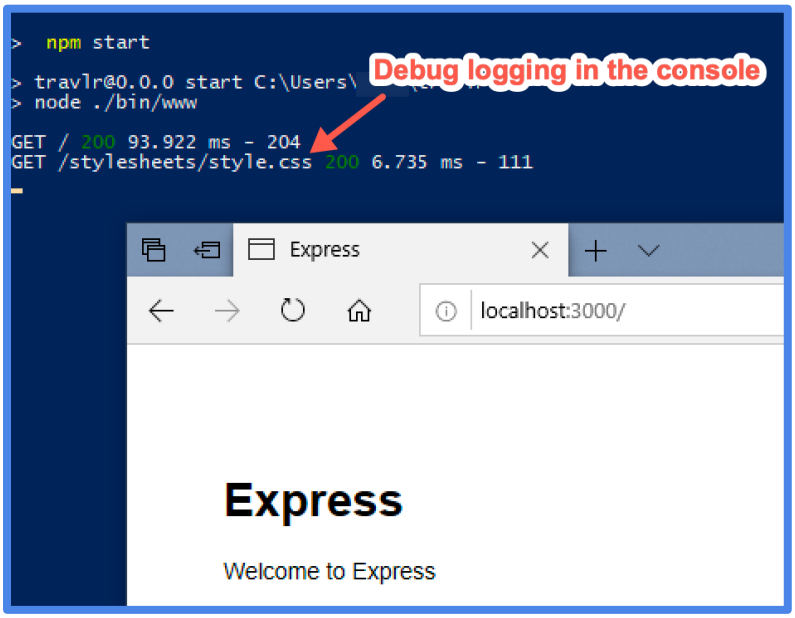


7. The first time you launch the web server, a Windows Defender Firewall dialog will be displayed. You must click **Allow access** in order for the website to run correctly.



Open a browser and navigate to the following URL:

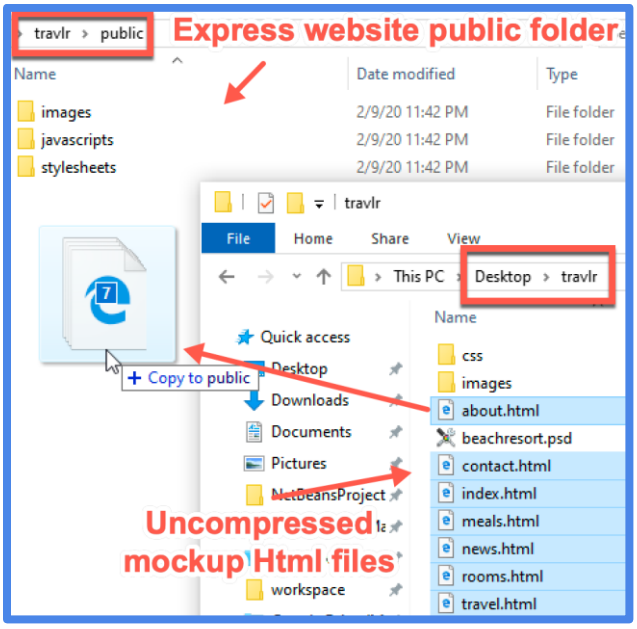
```
http://localhost:3000
```



You can close the browser and then type **Ctrl+C** in the command window to stop the web server.

Install Static Web Files

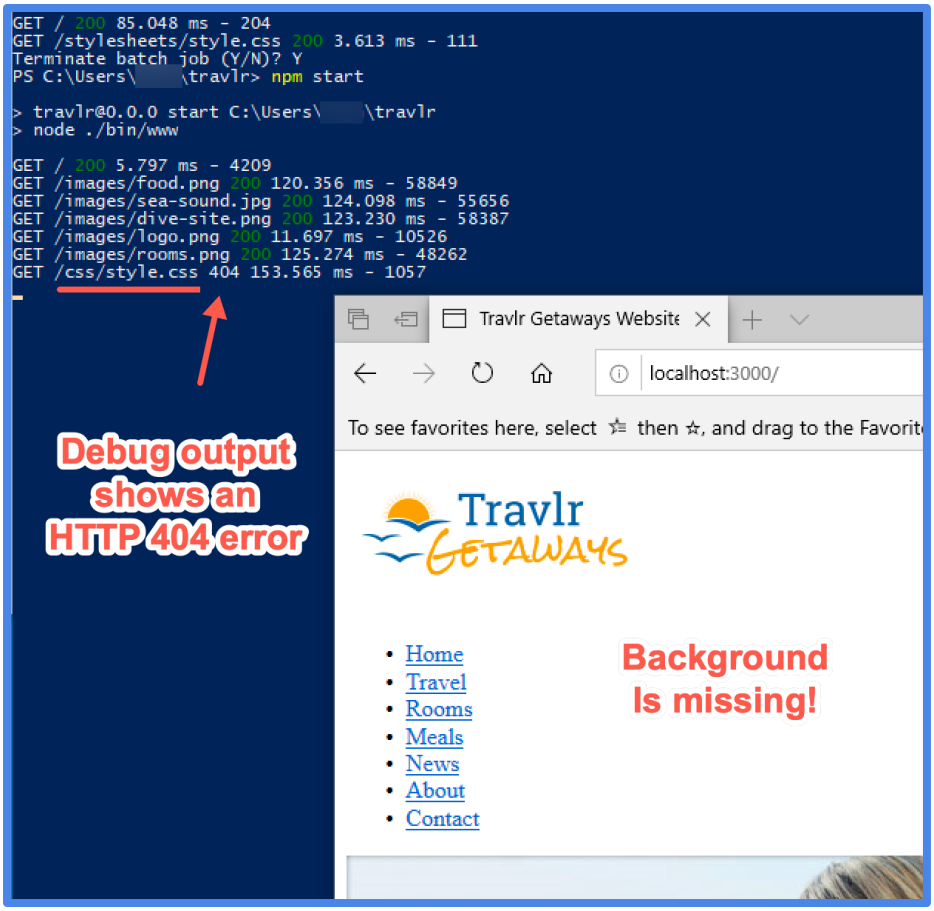
8. Copy the mock website content you downloaded from the course files into the public folder of the Express website.
    a. Open a second Windows File Explorer window and arrange it so you can see both windows.
    b. Select the **.html** files and drag them to the public folder while holding the right mouse button. Drop the files (by letting go of the right mouse button) over the public folder and choose **Copy files**.



9. Repeat the process to copy the image files into the **images** folder.

10. Copy the stylesheet from the **CSS** folder in the mockup site to the **stylesheets** folder on the Express website.

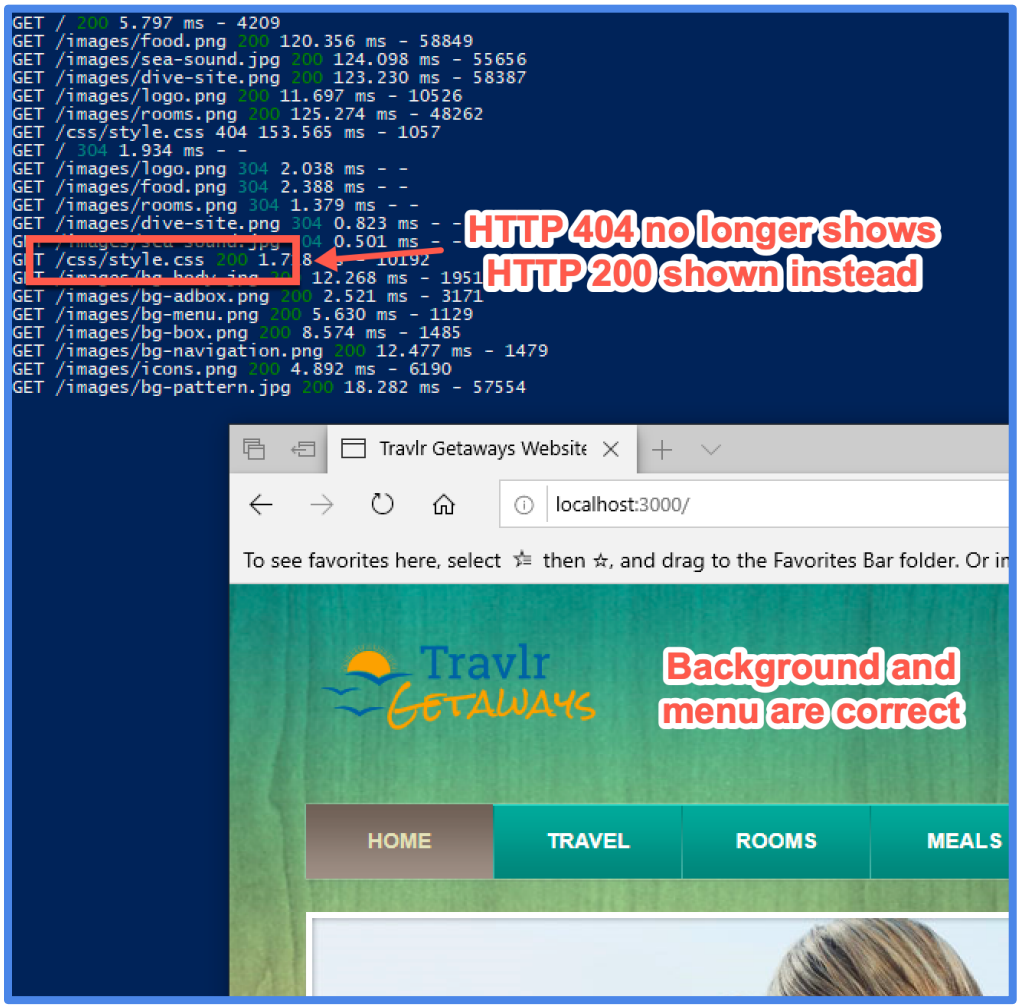
11. Launch the Express web server by switching to the command window and running the command `npm start` again.



Notice the output is not what was expected:

a. The page background image is missing.
b. The font is Times New Roman.
c. The menu is a vertical bulleted list and not horizontal.
d. The Node debug output shows an HTTP 404 error when retrieving the stylesheet.

12. In VS Code, right-click on the **stylesheets** folder and choose **Rename** to change the name to "css" and then refresh the browser.



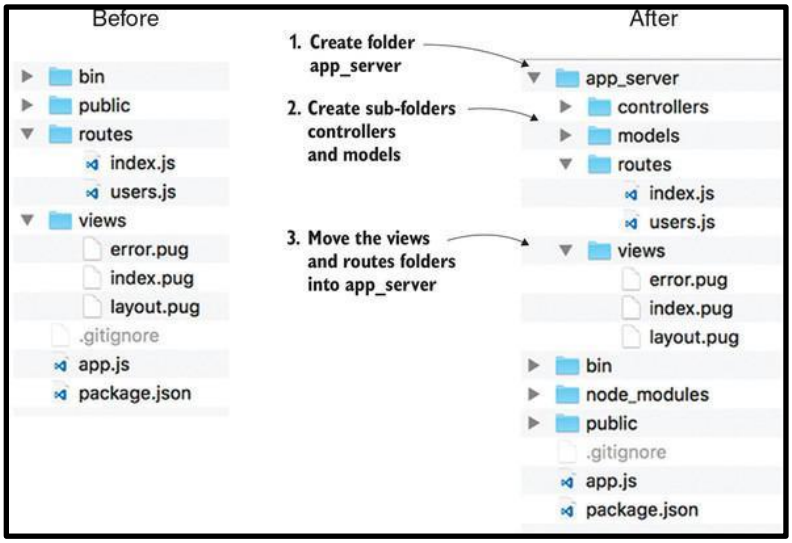
Note: Express assumes a default location for the CSS file in a **stylesheets** sub-folder under **assets**, while the mock website uses “css”. It is quicker to simply rename the default folder than to go through and edit every reference.

At this point, you have a Node.js webserver running that is serving the static HTML site content to your web browser.

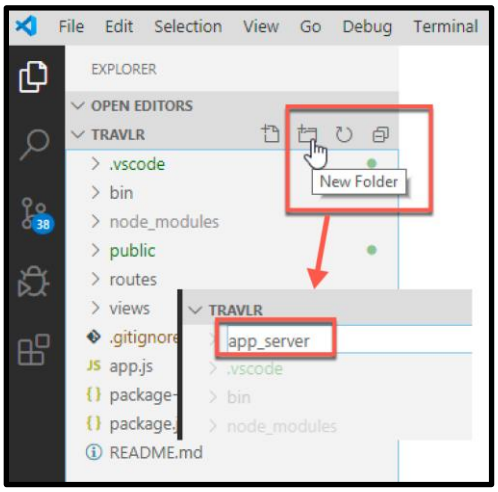
Module 2: MVC Routing

Create Web Application Folders

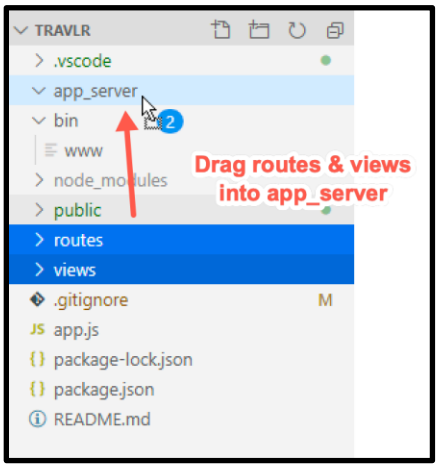
1. Create an **app_server** folder where we'll migrate the static HTML to Handlebars template views, and refactor common functionality such as headers and footers into "partials". This section corresponds to Chapter 3.3 in your textbook. Refer to Figure 3-6 in the textbook to visualize the reorganization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



2. In VS Code, create a new folder within the **travlr** project called **app_server**.

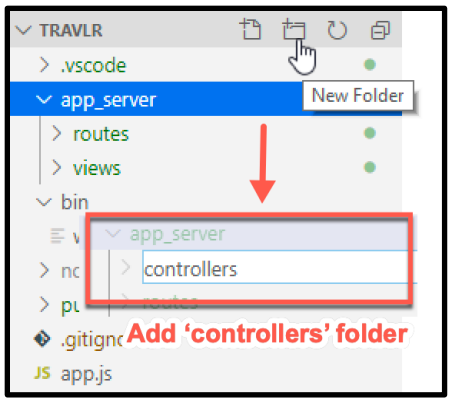


3. Move the existing **routes** and **views** folders into the newly created **app_server** folder.

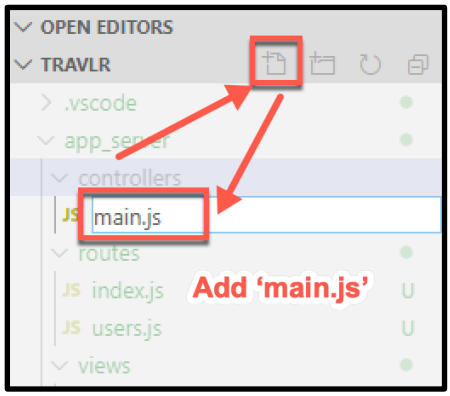


Create Controllers and Routes

4. With the **app_server** folder selected, click the **New Folder** icon and add a new folder named "controllers".

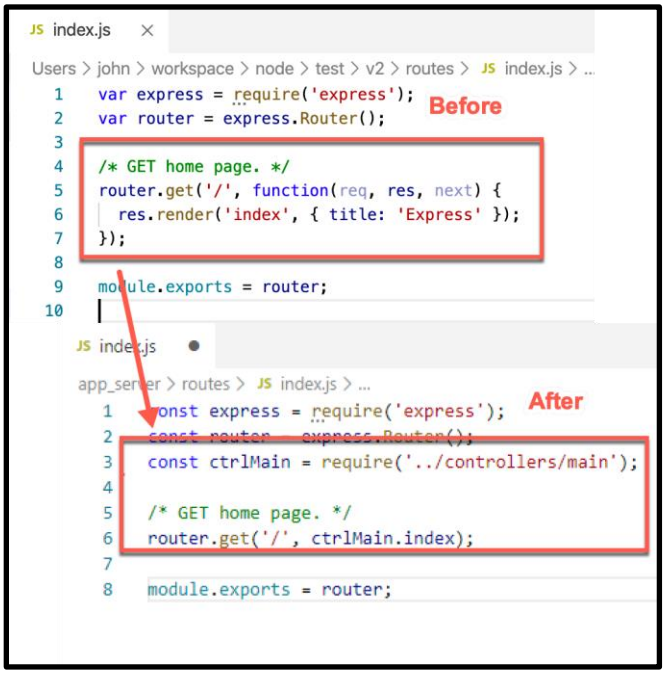


5. Select the newly created **controllers** folder, then click the **Add File** icon to add a new "main.js" file that will be our first controller.



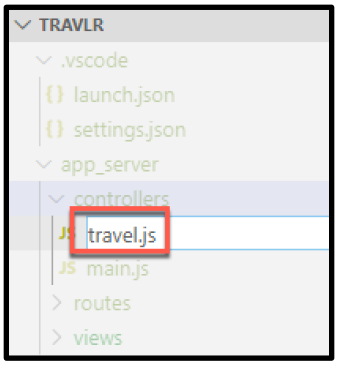
6. Edit the **main.js** file in the **controllers** folder to make it serve the initial "index" page.

7. Edit the **index.js** file in the **routes** folder to pass the request for the site default starting page (known as the root or "/") over to the new main controller.




```
const ctrlMain = require('../controllers/main');
/* GET home page. */
router.get('/', ctrlMain.index);
```

8. Create a controller for the Travlr page.



9. Add code to the Travlr controller.

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

10. Create a route for the Travlr page.


```
const express = require('express');
const router = express.Router();
const controller= require('../controllers/travel');

/* GET home page. */
router.get('/', controller.travel);

module.exports = router;
```

11. Edit the **app.js** file to insert the new **app_server** folder into the path and add a new route for **'/travel'**, wiring it to the new Travlr controller.

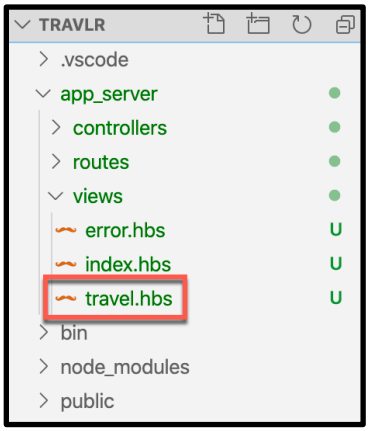
```
JS app.js > ...
1  const createError = require('http-errors');
2  const express = require('express');
3  const path = require('path');
4  const cookieParser = require('cookie-parser');
5  const logger = require('morgan');
6
7  const indexRouter = require('./app_server/routes/index');
8  const usersRouter = require('./app_server/routes/users');
9  const travelRouter = require('./app_server/routes/travel');
10
11 const app = express();
12
13 // view engine setup
14 app.set('views', path.join(__dirname, 'app_server', 'views'));
15 app.set('view engine', 'hbs');
16
17 app.use(logger('dev'));
18 app.use(express.json());
19 app.use(express.urlencoded({ extended: false }));
20 app.use(cookieParser());
21 app.use(express.static(path.join(__dirname, 'public')));
22
23 app.use('/', indexRouter);
24 app.use('/users', usersRouter);
25 app.use('/travel', travelRouter);
26
27 // catch 404 and forward to error handler
28 app.use(function(req, res, next) {
29   next(createError(404));
30 });
```

insert 'app_server' in path

add new travelRouter

Create Handlebars Views

12. Copy the static **travel.html** to the **views** folder and rename the extension to ".hbs" to indicate this will be a Handlebars view.



13. Notice that all the static HTML pages have the same code for a page header and page footer.

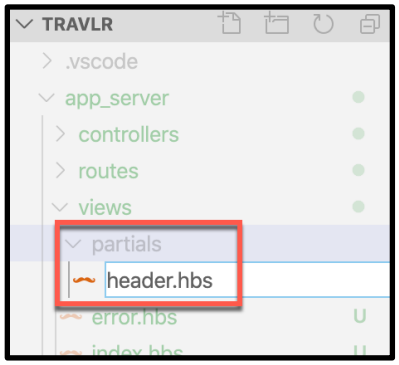
travel.hbs

app_server > views > travel.hbs > ...

```
<!DOCTYPE html>
<!-- Website template by freewebsitetemplates.com -->
<html>
<head>
    <meta charset="UTF-8">
    <title>{{title}}</title>
    <link rel="stylesheet" href="css/style.css" type="text/css">
</head>
<body>
    <div id="background">
        <div id="page">
            <div id="header">
                <div id="logo">
                    <a href="index.html"><img src="images/logo.png" alt="LOGO"
                </div>
                <div id="navigation">
                    <ul>
                        <li>
                            <a href="index.html">Home</a>
                        </li>
                        <li class="selected">
                            <a href="travel.html">Travel</a>
                        </li>
                        <li>
                            <a href="rooms.html">Rooms</a>
                        </li>
                        <li>
                            <a href="meals.html">Meals</a>
                        </li>
                        <li>
                            <a href="news.html">News</a>
                        </li>
                        <li>
                            <a href="about.html">About</a>
                        </li>
                        <li>
                            <a href="contact.html">Contact</a>
                        </li>
                    </ul>
                </div>
            </div>
            <div id="contents">
                <div class="box">
```

14. Handlebars supports creating partial page fragments to hold common HTML so it doesn't have to be coded in every page. To use this feature, create a **partials** folder inside the **views** folder.

15. In the **partials** folder, create a new file called "header.hbs".



16. Copy the previously shown lines of HTML for the heading into this new Handlebars template file.

Replace the anchor reference **travel.html** with just **/travel** so the view will be rendered using the updated route and controller when the link is clicked.

17. Back in the **traveler.hbs** file, replace the entire header “div” tag with a single-line Handlebars instruction to include the new partial file.

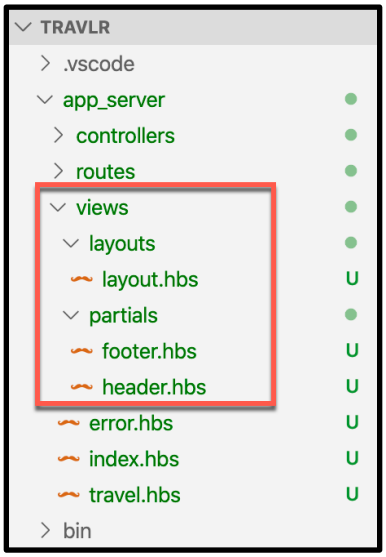
```
<body>
    <div id="background">
        <div id="page">
            {{> header}}
            <div id="contents">
                <div class="box">
                    <div>
                        <div class="body">
                            <h1>Travel</h1>
                            <ul id="sites">
```

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

Handlebars uses double curly braces to identify its code blocks from ordinary HTML. The greater than symbol is the instruction to the templating engine for including the header view; that is, to copy the contents of the **header.hbs** file into the **traveler.hbs** file, replacing the instruction tag.

18. Repeat this process to create a footer partial page named “footer.hbs”.

19. Create a “layouts” folder under views and move the **layout.hbs** file in there.



20. Edit **app.js** again to register the new partials folder.

```
11  const app = express();
12
13  // view engine setup
14  app.set('views', path.join(__dirname, 'app_server', 'views'));
15
16  // register handlebars partials (https://www.npmjs.com/package/hbs)
17  hbs.registerPartials(path.join(__dirname, 'app_server', 'views/partials'));
18
19  app.set('view engine', 'hbs');
20
21  app.use(logger('dev'));
22  app.use(express.json());
```

insert call to register the partials folder

```
// register handlebars partials
(https://www.npmjs.com/package/hbs)
hbs.registerPartials(path.join(__dirname, 'app_server',
'views/partials'));
```

At this point, you can see the handlebars rendered view.

Module 3: Static HTML to Templates With JSON

1. Replace the hard-coded HTML trip content with JSON data and a loop using Handlebars directives to render each trip.
    a. Here is the output of trips currently embedded as static HTML in the page:

    b. Create a new top-level "/data" folder in the site with a "trips.json" file that will contain the JSON versions of trips for testing purposes.

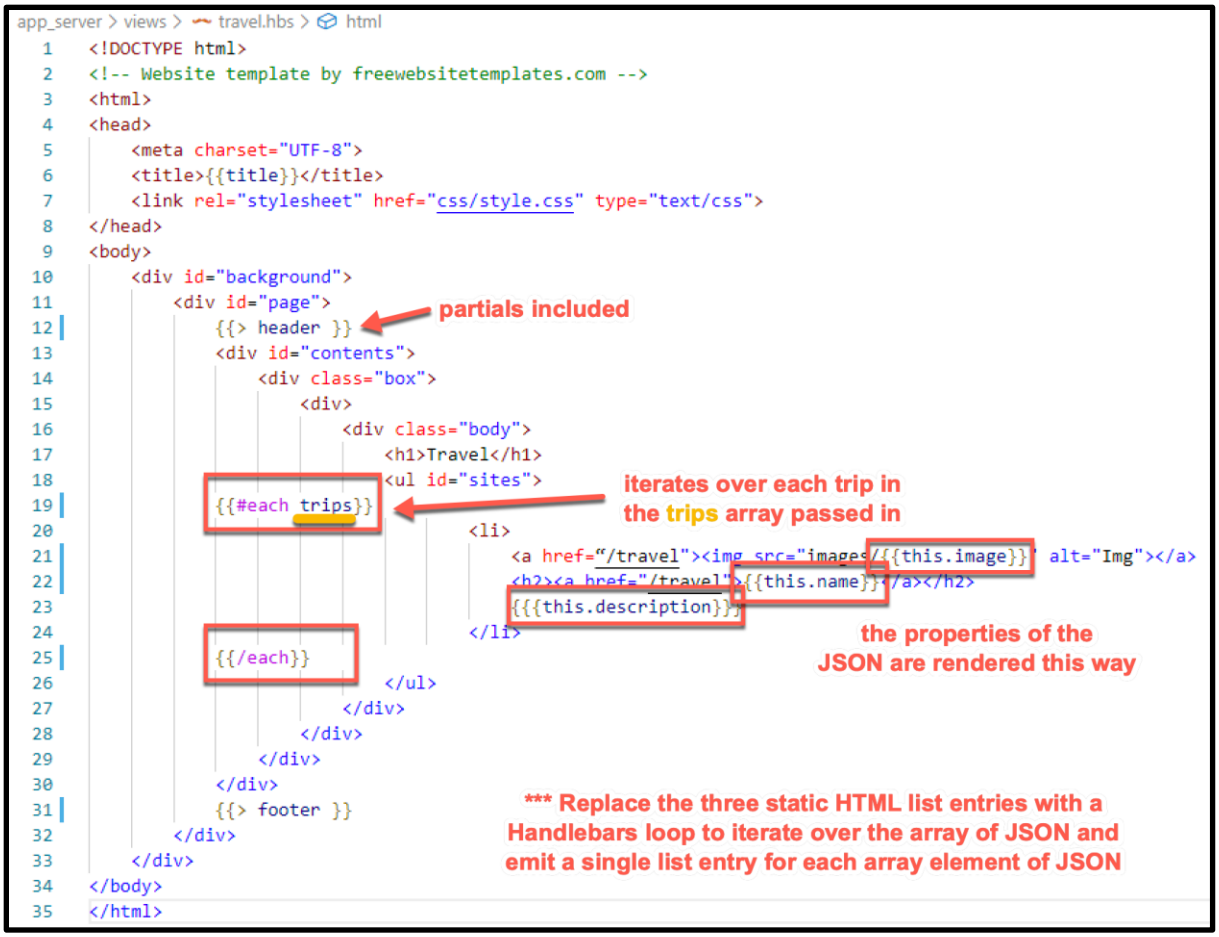
Notice that the HTML paragraph within "description" was tweaked slightly to contain the name of the trip embedded within, to easily determine when this JSON data is being rendered.

c. Edit the **travel.js** controller to use the built-in Node JS file system component with its **fs.readFileSync()** method to retrieve the JSON data just created.

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

Warning: Reading a JSON file every time a request is processed by the web server is **not** best practice. This is a quick and dirty technique during development to aid in rapid prototyping.

2. Edit the **travel.hbs** template and replace the three static HTML list entries with a **{{#each trips}}**...**{{/each}}** directive to create a loop over each JSON array element.




```
<!DOCTYPE html>
<!-- Website template by freewebsitetemplates.com -->
<html>
<head>
    <meta charset="UTF-8">
    <title>{{title}}</title>
    <link rel="stylesheet" href="css/style.css" type="text/css">
</head>
<body>
    <div id="background">
        <div id="page">
            {{> header }}
            <div id="contents">
                <div class="box">
                    <div>
                        <div class="body">
                            <h1>Travel</h1>
                            <ul id="sites">
            {{#each trips}}
                                <li>
                                    <a href="/travel"><img src="images/{{this.image}}" alt="Img"></a>
                                    <h2><a href="/travel">{{this.name}}</a></h2>
                                    {{{this.description}}}
                                </li>
            {{/each}}
                            </ul>
                        </div>
                    </div>
                </div>
            </div>
            {{> footer }}
        </div>
    </div>
</body>
</html>
```

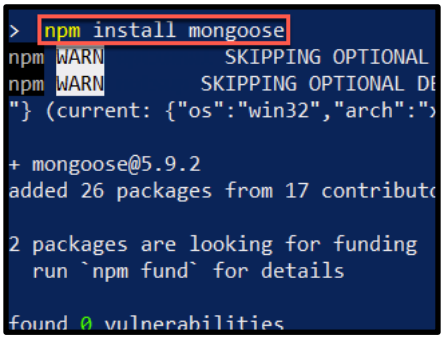
At this point, save everything, then start the Node server using the command `npm start` and test your work. The 120 lines of static HTML have been replaced with 35 lines, including a few Handlebars directives that allow the rendered page to be driven dynamically by the data passed into the template.

Optional: Repeat these steps to convert other static HTML pages to Handlebars templates, either with or without JSON data. Use your imagination and experiment!

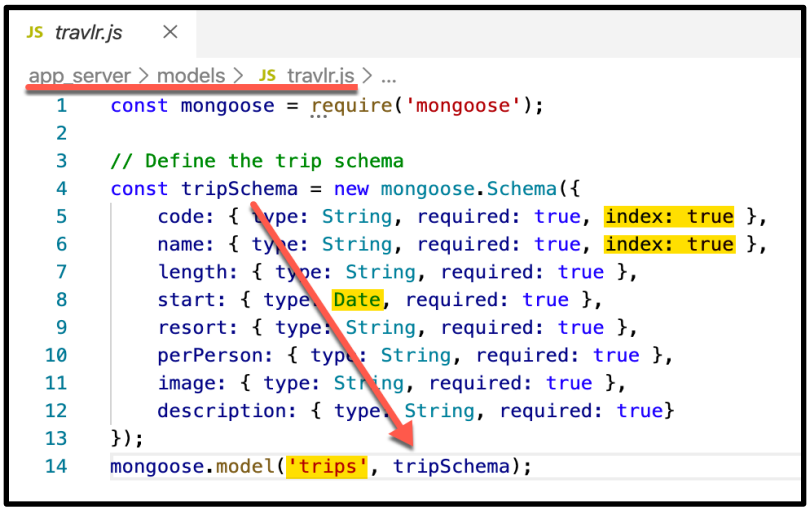
Module 4: NoSQL Databases, Models, and Schemas

Install and Configure Mongoose

1. Begin by installing Mongoose.



2. Create a new **models** folder under **app_server** and create a new module to hold the schema for a trip.



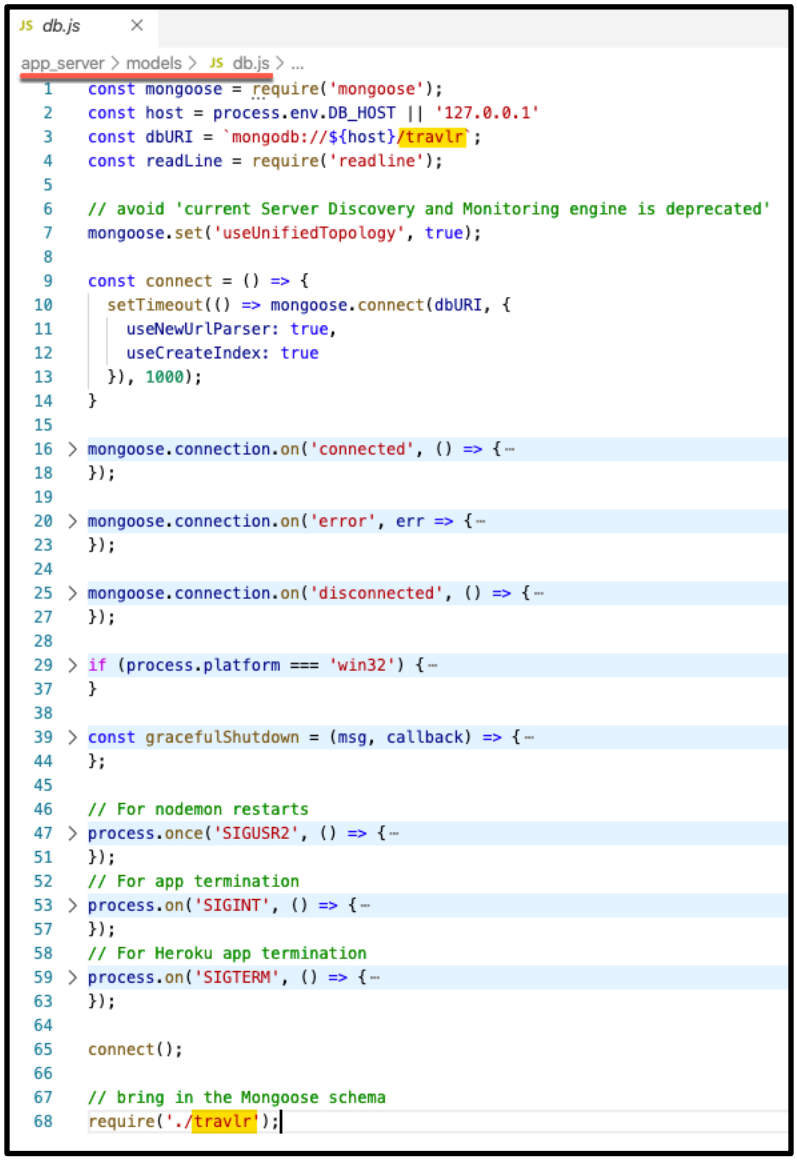
Notice the highlighted portions above. The trip code and name will be indexed in MongoDB for faster retrieval. The start date will be stored using standard ISO date format, and the collection will be named “trips”.

Here’s the code for the schema:

```
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}
});
mongoose.model('trips', tripSchema);
```

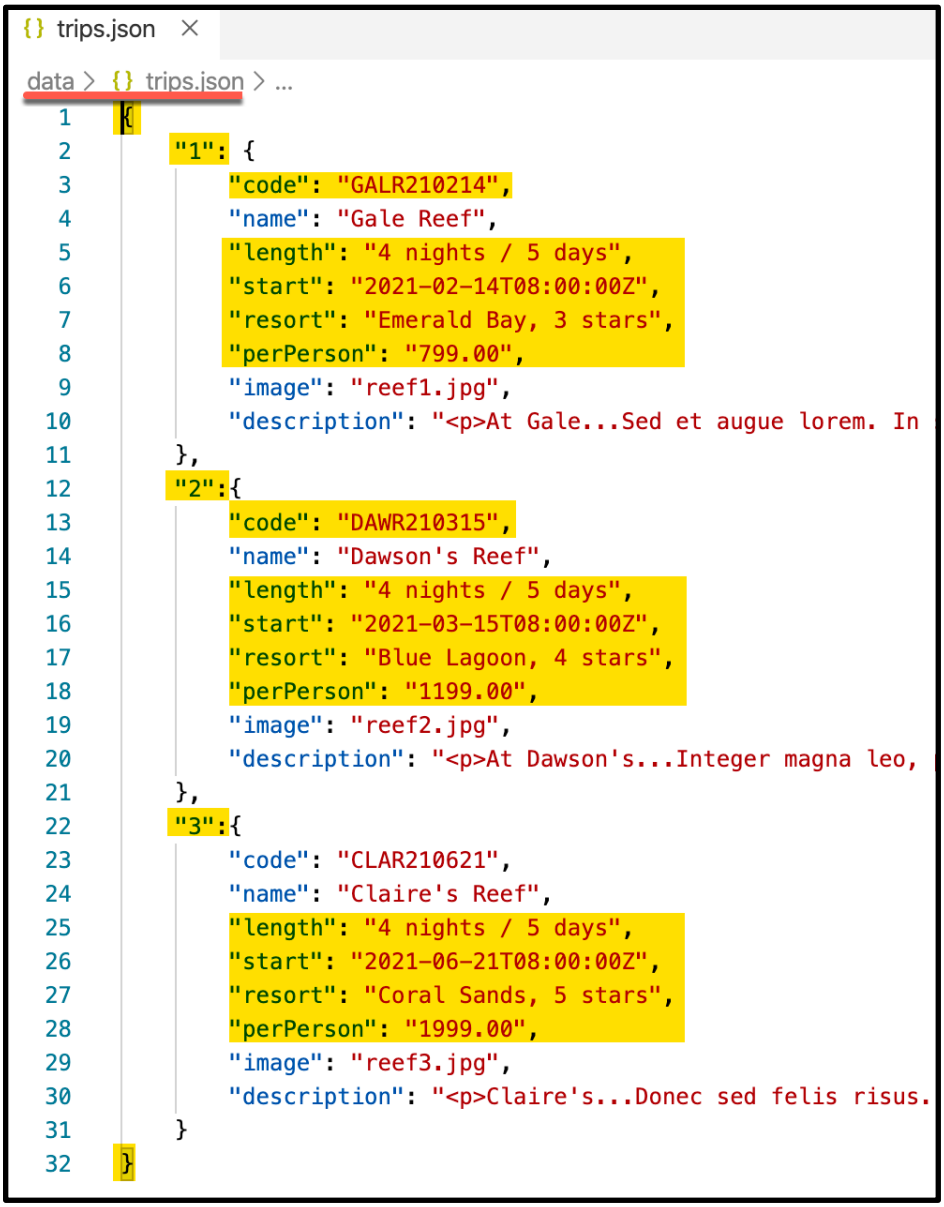
3. We can use the **db.js** module from Chapter 5, section 1 of your textbook with only minor changes for our project.



```
db.js
app_server > models > db.js > ...
1   const mongoose = require('mongoose');
2   const host = process.env.DB_HOST || '127.0.0.1'
3   const dbURI = `mongodb://${host}/travlr`;
4   const readLine = require('readline');
5
6   // avoid 'current Server Discovery and Monitoring engine is deprecated'
7   mongoose.set('useUnifiedTopology', true);
8
9   const connect = () => {
10    setTimeout(() => mongoose.connect(dbURI, {
11      useNewUrlParser: true,
12      useCreateIndex: true
13    }), 1000);
14  }
15
16  mongoose.connection.on('connected', () => {…
18  });
19
20  mongoose.connection.on('error', err => {…
23  });
24
25  mongoose.connection.on('disconnected', () => {…
27  });
28
29  if (process.platform === 'win32') {…
37  }
38
39  const gracefulShutdown = (msg, callback) => {…
44  };
45
46  // For nodemon restarts
47  process.once('SIGUSR2', () => {…
51  });
52  // For app termination
53  process.on('SIGINT', () => {…
57  });
58  // For Heroku app termination
59  process.on('SIGTERM', () => {…
63  });
64
65  connect();
66
67  // bring in the Mongoose schema
68  require('./travlr');
```

Seed the Database

4. Modify the **trips.json** file containing sample data to provide the additional properties and instances required by the schema.



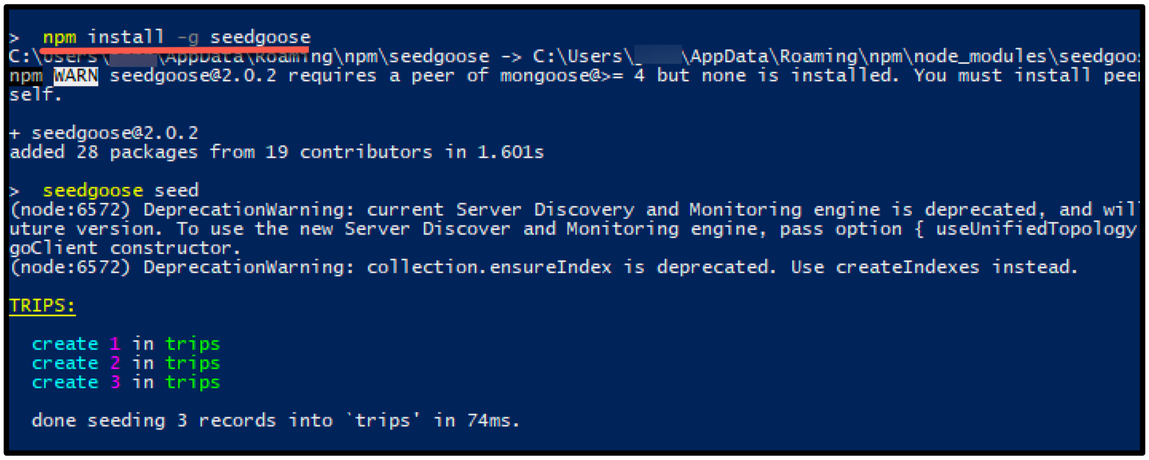
trips.json

data > {} trips.json > ...

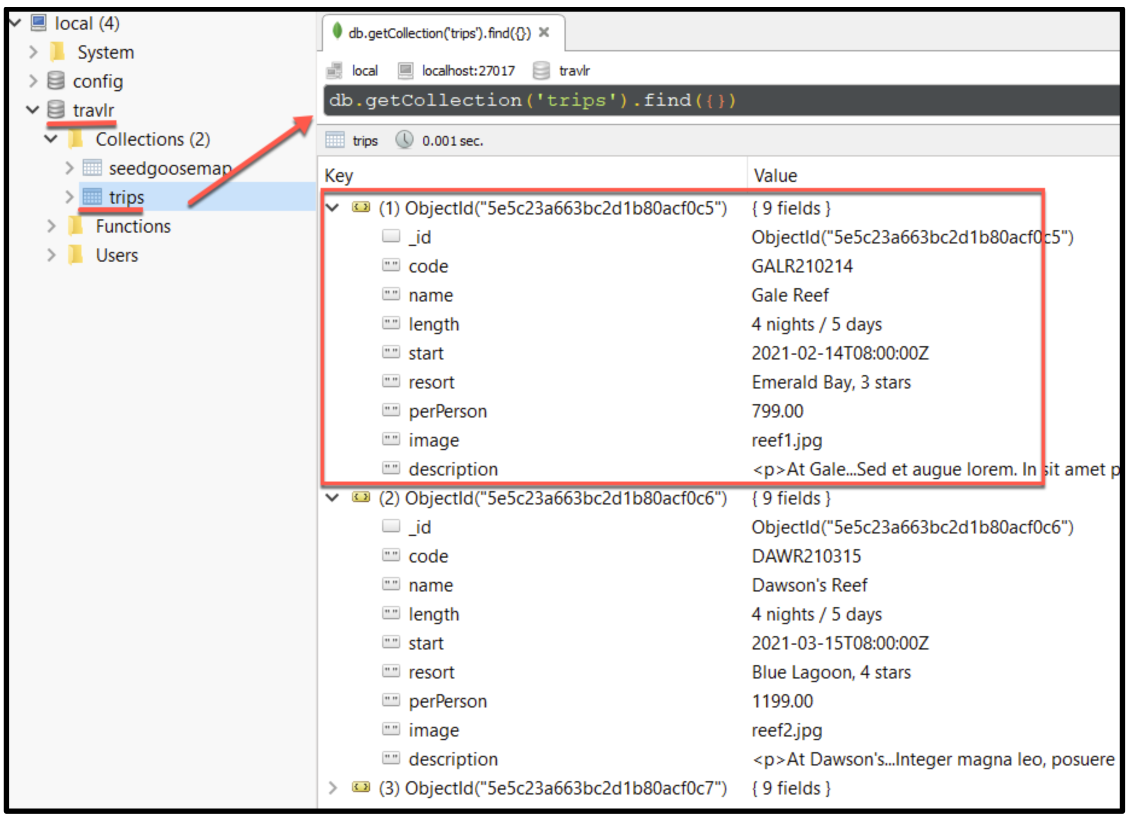
```
{
    "1": {
        "code": "GALR210214",
        "name": "Gale Reef",
        "length": "4 nights / 5 days",
        "start": "2021-02-14T08:00:00Z",
        "resort": "Emerald Bay, 3 stars",
        "perPerson": "799.00",
        "image": "reef1.jpg",
        "description": "<p>At Gale...Sed et augue lorem. In
    },
    "2":{
        "code": "DAWR210315",
        "name": "Dawson's Reef",
        "length": "4 nights / 5 days",
        "start": "2021-03-15T08:00:00Z",
        "resort": "Blue Lagoon, 4 stars",
        "perPerson": "1199.00",
        "image": "reef2.jpg",
        "description": "<p>At Dawson's...Integer magna leo,
    },
    "3":{
        "code": "CLAR210621",
        "name": "Claire's Reef",
        "length": "4 nights / 5 days",
        "start": "2021-06-21T08:00:00Z",
        "resort": "Coral Sands, 5 stars",
        "perPerson": "1999.00",
        "image": "reef3.jpg",
        "description": "<p>Claire's...Donec sed felis risus.
    }
}
```

A unique trip code has been created to identify each trip along with the additional values defined in the schema for a trip. The numbers “1”, “2”, et cetera signify that this is an array of nested JSON objects, which is the format used by Seedgoose to populate the database.

5. Install Seedgoose globally (-g) to allow easily populating (seeding) the database with test data, even before the application supports it. This will allow iterative coding and testing of the RESTful endpoints as you go. Import the seed data using the command `seedgoose seed`.
    - If you run into an error, install Seedgoose locally in the “travlr\” root directory using `npm install seedgoose`. You will then execute the following in Powershell: `.\node_modules\.bin\seedgoose seed`.



6. After running the Seedgoose command, open the MongoDB GUI and examine the newly loaded data.



Module 5: RESTful API

Begin by creating a top-level folder called "app_api" with "controllers" and "routes" subfolders. Drag the **models** folder from **app_server** up to **app_api**.

Create Mongoose Schema and DB Access Code

1. Create a new “trips.js” module that uses Mongoose syntax to retrieve both a list of trips and a single trip.

trips.js

app_api > controllers > trips.js > ...

```
1  const mongoose = require('mongoose'); //.set('debug', true);
2  const Model = mongoose.model('trips');  (a)
3
4  // GET: /trips - lists all the trips      (b)
5  const tripsList = async (req, res) => {
6      Model
7          .find({}) // empty filter for all   (c)
8          .exec((err, trips) => {
9              if (!trips) {
10                 return res                  (d)
11                     .status(404)
12                     .json({ "message": "trips not found" });
13             } else if (err) {
14                 return res
15                     .status(404)
16                     .json(err);
17             } else {
18                 return res
19                     .status(200)            (e)
20                     .json(trips);
21             }
22         });
23 };
24
25 // GET: /trips/:tripCode - returns a single trip   (f)
26 const tripsFindByCode = async (req, res) => {
27     Model
28         .find({ 'code': req.params.tripCode })   (g)
29         .exec((err, trip) => {
30             if (!trip) {
31                 return res
32                     .status(404)
33                     .json({ "message": "trip not found" });
34             } else if (err) {
35                 return res
36                     .status(404)
37                     .json(err);
38             } else {
39                 return res
40                     .status(200)
41                     .json(trip);
42             }
43         });
44 };
45
46 module.exports = {
47     tripsList,
48     tripsFindByCode
49 };
```

These two functions use the mongoose-provided .find() method; the first with no filter so all trips are returned, and the second with the tripCode passed on the URL to filter to just that trip. Specific key points shown in the image are as follows.

a. Import the Mongoose database library and our schema so it is available to use here.
b. Callback method registered in the **/api/trips/** route
c. Use the mongoose **.find()** method with no filter criteria to return all instances.
d. If nothing is returned, send an HTTP 404 status.
e. Finally, if data was retrieved, send it to the client with the HTTP 200 success status.
f. Callback method registered in the **/api/trips/{tripCode}** route
g. Use the mongoose **.find()** method with a filter set to the tripCode passed on the URL.

2. With the model, controller, and route completed, the last thing to do is to wire up the new API logic in the application.

app.js > ...

```
1   const createError = require('http-errors');
2   const express = require('express');
3   const path = require('path');
4   const cookieParser = require('cookie-parser');
5   const logger = require('morgan');
6   const hbs = require('hbs');
7   require('./app_api/models/db');
8
9   const indexRouter = require('./app_server/routes/index');
10  const usersRouter = require('./app_server/routes/users');
11  const travelRouter = require('./app_server/routes/travel');
12  const apiRouter = require('./app_api/routes/index');
13
14  const app = express();
15
16  // view engine setup
17  app.set('views', path.join(__dirname, 'app_server', 'views'));
18
19  // register handlebars partials (https://www.npmjs.com/package/hbs)
20  hbs.registerPartials(path.join(__dirname, 'app_server', 'views/partials'));
21
22  app.set('view engine', 'hbs');
23
24  app.use(logger('dev'));
25  app.use(express.json());
26  app.use(express.urlencoded({ extended: false }));
27  app.use(cookieParser());
28  app.use(express.static(path.join(__dirname, 'public')));
29
30  app.use('/', indexRouter);
31  app.use('/users', usersRouter);
32  app.use('/travel', travelRouter);
33  app.use('/api', apiRouter);
34
35  // catch 404 and forward to error handler
36  app.use(function(req, res, next) {
37    next(createError(404));
38  });
```

Trigger database connection and mongoose schema models to be loaded

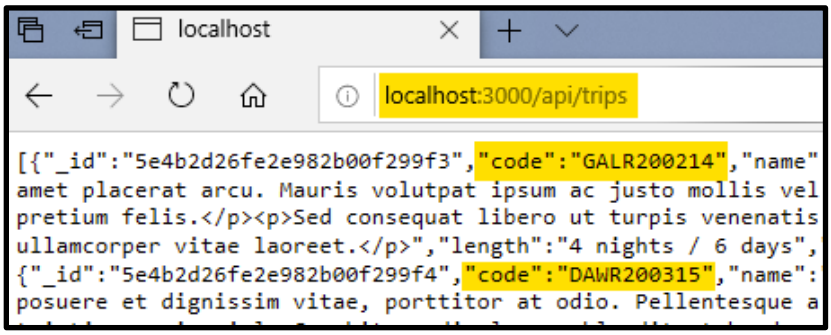
Reference new app_api router

Send requests for '/api' to the api router

```
require('./app_api/models/db');
...
const apiRouter = require('./app_api/routes/index');
...
app.use('/api', apiRouter);
```

Test the Data Access Code

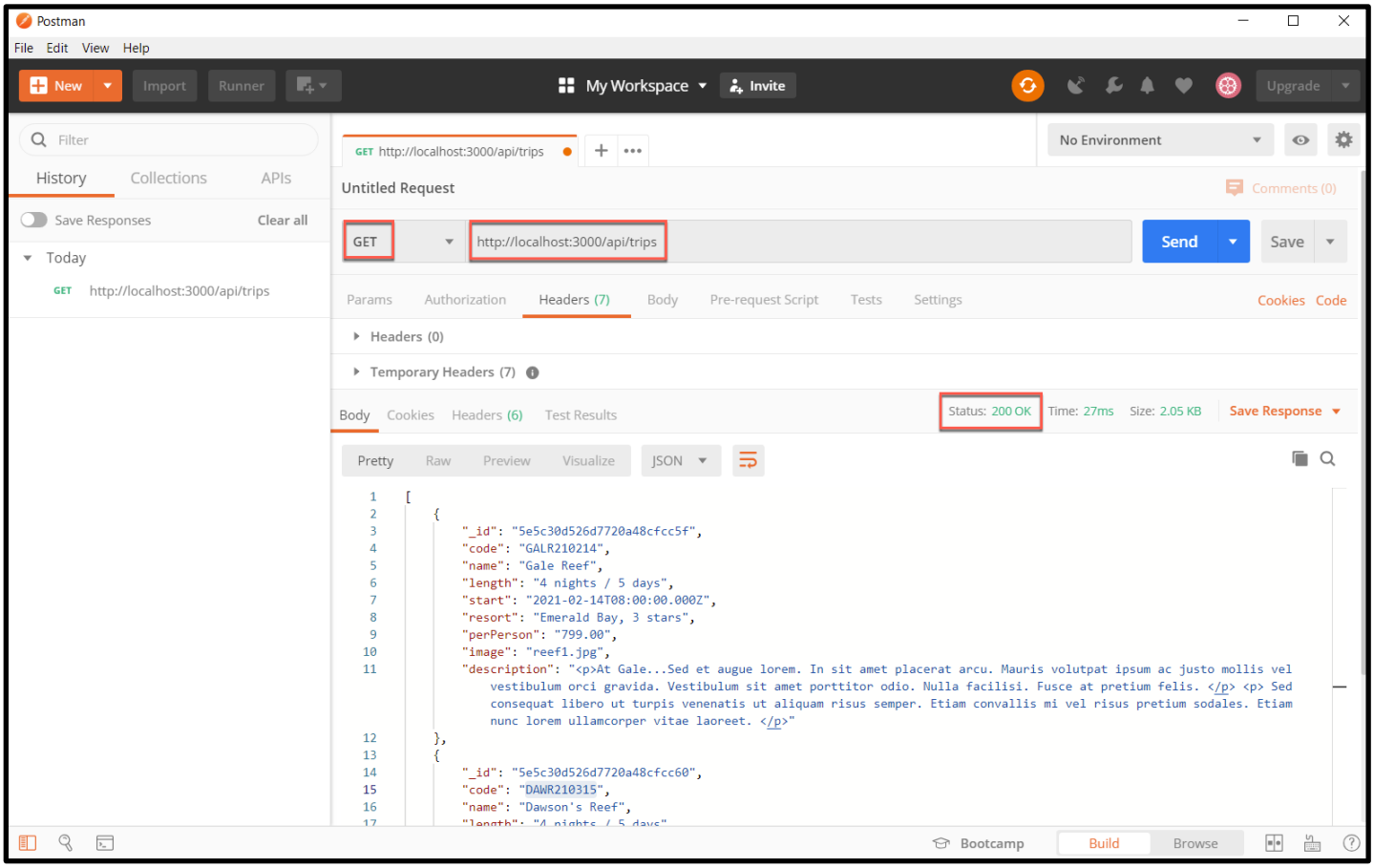
3. You can perform a quick and dirty test of the new API endpoints by typing in the URL using your browser.



Retrieve a list of all trips

Retrieve a single trip

4. You can also use Postman, a powerful testing tool, to send HTTP requests and process HTTP responses returned.

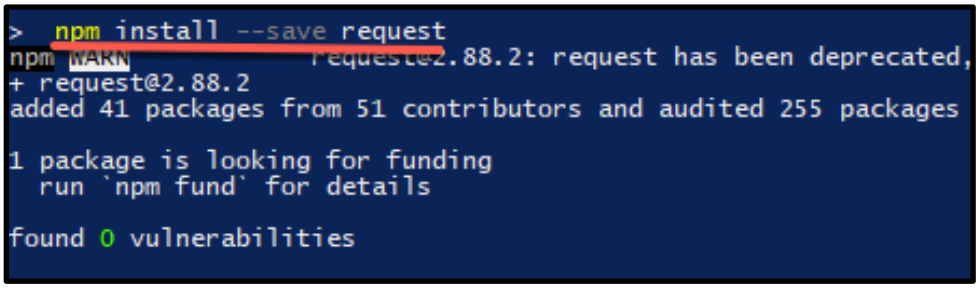


5. Since we modified the local JSON file containing test data, let's use the additional data there to test a bit further. Modify the **travel** Handlebars view to set the trip code into the URL, to allow clicking the image to take you to the API results.

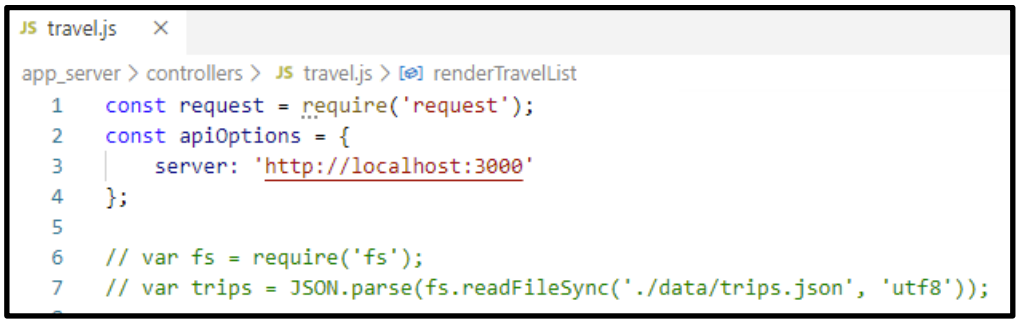
By modifying the image anchor tag to reference the API path including the trip code, you can now click on the image and see the JSON data for that trip. This is, of course, a contrived example that you wouldn't do on a production website. However, while developing, these kinds of tricks can be useful time savers to allow quick testing.

Modify Public Website to Use API Endpoints

6. The previous step uncovered another "to-do"; the rendered view is still using static JSON from a file. Follow the steps in Chapter 7 of your textbook to wire up the view to the new API endpoints that pull data from the database.
    a. Install the **request** Node module.



    b. In the **travel** controller, replace the filesystem component retrieval of the static JSON file with importing the request module so it can make HTTP requests.




```
const request = require('request');
const apiOptions = {
   server: 'http://localhost:3000'
}
```

    c. Replace logic in the **travelList** method to make the HTTP request.

travel.js

app_server > controllers > travel.js > travelList

```
29
30  /* GET travel list view */
31  const travelList = (req, res) => {
32      const path = '/api/trips';
33      const requestOptions = {
34          url: `${apiOptions.server}${path}`,
35          method: 'GET',
36          json: {},
37      };
38      console.info('>> travelController.travelList calling ' + requestOptions.url);
39      request(
40          requestOptions,
41          (err, { statusCode }, body) => {
42              if (err) {
43                  console.error(err);
44              }
45              renderTravelList(req, res, body);
46          }
47      );
48  };
49
```

log on the console this call over to the api

call new method to handle rendering the view

d. Add a new internal method to contain the logic for processing the response body returned from the API call and render the **travel** view.

travel.js

app_server > controllers > travel.js > renderTravelList

```
8
9   /* internal method to render the travel list */
10  const renderTravelList = (req, res, responseBody) => {
11      let message = null;
12      let pageTitle = process.env.npm_package_description + ' - Travel';
13      if (!(responseBody instanceof Array)) {
14          message = 'API lookup error';
15          responseBody = [];
16      } else {
17          if (!responseBody.length) {
18              message = 'No trips exist in our database!';
19          }
20      }
21      res.render('travel',
22          {
23              title: pageTitle,
24              trips: responseBody,
25              message
26          }
27      );
28  }
29
```

```
const requestOptions: {
    url: string;
```

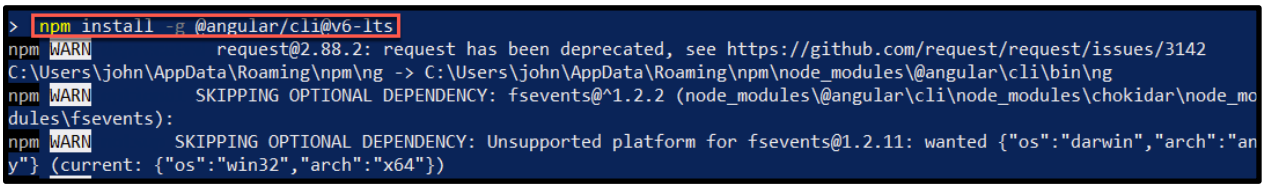
e. Optional: Repeat the process to create **travelDetails** and **renderTravelDetails** methods.

Module 6: SPA

Create Angular Admin Site

1. Ensure Angular's Command Line Interface (CLI) v6-LTS (version 6 long-term support) is installed.

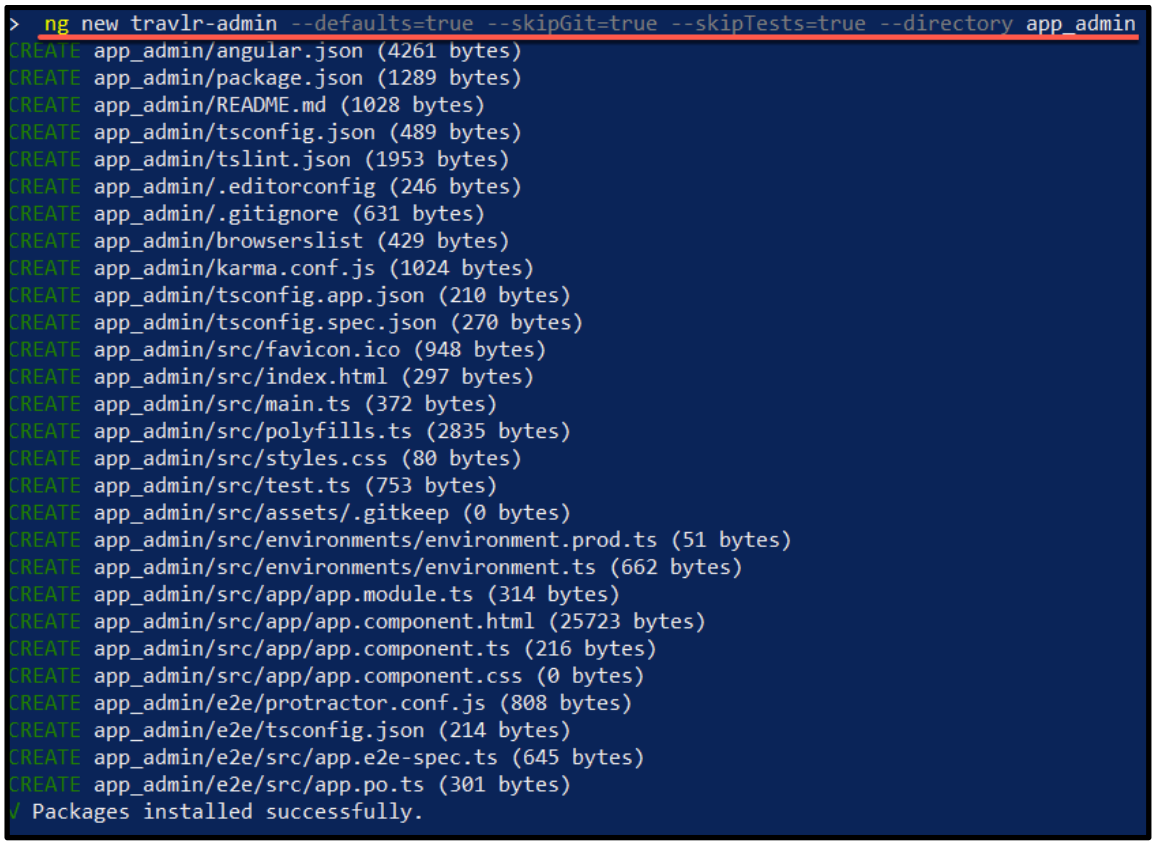
   ```
   npm install -g @angular/cli@v6-lts
   ```



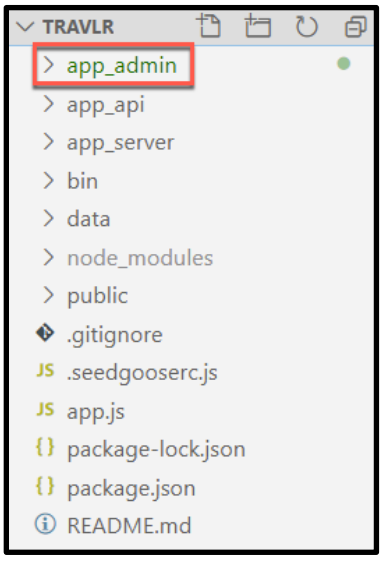
   Note: It is very important that you install Angular CLI v6-LTS because the code we will be writing matches the textbook. Newer versions of Angular have undergone significant changes and our code, as written, will not transpile (compile) nor run properly!

2. Create a new AngularJS application.

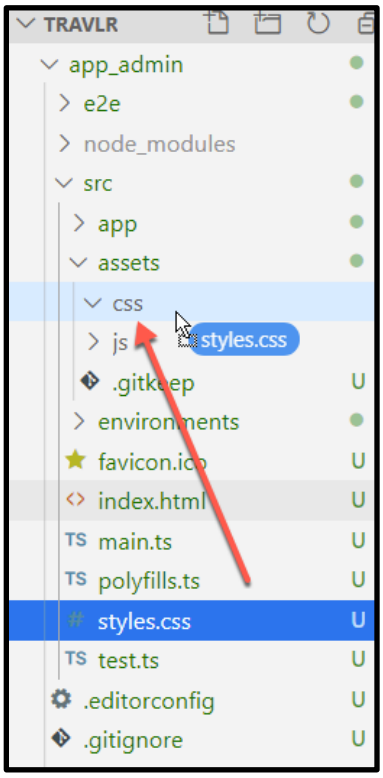
   ```
   ng new travlr-admin --defaults=true --skipGit=true --
   skipTests=true --directory app_admin
   ```



3. Opening VS Code, you will see there's now a new **app_admin** folder.



4. Create a new **css** folder under **/src/assets** within the **app_admin** folder, then drag the **style.css** file into the new folder.

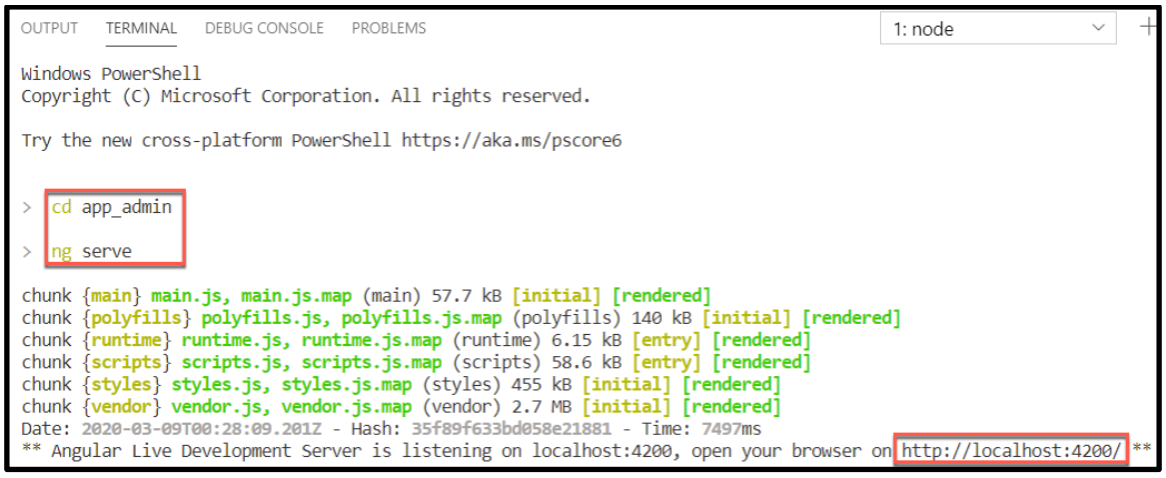


After moving the stylesheet, update the two references within **angular.json** from **src/styles.css** to **src/assets/css/styles.css**.

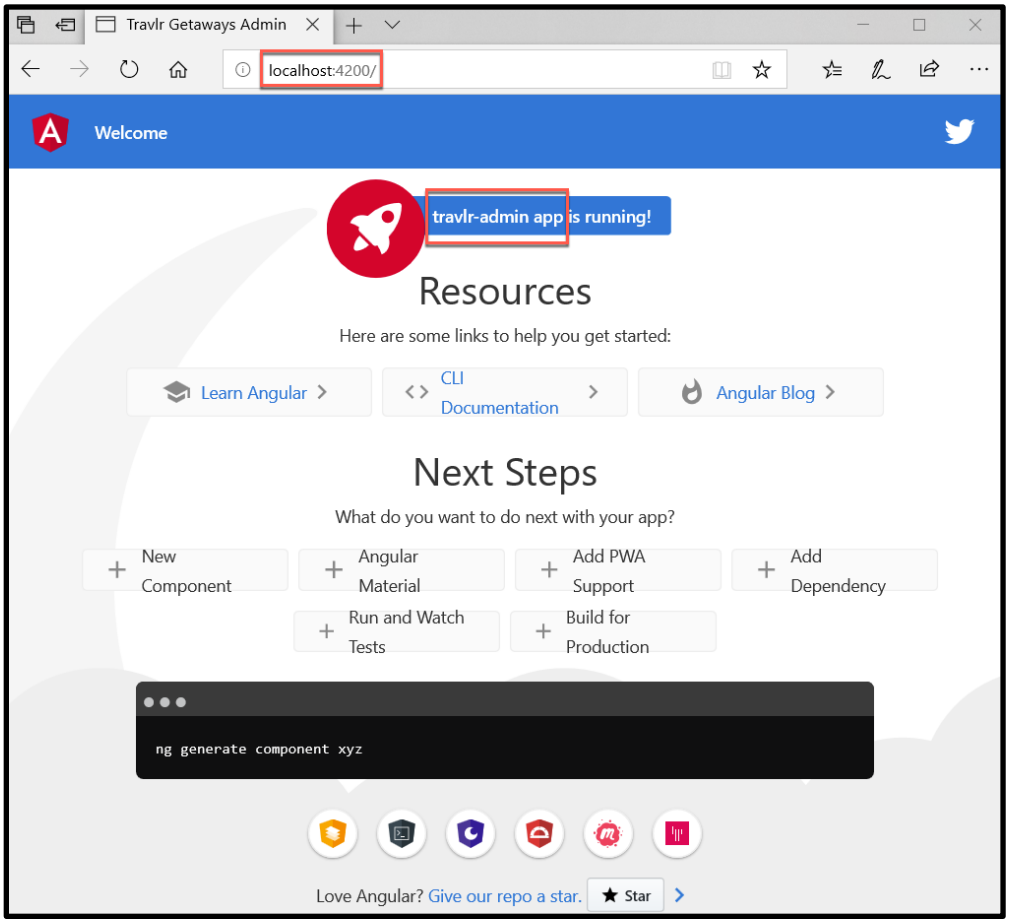
5. You may also wish to replace the title and add a loading message.

6. From the Terminal menu in VS Code, choose **New Terminal**, then launch the Angular website using the following commands:

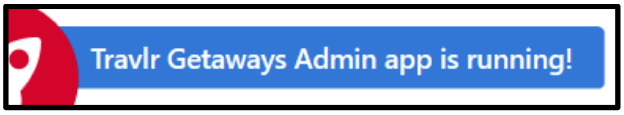
```
cd app_admin
ng serve
```



7. Open a web browser and navigate to “http://localhost:4200”.



8. Notice the blue bar next to the rocket; the text is not very user friendly. Edit **/src/app/app.component.ts** to change the title string and save it. You should immediately see the browser refresh and the text render properly.



You have successfully created an Angular application to support the administrative functionality required to support Travlr Getaways!

Create Trip Listing Component

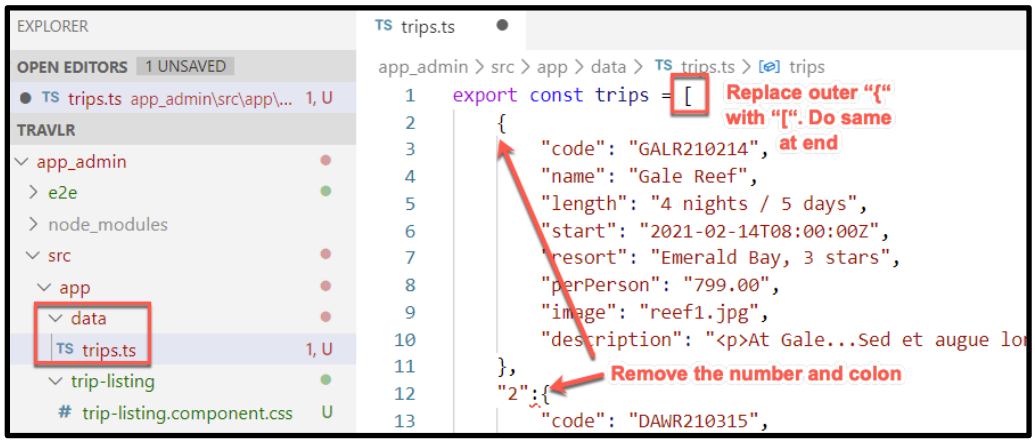
This section will detail the steps needed to build your first Angular component to display the list of trips.

9. In the PowerShell window, change the directory to “app_admin” and generate an Angular component called “trip-listing” to list the trips.

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

```
> cd app_admin

> ng generate component trip-listing
CREATE src/app/trip-listing/trip-listing.component.html (27 bytes)
CREATE src/app/trip-listing/trip-listing.component.ts (298 bytes)
CREATE src/app/trip-listing/trip-listing.component.css (0 bytes)
UPDATE src/app/app.module.ts (418 bytes)
```

10. Create a data folder under **/src/app** and then create a **trips.ts** TypeScript file to contain some test trip JSON data (collection of trip records). Copy the contents of **trips.json** from the earlier test file in the top-level **/data** folder, replace the outer square brackets "[" with curly braces "{", and remove the numbered identifiers.



11. Edit the **trip-listing.component.ts** file to add an import of the new **trips.ts** file, and define a class variable within the **TripListingComponent** class to contain the data.

12. Edit the **trip-listing.component.html** file and completely replace the paragraph tag with the following Angular expression:

```
<pre>{{ trips | json }}</pre>
```

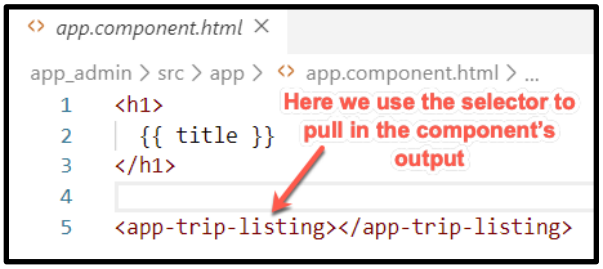
By adding the trips array to the class file, the variable becomes accessible from within the HTML. Notice that Angular uses the double curly brace notation, similar to Handlebars.

13. Edit the **app.component.html** file to render the trip data by completely replacing the contents with the following code:

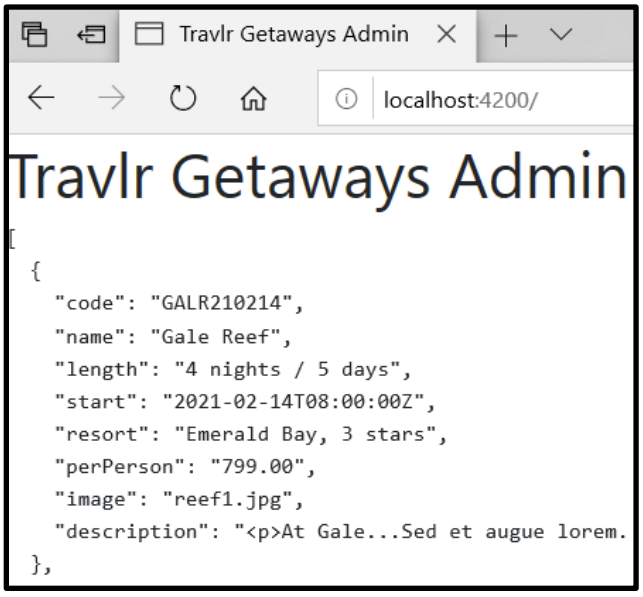
```
<h1>{{ title }}</h1>
<app-trip-listing></app-trip-listing>
```

This angle-bracket syntax, similar to HTML's tags, is known as a "selector". The Angular framework will inject the output of the component that contains the selector definition shown above.

14. It should now look like this:



15. As soon as you save the **app.component.html** file, you will see the browser refresh and display the raw JSON data.



16. Navigate to the Bootstrap CSS framework documentation. Copy the CSS reference and the three JS references into the **index.html** file as shown here.

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.

```
<link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/boot
strap.min.css" integrity="sha384-
ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js"
integrity="sha384-
q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
crossorigin="anonymous"></script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/
popper.min.js" integrity="sha384-
UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIHNDz0W1"
crossorigin="anonymous"></script>
<script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootst
rap.min.js" integrity="sha384-
JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM"
crossorigin="anonymous"></script>
```

17. Copy the **images** folder from the Express public folder to **/src/app/assets** so the images are available to the Angular application.

18. Replace the contents of **trip-listing.component.html** with the following HTML:


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

The above code fragment relies heavily on the Bootstrap CSS framework to format the trips in a pleasing manner.

19. The **app.component.html** page can be cleaned up a bit to render the listing with a navigation bar and some whitespace.

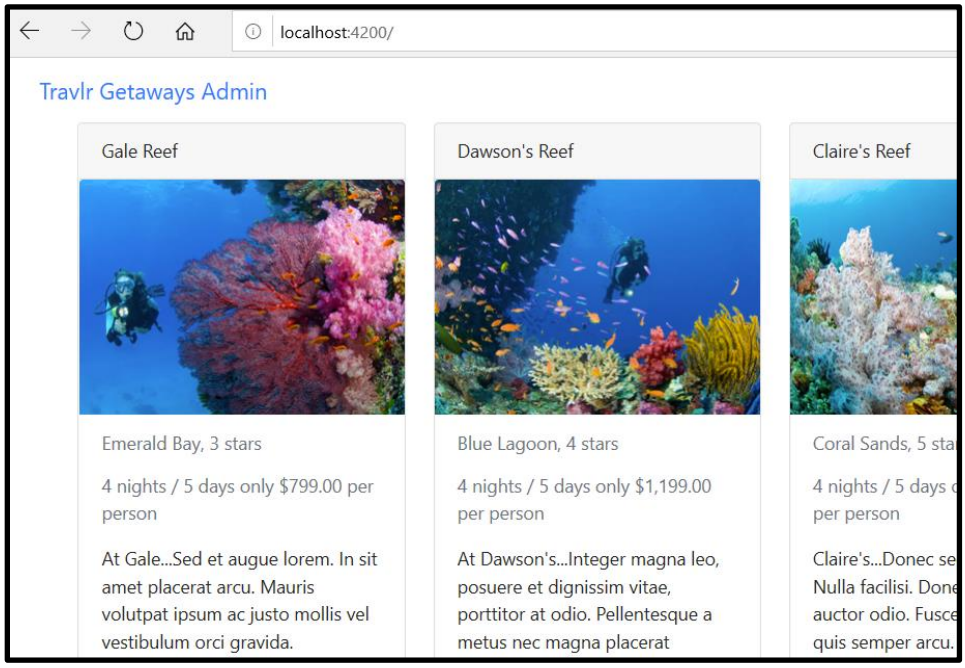
app.component.html

app_admin > src > app > app.component.html > ...

```
1   <!-- <h1>{{ title }}</h1> -->
2   <nav class="navbar navbar-default">
3     <div class="container-fluid">
4       <a class="navbar-brand" href="#">{{ title }}</a>
5     </div>
6   </nav>
7
8   <div class="container">
9     <app-trip-listing></app-trip-listing>
10  </div>
```

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

20. The Trip Listing page now looks like this:



Refactor Trip Rendering Logic Into an Angular Component

While the trip listing component looks better, there's still a problem: It can only show the trip listing one way, as cards. If we wish to give the user the ability to toggle between a card view and tabular list view, it quickly becomes a nightmare of coding to handle two different kinds of rendering in the same place.

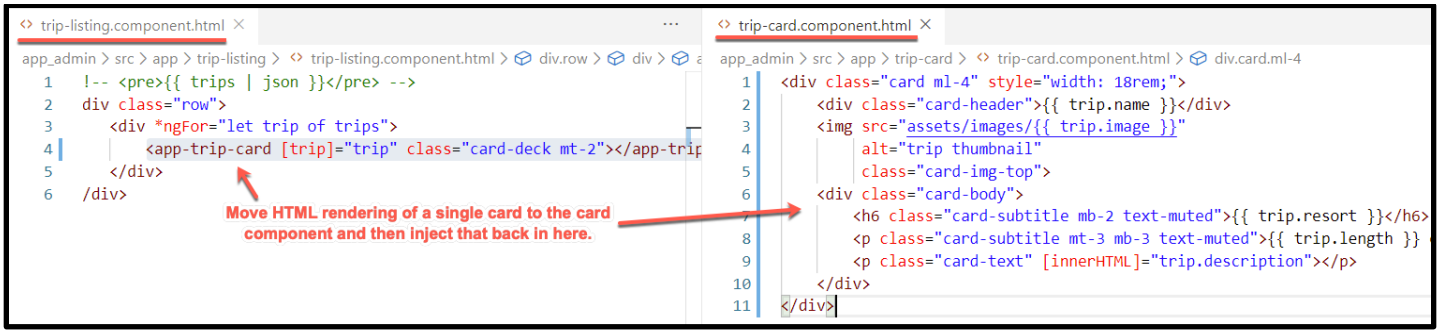
However, if the card-rendering logic were itself a component, then the trip listing page would only need to toggle the correct component (via a selector tag) to switch the layout. This technique of separating logic so that a class only does one thing, instead of being a single, large "mega-class" that does everything, is known as Separation of Concerns or SoC—a powerful software engineering principle.

21. Generate an Angular component called "trip-card" to hold the rendering logic of a card view.

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

```
> ng generate component trip-card
CREATE src/app/trip-card/trip-card.component.html (24 bytes)
CREATE src/app/trip-card/trip-card.component.ts (286 bytes)
CREATE src/app/trip-card/trip-card.component.css (0 bytes)
UPDATE src/app/app.module.ts (510 bytes)
```

22. Move the rendering logic from within the for-loop inside **trip-listing.component.html** into **trip-card.component.html** and replace it with a selector for the new **app-trip-card** component.




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

```
<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':'symbol' }} per person</p>
        <p class="card-text" [innerHTML]="trip.description"></p>
    </div>
</div>
```

23. Edit the **trip-card.component.ts** file and add the **Input** directive, so it can accept the trip passed into it from the trip-listing component and make the data available for rendering a single instance of a trip.


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

@Component({
 selector: 'app-trip-card',
 templateUrl: './trip-card.component.html',
 styleUrls: ['./trip-card.component.css']
})
export class TripCardComponent implements OnInit {
 @Input('trip') trip: any;
 constructor() { }
 ngOnInit(): void { }
}
```

With a few quick steps, the rendering of a single trip in a card-style view has been refactored into a separate Angular component that can now be used anywhere in the application.

Create Trip Data Service

It is common to need to obtain information from another part of the application in a distributed architecture. In a single-page application (SPA), this means calling a REST endpoint to obtain data. Angular supports using SoC by defining a "service" to contain functions or objects that are used by components to accomplish something.

Earlier we created a "/api" REST endpoint on the Express backend application to handle data requests. Now we will create an Angular service to handle access to the backend endpoint for trip information.

24. In order to call from the Angular admin site over to the Express backend API, you must first add code to the Express application in **app.js** to allow calls to be made.

```
JS app.js > ...
27  app.use(cookieParser());
28  app.use(express.static(path.join(__dirname, 'public')));
29
30  // allow CORS
31  app.use('/api', (req, res, next) => {
32    res.header('Access-Control-Allow-Origin', 'http://localhost:4200');
33    res.header('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content-Type, Accept');
34    next();
35  });
36
37  app.use('/', indexRouter);
38  app.use('/users', usersRouter);
39  app.use('/travel', travelRouter);
40  app.use('/api', apiRouter);
```

See the section "Allowing CORS requests in Express" in Chapter 9 of the textbook for more information.

25. Create a new Angular service called "trip-data" with the following command:

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

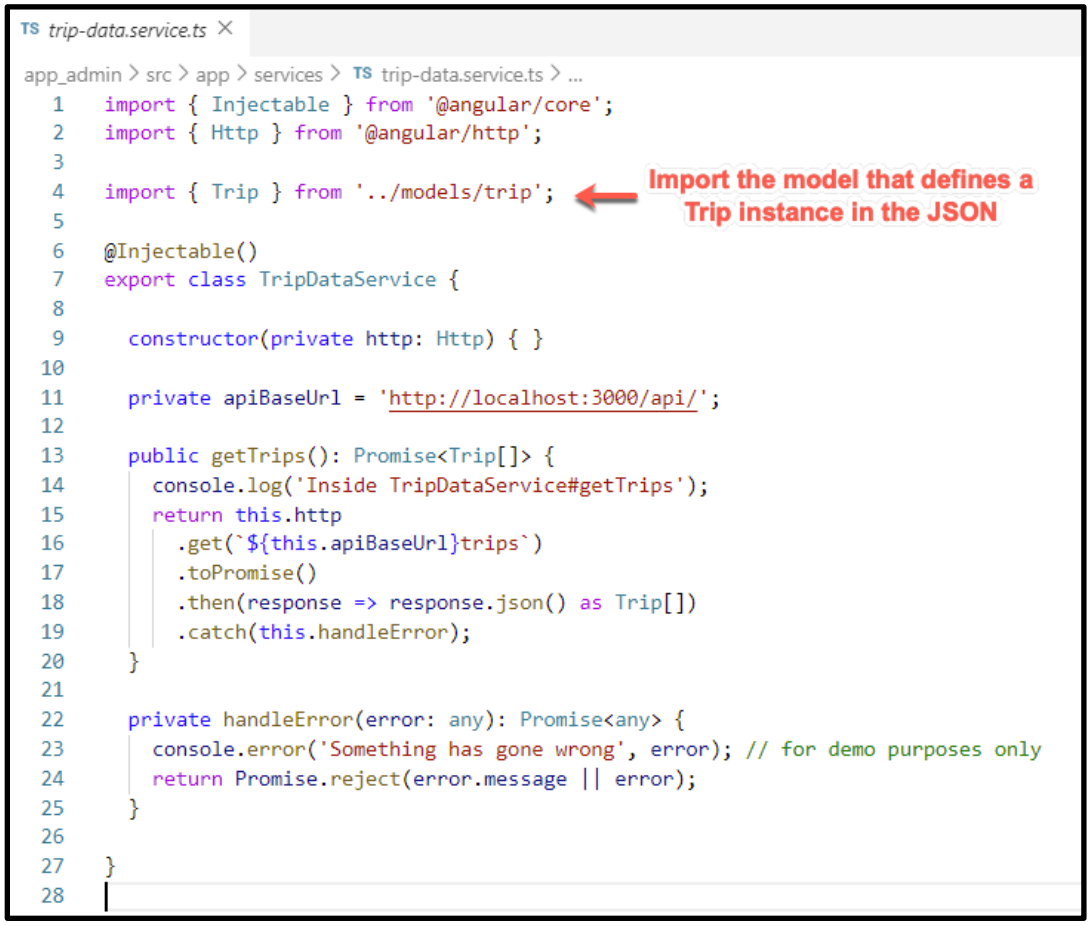
26. Create a new folder under **app_admin** called "services" and move the **trip-data.service.ts** file from the **app_admin** folder into this new folder. As more services are created, the code will be organized together in one place.

27. Create a new folder under **app_admin** called "models" and create an interface to define the data for a single Trip that will be received from the API endpoint as JSON.

```
export interface Trip {
    _id: string, // internal MongoDB primary key
    code: string,
    name: string,
    length: string,
    start: Date,
    resort: string,
    perPerson: string,
    image: string,
    description: string
}
```

Instances of this interface will be used to transfer the HTML form data back to your component as well as back and forth with the REST endpoint. Angular will take care of automatically converting or mapping the JSON data into a JavaScript object and back.

28. Now edit the **trip-data.service.ts** file and paste in the following code:




```
import { Injectable } from '@angular/core';
import { Http } from '@angular/http';

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

@Injectable()
export class TripDataService {

  constructor(private http: Http) { }

  private apiBaseUrl = 'http://localhost:3000/api/';

  public getTrips(): Promise<Trip[]> {
    console.log('Inside TripDataService#getTrips');
    return this.http
      .get(`${this.apiBaseUrl}trips`)
      .toPromise()
      .then(response => response.json() as Trip[])
      .catch(this.handleError);
  }

  private handleError(error: any): Promise<any> {
    console.error('Something has gone wrong', error); // for demo purposes only
    return Promise.reject(error.message || error);
  }

}
```

This logic is very much like the textbook's "loc8r-data" class, adapted for our Trips.

29. To register this new service with the Angular application, edit **app.module.ts** to import the new module as well as the **HttpModule**.

app.module.ts

app_admin > src > app > app.module.ts > ...

```
1   import { BrowserModule } from '@angular/platform-browser';
2   import { NgModule } from '@angular/core';
3   import { HttpModule } from '@angular/http';
4
5   import { AppComponent } from './app.component';
6   import { TripListingComponent } from './trip-listing/trip-listing.component';
7   import { TripCardComponent } from './trip-card/trip-card.component';
8   import { TripDataService } from './services/trip-data.service';
9
10  @NgModule({
11    declarations: [
12      AppComponent,
13      TripListingComponent,
14      TripCardComponent
15    ],
16    imports: [
17      BrowserModule,
18      HttpModule
19    ],
20    providers: [
21      TripDataService
22    ],
23    bootstrap: [AppComponent]
24  })
25  export class AppModule { }
26
```

30. Edit **trip-listing.component.ts** to replace the loading of trips from the JSON test file with a call to the new **TripDataService** which will, in turn, call the **/api** endpoint on the backend Express application.


```
 import { Component, OnInit } from '@angular/core';
// import { trips } from '../data/trips';
import { TripDataService } from '../services/trip-data.service';
import { Trip } from '../models/trip';

@Component({
 selector: 'app-trip-listing',
 templateUrl: './trip-listing.component.html',
 styleUrls: ['./trip-listing.component.css'],
```

```
 providers: [TripDataService]
})
export class TripListingComponent implements OnInit {

 // trips: Array<any> = trips;
 trips: Trip[];
  message: string;

 constructor(private tripDataService: TripDataService) { }

 private getTrips(): void {
   console.log('Inside TripListingComponent#getTrips');
   this.message = 'Searching for trips';
   this.tripDataService
     .getTrips()
       .then(foundTrips => {
         this.message = foundTrips.length > 0 ? ''
                                              : 'No trips found';
         this.trips = foundTrips;
       });
 }

 ngOnInit(): void {
   this.getTrips();
 }
}
```

The local testing file containing JSON trip data helped to quickly build out the user interface functionality. The mock data has now been replaced with a service class that handles the interaction with the backend API and ultimately the MongoDB database.

Add/Edit Trips

This section focuses on creating code to support adding a new trip and updating an existing trip. Doing so will require adding new components, forms, and routes as well as exposing additional endpoint logic on the **/api** backend server to support these new requirements.

31. Generate a new Angular component called “add-trip”.

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

```
>  ng generate component add-trip
CREATE src/app/add-trip/add-trip.component.html (27 bytes)
CREATE src/app/add-trip/add-trip.component.ts (276 bytes)
CREATE src/app/add-trip/add-trip.component.css (0 bytes)
UPDATE src/app/app.module.ts (745 bytes)
```

32. Put an “Add Trip” button on the top of the trip listing page.

trip-listing.component.html

app_admin > src > app > trip-listing > trip-listing.component.html > div.row

```
1  <!-- <pre>{{ trips | json }}</pre> -->
2  <div>
3      <button (click)="addTrip()" class="btn btn-info">Add Trip</button>
4  </div>
5  <div class="row">
6      <div *ngFor="let trip of trips">
```

33. The button click shown above is configured to invoke the **addTrip()** function. The **addTrip()** function is going to use Angular's built-in routing capabilities to navigate the UI to the **add-trip-component**.

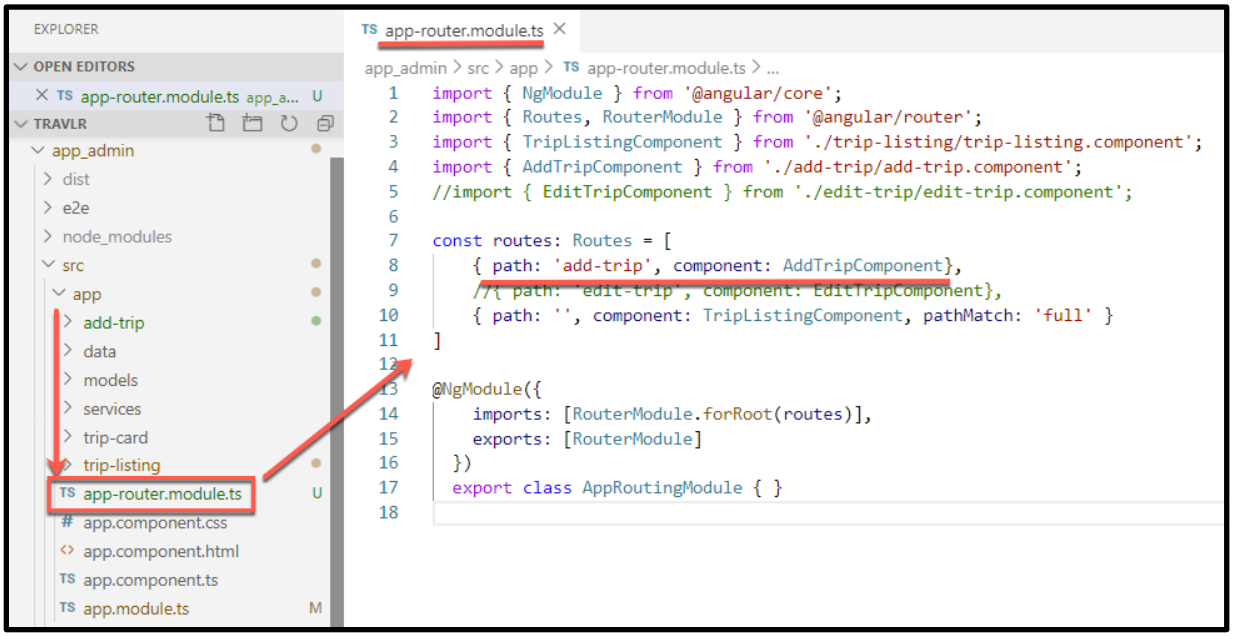
trip-listing.component.ts

app_admin > src > app > trip-listing > TS trip-listing.component.ts > ...

```
1  import { Component, OnInit } from '@angular/core';
2  import { Router } from "@angular/router";
3  // import { trips } from '../data/trips';
4  import { TripDataService } from '../services/trip-data.service';
5  import { Trip } from '../models/trip';
6
7  @Component({
8    selector: 'app-trip-listing',
9    templateUrl: './trip-listing.component.html',
10   styleUrls: ['./trip-listing.component.css'],
11   providers: [TripDataService]
12 })
13 export class TripListingComponent implements OnInit {
14
15   // trips: Array<any> = trips;
16   trips: Trip[];
17
18   message: string;
19
20   constructor(
21     private tripDataService: TripDataService,
22     private router: Router
23     ) { }
24
25   private addTrip(): void {
26     this.router.navigate(['add-trip']);
27   }
28
29   private getTrips(): void {
30     console.log('Inside TripListingComponent#getTrips');
```

The additional code first imports the Angular **Router** module, then defines an additional parameter in the constructor so the Angular framework knows to inject an instance of the router into the class. Finally, the **addTrip()** function simply uses the router to navigate to the new component.

34. It is common for SPAs, and web applications in general, to have many URLs for their various pages. It is also a best practice to keep the routing code separate and modular. Add a new file to the **app_admin/src/app** folder called "app-router.module.ts", alongside the other application files there.



Line #8 is where the "add-trip" route is defined and mapped to the **AddTripComponent**.

Line #10 defines the default path when none is specified to invoke the **TripListingComponent**.

35. The new module must be imported into the main application module so it will be loaded when the Angular application starts up.

app.module.ts

app_admin > src > app > app.module.ts > ...

```
1   import { BrowserModule } from '@angular/platform-browser';
2   import { NgModule } from '@angular/core';
3   import { HttpModule } from '@angular/http';
4   import { ReactiveFormsModule } from "@angular/forms";
5
6   import { AppComponent } from './app.component';
7   import { AppRoutingModule } from './app-router.module';
8   import { TripListingComponent } from './trip-listing/trip-listing.
9   import { TripCardComponent } from './trip-card/trip-card.component
10  import { TripDataService } from './services/trip-data.service';
11  import { AddTripComponent } from './add-trip/add-trip.component';
12
13  @NgModule({
14    declarations: [
15      AppComponent,
16      TripListingComponent,
17      TripCardComponent,
18      AddTripComponent
19    ],
20    imports: [
21      BrowserModule,
22      HttpModule,
23      ReactiveFormsModule,
24      AppRoutingModule
25    ],
26    providers: [
27      TripDataService
28    ],
29    bootstrap: [AppComponent]
30  })
31  export class AppModule { }
32
```

The **ReactiveFormsModule** is also needed to support interacting with HTML forms, child controls, and handling form submissions.

36. Rather than hard-coding the initial application display view of the trip listing, a small change is needed to support routing. The router is now in control of all the page transitions, including the initial page displayed. Replace the highlighted HTML snippets in **app.component.html**:

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

With the following HTML snippet:

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

The router-outlet selector is a placeholder that Angular will dynamically fill based on the router's current state.

37. The **add-trip.component.html** file is fairly simplistic in layout and provides a minimal error message if a field is missing.

```
<div class="col-md-3">
  <h2 class="text-center">Add Trip</h2>
  <form [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>
```

The remainder of the code is similar to the above and not shown. SNIP!

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

38. Here is the **add-trip.component.ts**.

TS add-trip.component.ts app_admin\src\app\add-trip\add-trip.component.ts\ AddTripComponent\ onSubmit

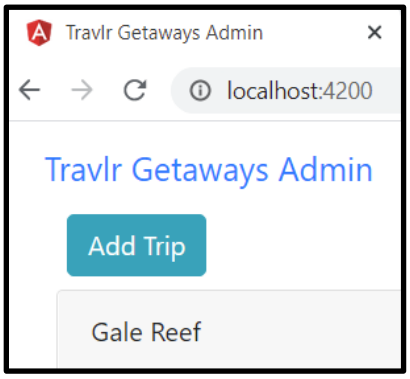
```
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from "@angular/forms";
import { Router } from "@angular/router";
import { TripDataService } from '../services/trip-data.service';

@Component({
  selector: 'app-add-trip',
  templateUrl: './add-trip.component.html',
  styleUrls: ['./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],
    })
  }

  onSubmit() {
    this.submitted = true;
    if(this.addForm.valid){
      this.tripService.addTrip(this.addForm.value)
      .subscribe( data => {
        console.log(data);
        this.router.navigate(['']);
      });
    }
  }

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

}
```

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

@Component({
 selector: 'app-add-trip',
 templateUrl: './add-trip.component.html',
 styleUrls: ['./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],
   })
 }

 onSubmit() {
   this.submitted = true;
   if(this.addForm.valid){
     this.tripService.addTrip(this.addForm.value)
     .then( data => {
       console.log(data);
       this.router.navigate(['']);
     });
   }
 }
 // get the form short name to access the form fields
 get f() { return this.addForm.controls; }
}
```

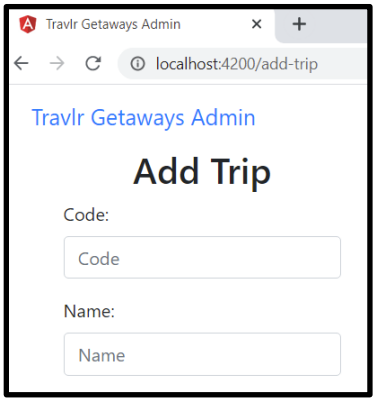
The **@angular/forms** module is the magic that allows code to access the data using what is known as "data binding". The JavaScript variables are "bound" to the HTML form controls, with Angular's forms

library taking care of moving data back and forth between the view (HTML) and the model (JavaScript).

39. Running the application will show the new **Add Trip** button at the top of the application.



40. Clicking the **Add Trip** button brings up the new **AddTripComponent** and view just created.



41. In the **AddTripComponent** above, there was a red jagged underline indicating an error in the **onSubmit()** function where it is trying to call the **addTrip()** function of the **TripDataService**. That function still needs to be written.

```
public addTrip(formData: Trip): Promise<Trip> {
  console.log('Inside TripDataService#addTrip');
  return this.http
    .post(`${this.apiBaseUrl}trips`, formData)
    .toPromise()
    .then(response => response.json() as Trip[])
    .catch(this.handleError);
}
```

This function is very similar to the **getTrips()** function (not shown here), with the notable exception of using the HTTP POST method and passing the form data in the body of the request.

42. The API backend route needs to be modified to handle the HTTP POST verb sent by the SPA frontend application. Add the following function call to **app_api/routes/index.js**:

The above change tells Express to route the POST request to the **/api/trips** endpoint to the **tripsAddTrip** function in the trips controller.

43. The trips controller needs to be modified to handle the incoming HTTP request containing the trip JSON passed within the request body.

trips.js

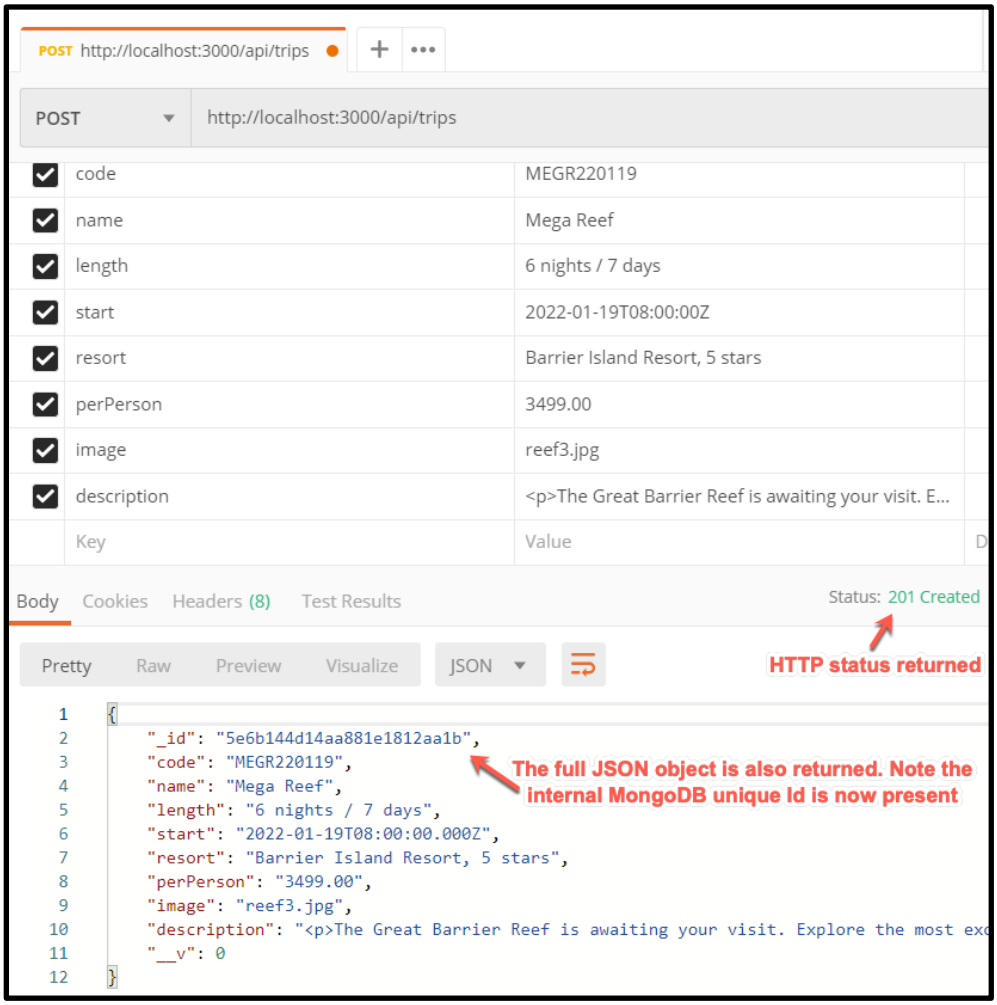
app_api > controllers > trips.js > ...

```
45
46  const tripsAddTrip = async (req, res) => {
47      model
48      .create({
49          code: req.body.code,
50          name: req.body.name,
51          length: req.body.length,
52          start: req.body.start,
53          resort: req.body.resort,
54          perPerson: req.body.perPerson,
55          image: req.body.image,
56          description: req.body.description
57      },
58      (err, trip) => {
59          if (err) {
60              return res
61                  .status(400) // bad request, invalid content
62                  .json(err);
63          } else {
64              return res
65                  .status(201) // created
66                  .json(trip);
67          }
68      });
69  }
70
71  module.exports = {
72      tripsList,
73      tripsFindCode,
74      tripsAddTrip
75  };
```

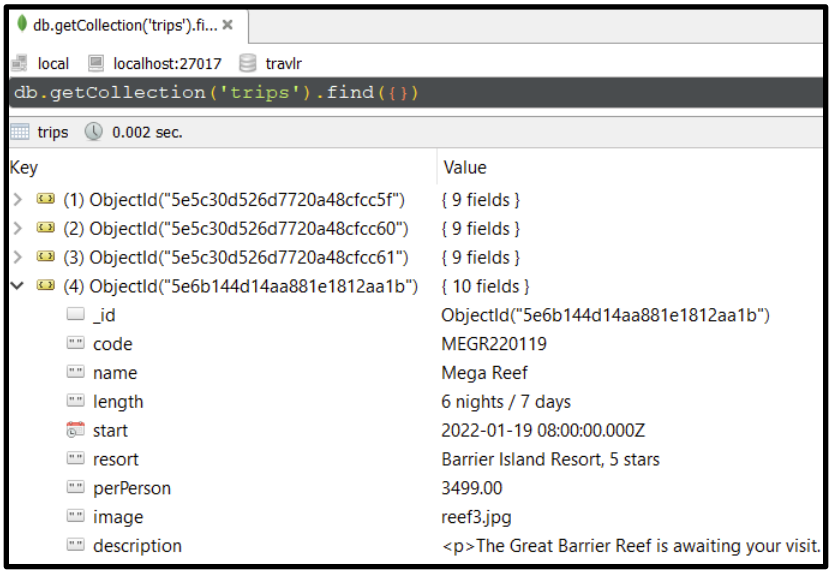
44. The backend API method can be first tested with Postman. Here we can see the console log showing the POST sent to the API endpoint, along with the response returned to the caller—in this case, Postman.

```
connected
POST /api/trips 201 88.164 ms - 342
```

Note the HTTP status code of 201 being returned to indicate the result of “created”.



Here is the new trip that has been persisted in MongoDB:



45. Having worked out the logic for handling an **add** operation using the HTTP POST verb, coding support for **edit/update** becomes trivial. Updating data at a REST endpoint uses the HTTP PUT verb. The steps working from the backend to the frontend are as follows.

    a. Create a new method on the **/app_api/controllers/trips.js** controller called "tripsUpdateTrip".

trips.js

app_api > controllers > trips.js > ...

```
70
71  const tripsUpdateTrip = async (req, res) => {
72      console.log(req.body);
73      model
74          .findOneAndUpdate({ 'code': req.params.tripCode }, {
75              code: req.body.code,
76              name: req.body.name,
77              length: req.body.length,
78              start: req.body.start,
79              resort: req.body.resort,
80              perPerson: req.body.perPerson,
81              image: req.body.image,
82              description: req.body.description
83          }, { new: true })
84          .then(trip => {
85              if (!trip) {
86                  return res
87                      .status(404)
88                      .send({
89                          message: "Trip not found with code " + req.params.tripCode
90                      });
91              }
92              res.send(trip);
93          }).catch(err => {
94              if (err.kind === 'ObjectId') {
95                  return res
96                      .status(404)
97                      .send({
98                          message: "Trip not found with code " + req.params.tripCode
99                      });
100             }
101             return res
102                 .status(500) // server error
103                 .json(err);
104         });
105 }
106
107 module.exports = {
108     tripsList,
109     tripsFindCode,
110     tripsAddTrip,
111     tripsUpdateTrip
112 };
```

```
const tripsUpdateTrip = async (req, res) => {
 console.log(req.body);
 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
        }, { new: true })
        .then(trip => {
            if (!trip) {
                return res
                    .status(404)
                    .send({
                        message: "Trip not found with code "
+ req.params.tripCode
                    });
            }
            res.send(trip);
        }).catch(err => {
            if (err.kind === 'ObjectId') {
                return res
                    .status(404)
                    .send({
                        message: "Trip not found with code "
+ req.params.tripCode
                    });
            }
            return res
                .status(500) // server error
                .json(err);
        });
}
```

This method is very similar in structure to the **tripsAddTrip** method, with the exception of using the **findOneAndUpdate** method from Mongoose.

b. Add a new route to support the PUT verb.

index.js

app_api > routes > index.js > ...

```
1   const express = require('express');
2   const router = express.Router();
3
4   const tripsController = require('../controllers/trips');
5
6   router
7       .route('/trips')
8       .get(tripsController.tripsList)
9       .post(tripsController.tripsAddTrip);
10
11  router
12      .route('/trips/:tripCode')
13      .get(tripsController.tripsFindCode)
14      .put(tripsController.tripsUpdateTrip);
15
16  module.exports = router;
```

c. Since we didn't include explicit methods in the CORS configuration, only GET and POST are allowed. Add the following line to the application startup in **app.js** to explicitly allow PUT and DELETE:

app.js

app.js > app.use() callback

```
28  app.use(express.static(path.join(__dirname, 'public')));
29
30  // allow CORS
31  app.use('/api', (req, res, next) => {
32    res.header('Access-Control-Allow-Origin', 'http://localhost:4200');
33    res.header('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content-Type, Accept');
34    res.header('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE');
35    next();
36  });
```

d. In the Angular SPA frontend site, update the **TripDataService** with methods to get a single trip and to update a trip.

trip-data.service.ts

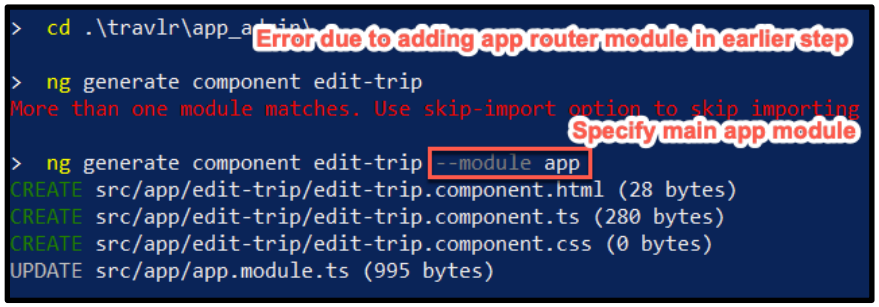
app_admin > src > app > services > trip-data.service.ts > ...

```
6   @Injectable()
7   export class TripDataService {
8
9     constructor(private http: Http) { }
10
11    private apiBaseUrl = 'http://localhost:3000/api/';
12    private tripUrl = `${this.apiBaseUrl}trips/`;
13
14    public addTrip(formData: Trip): Promise<Trip> {
15      console.log('Inside TripDataService#addTrip');
16      return this.http
17        .post(this.tripUrl, formData) // pass form data in request body
18        .toPromise()
19        .then(response => response.json() as Trip[])
20        .catch(this.handleError);
21    }
22
23    public getTrip(tripCode: string): Promise<Trip> {
24      console.log('Inside TripDataService#getTrip(tripCode)');
25      return this.http
26        .get(this.tripUrl + tripCode)
27        .toPromise()
28        .then(response => response.json() as Trip)
29        .catch(this.handleError);
30    }
31
32    public getTrips(): Promise<Trip[]> {
33      console.log('Inside TripDataService#getTrips');
34      return this.http
35        .get(this.tripUrl)
36        .toPromise()
37        .then(response => response.json() as Trip[])
38        .catch(this.handleError);
39    }
40
41    public updateTrip(formData: Trip): Promise<Trip> {
42      console.log('Inside TripDataService#upateTrip');
43      console.log(formData);
44      return this.http
45        .put(this.tripUrl + formData.code, formData)
46        .toPromise()
47        .then(response => response.json() as Trip[])
48        .catch(this.handleError);
49    }
```

```
  public getTrip(tripCode: string): Promise<Trip> {
    console.log('Inside TripDataService#getTrip(tripCode)');
    return this.http
      .get(this.tripUrl + tripCode)
      .toPromise()
      .then(response => response.json() as Trip)
      .catch(this.handleError);
  }
...
  public updateTrip(formData: Trip): Promise<Trip> {
    console.log('Inside TripDataService#upateTrip');
    console.log(formData);
    return this.http
      .put(this.tripUrl + formData.code, formData)
      .toPromise()
      .then(response => response.json() as Trip[])
      .catch(this.handleError);
  }
```

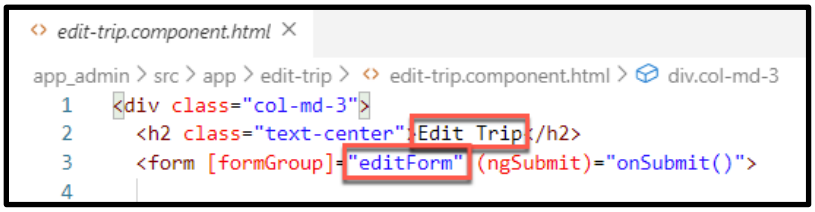
Since multiple parts of the code need the trip URL, it has been refactored into a single variable that can be referenced throughout the rest of the class. The **getTrip()** function is basic and very similar to the **getTrips()** function except that it returns a single trip instead of an array of trips. The **updateTrip()** function uses the **.put()** function of the HTTP module and passes the entire form data in the body of the request.

e. Add a new component for editing a trip.



As noted above, because we added a second application module for routing, you now need to specify which module the new component is to be generated within.

f. The HTML for an edit form is practically identical to that for the add form. Copy the contents of **add-trip.component.html** into **edit-component.html** and replace the title and name of the form.



g. Add an "Edit" button to the bottom of the HTML for the Trip Card.

```
6           <p class="card-subtitle mt-3 mb-3 text-muted">{{ trip.length }} only
7               {{ trip.perPerson | currency:'USD':'symbol' }} per person</p>
8           <p class="card-text" [innerHTML]="trip.description"></p>
9           <div>
10              <button (click)="editTrip(trip)" class="btn btn-info">Edit</button>
11          </div>
12      </div>
13  </div>
```

Note that when the "Edit" button is clicked, the **editTrip(trip)** function will be called and pass the trip instance that is displayed on the card.

h. The TripCard component needs to define the **editTrip()** function.

trip-card.component.ts

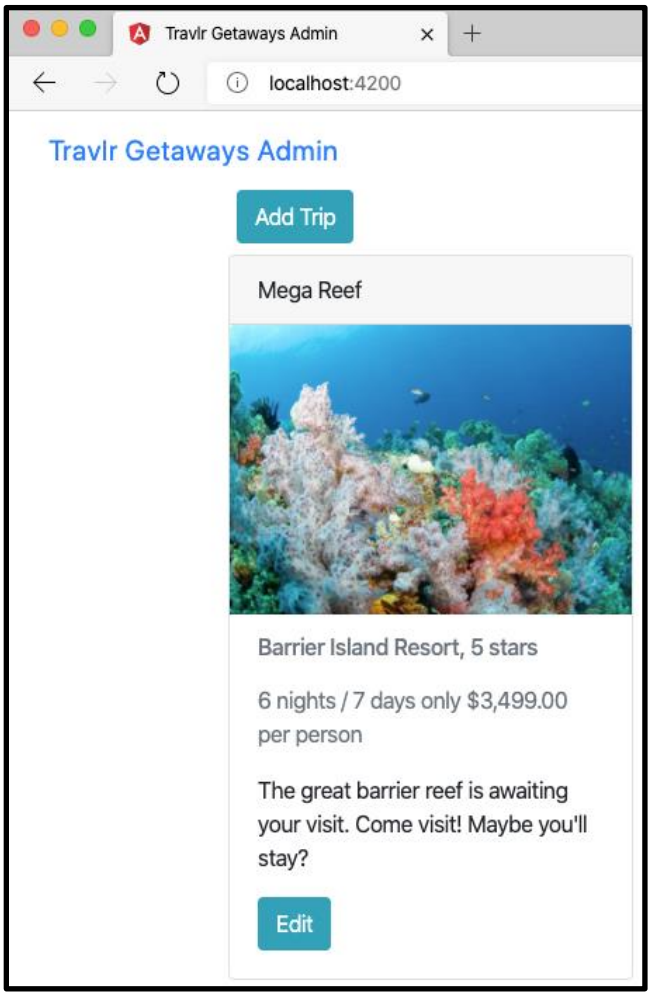
app_admin > src > app > trip-card > trip-card.component.ts > ...

```
1   import { Component, OnInit, Input } from '@angular/core';
2   import { Router } from "@angular/router";
3   import { Trip } from '../models/trip';
4
5   @Component({
6     selector: 'app-trip-card',
7     templateUrl: './trip-card.component.html',
8     styleUrls: ['./trip-card.component.css']
9   })
10  export class TripCardComponent implements OnInit {
11
12    @Input('trip') trip: Trip;
13
14    constructor(
15      private router: Router
16    ) { }
17
18    ngOnInit(): void {
19    }
20
21    private editTrip(trip: Trip): void {
22      localStorage.removeItem("tripCode");
23      localStorage.setItem("tripCode", trip.code);
24      this.router.navigate(['edit-trip']);
25    }
26  }
```

Stash the trip code in browser's local storage for the edit component to retrieve later

With an Angular SPA, the next page to display is managed by the router. Therefore, we must first import the **Router** module and update the constructor so the actual router instance can be injected by the Angular framework when this class is instantiated. Note the common practice of passing data between components by first storing it in the browser's local storage area.

i. Here's the Trip Card page with the added Edit button:



j. The Edit Trip component does extra work when it is initialized. Retrieve the trip code that was stashed, call the **getTrip()** function to retrieve the trip from the backend API endpoint, and then set the data retrieved into the form's data fields.

TS edit-trip.component.ts

app_admin > src > app > edit-trip > TS edit-trip.component.ts > ...

```
22    ngOnInit() {
23      // retrieve stashed tripId
24      let tripCode = localStorage.getItem("tripCode");
25      if (!tripCode) {
26        alert("Something wrong, couldn't find where I stashed tripCode!");
27        this.router.navigate(['']);
28        return;
29      }
30
31      // initialize form
32      this.editForm = this.formBuilder.group({
33        _id: [],
34        code: [tripCode, Validators.required],
35        name: ['', Validators.required],
36        length: ['', Validators.required],
37        start: ['', Validators.required],
38        resort: ['', Validators.required],
39        perPerson: ['', Validators.required],
40        image: ['', Validators.required],
41        description: ['', Validators.required],
42      })
43
44      this.tripService.getTrip(tripCode)
45        .then(data => {
46          console.log(data);
47          // Don't use editForm.setValue() as it will throw console error
48          this.editForm.patchValue(data);
49        })
50    }
51
52    onSubmit() {
53      this.submitted = true;
54
55      if (this.editForm.valid) {
56        this.tripService.updateTrip(this.editForm.value)
57          .then(data => {
58            console.log(data);
59            this.router.navigate(['']);
60          });
61      }
62    }
```

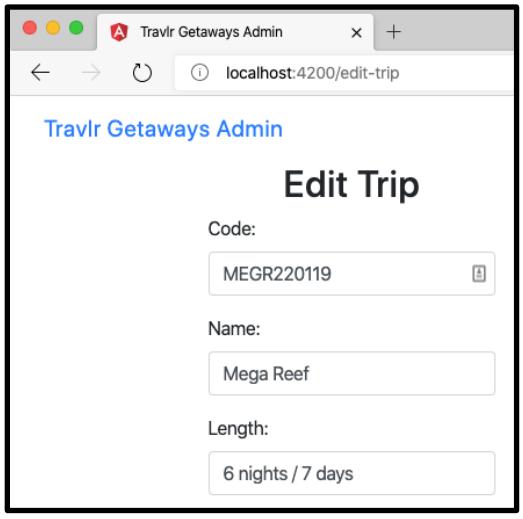
```
 ngOnInit() {
   // retrieve stashed tripId
   let tripCode = localStorage.getItem("tripCode");
   if (!tripCode) {
     alert("Something wrong, couldn't find where I stashed
tripCode!");
     this.router.navigate(['']);
     return;
   }

   console.log('EditTripComponent#onInit found tripCode ' +
tripCode);

   // initialize form
   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],
   })

   console.log('EditTripComponent#onInit calling
TripDataService#getTrip(\'' + tripCode + '\')');

   this.tripService.getTrip(tripCode)
     .then(data => {
       console.log(data);
       // Don't use editForm.setValue() as it will throw
console error
       this.editForm.patchValue(data[0]);
     })
 }

 onSubmit() {
   this.submitted = true;

   if (this.editForm.valid) {
     this.tripService.updateTrip(this.editForm.value)
       .then(data => {
         console.log(data);
         this.router.navigate(['']);
       });
   }
 }
```

The submission of the changed data is very similar to when adding a new Trip.

k. Remember to add the new route to the application router.

```
TS app-router.module.ts
app_admin > src > app > TS app-router.module.ts > ...
1   import { NgModule } from '@angular/core';
2   import { Routes, RouterModule } from '@angular/router';
3   import { TripListingComponent } from './trip-listing/trip-listing.component';
4   import { AddTripComponent } from './add-trip/add-trip.component';
5   import { EditTripComponent } from './edit-trip/edit-trip.component';
6
7   const routes: Routes = [
8       { path: 'add-trip', component: AddTripComponent },
9       { path: 'edit-trip', component: EditTripComponent },
10      { path: '', component: TripListingComponent, pathMatch: 'full' }
11  ]
```

l. The "Edit Trip" page should display the existing trip data.



Module 7: Security

Add Authentication to Express Backend

Chapter 11 in your textbook covers the aspects of adding security to support authenticating users and ensuring only an authenticated user can perform certain actions. This section will summarize the steps and point out differences with the Travlr Getaways application.

1. Install the following Node packages:

```
npm install jsonwebtoken
npm install crypto
npm install dotenv
npm install passport passport-local
npm install express-jwt
```

2. Create a new file named ".env" in the **/travlr** folder of the website with the following line:

```
JWT_SECRET=NewHampsh!reCollege0fAccounting@Commerce
```

Note that the filename begins with a dot. It is very important that you add or verify that this filename is listed in your **.gitignore** file. You never want your secrets to be stored in a source repository where all can see them.

3. Create a Mongoose schema in **/app_api/models/user.js** to represent an authenticated user.

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

const userSchema = new mongoose.Schema({
 email: {
   type: String,
   unique: true,
   required: true
 },
 name: {
   type: String,
   required: true
 },
 hash: String,
 salt: String
});

userSchema.methods.setPassword = function(password){
 this.salt = crypto.randomBytes(16).toString('hex');
 this.hash = crypto.pbkdf2Sync(password, this.salt,
   1000, 64, 'sha512').toString('hex');
};

userSchema.methods.validPassword = function(password) {
 var hash = crypto.pbkdf2Sync(password,
   this.salt, 1000, 64, 'sha512').toString('hex');
 return this.hash === hash;
};

userSchema.methods.generateJwt = function() {
 const expiry = new Date();
 expiry.setDate(expiry.getDate() + 7);

 return jwt.sign({
   _id: this._id,
   email: this.email,
   name: this.name,
   exp: parseInt(expiry.getTime() / 1000, 10),
```

```
 }, process.env.JWT_SECRET); // DO NOT KEEP YOUR SECRET IN THE
CODE!
};

mongoose.model('users', userSchema);
```

4. Configure Passport by adding a **config** folder under **/app_api** and then creating **passport.js** in there.

```
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'
 },
 (username, password, done) => {
   User.findOne({ email: username }, (err, user) => {
     if (err) { return done(err); }
     if (!user) {
       return done(null, false, {
         message: 'Incorrect username.'
       });
     }
     if (!user.validPassword(password)) {
       return done(null, false, {
         message: 'Incorrect password.'
       });
     }
     return done(null, user);
   });
 }
));
```

5. Create a controller for authenticating users in **/app_api/controllers/authentication.js**.

```
const passport = require('passport');
const mongoose = require('mongoose');
const User = mongoose.model('users');

const register = (req, res) => {
 if (!req.body.name || !req.body.email || !req.body.password) {
   return res
     .status(400)
     .json({"message": "All fields required"});
 }

 const user = new User();
 user.name = req.body.name;
 user.email = req.body.email;
```

```
  user.setPassword(req.body.password);
  user.save((err) => {
    if (err) {
      res
        .status(400)
        .json(err);
    } else {
      const token = user.generateJwt();
      res
        .status(200)
        .json({token});
    }
  })
};

const login = (req, res) => {
  if (!req.body.email || !req.body.password) {
    return res
      .status(400)
      .json({"message": "All fields required"});
  }
  passport.authenticate('local', (err, user, info) => {
    if (err) {
      return res
        .status(404)
        .json(err);
    }
    if (user) {
      const token = user.generateJwt();
      res
        .status(200)
        .json({token});
    } else {
      res
        .status(401)
        .json(info);
    }
  })(req, res);
};

module.exports = {
  register,
  login
};
```

6. Register the new authentication routes in **/app_api/routes/index.js**.

```
index.js
app_api > routes > index.js > ...
 9  const authController = require('../controllers/authentication');
10  const tripsController = require('../controllers/trips');
11
12  router
13      .route('/login')
14      .post(authController.login);
15
16  router
17      .route('/register')
18      .post(authController.register);
```

7. Initialize the Passport authentication in the application file **app.js**.

```
app.js
app.js >
 1  require('dotenv').config();
 2
 3  const createError = require('http-errors');
 4  const express = require('express');
 5  const path = require('path');
 6  const logger = require('morgan');
 7  const cookieParser = require('cookie-parser');
 8  const hbs = require('hbs');
 9  const passport = require('passport');
10
11  require('./app_api/models/db');
12
13  require('./app_api/config/passport');
14
```

```
require('dotenv').config();
  …
  const passport = require('passport');
  …
  require('./app_api/config/passport');
```

```
31  app.use(logger('dev'));
32  app.use(express.json());
33  app.use(express.urlencoded({ extended: false }));
34  app.use(cookieParser());
35  app.use(express.static(path.join(__dirname, 'public')));
36  app.use(passport.initialize());
37
```

```
38  // allow CORS
39  app.use('/api', (req, res, next) => {
40    res.header('Access-Control-Allow-Origin', 'http://localhost:4200');
41    res.header('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content-Type, Accept, Authorization');
42    res.header('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE');
43    next();
44  });
45
```

```
52
53  // catch unauthorized error and create 401
54  app.use((err, req, res, next) => {
55    if (err.name === 'UnauthorizedError') {
56      res
57        .status(401)
58        .json({"message": err.name + ": " + err.message});
59    }
60  });
```

8. Test the new endpoints using Postman. First try registering a new user with the **/register** endpoint, and then try logging in with the **/login** endpoint.

9. Update the router to add JWT authentication capabilities in **/app_api/routes/index.js**.

index.js

app_api > routes > index.js > ...

```
1  const express = require('express');
2  const router = express.Router();
3  const jwt = require('express-jwt');
4  const auth = jwt({
5    secret: process.env.JWT_SECRET,
6    userProperty: 'payload'
7  });
```

10. Update the routes that alter the database (add, update, and delete) by injecting the authentication middleware just added above.

index.js

app_api > routes > index.js > ...

```
19
20  router
21      .route('/trips')
22      .get(tripsController.tripsList)
23      .post(auth, tripsController.tripsAddTrip);
24
25  router
26      .route('/trips/:tripCode')
27      .get(tripsController.tripsFindCode)
28      .put(auth, tripsController.tripsUpdateTrip);
29
```

Inject authentication middleware

Note: These authentication middleware injections are the key security feature to protect the API routes from unauthenticated callers. Without them, the API calls are unrestricted so anyone can invoke them!

11. Update the Trips controller **/app_api/controller/trips.js** by adding a **getUser(req, res, callback)** function. To do so, copy the **getAuthor** function from the textbook and change the name to "getUser".

12. Update the **tripsAddTrip** function in the Trips controller **/app_api/controller/trips.js** by wrapping the body with a call to the **getUser** function just added above.

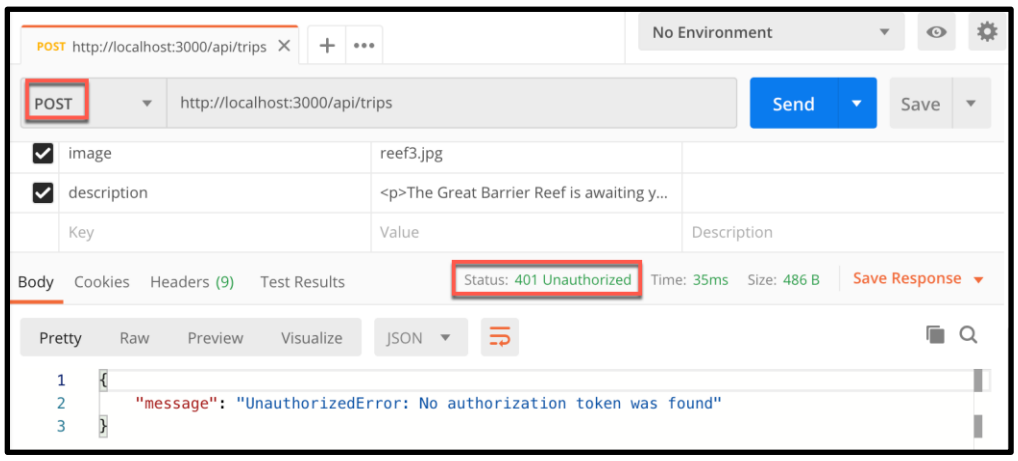
```
const tripsAddTrip = async (req, res) => {
   getUser(req, res,
       (req, res) => {
           Trip
               .create({
                   code: req.body.code,
                   name: req.body.name,
                   length: req.body.length,
                   start: req.body.start,
                   resort: req.body.resort,
                   perPerson: req.body.perPerson,
                   image: req.body.image,
                   description: req.body.description
               },
               (err, trip) => {
                   if (err) {
                       return res
                           .status(400) // bad request
                           .json(err);
                   } else {
                       return res
                           .status(201) // created
                           .json(trip);
                   }
               });
       }
   );
}
```

13. Update the **tripsUpdateTrip** function in the Trips controller **/app_api/controller/trips.js** by wrapping the body with a call to the **getUser** function just added above.

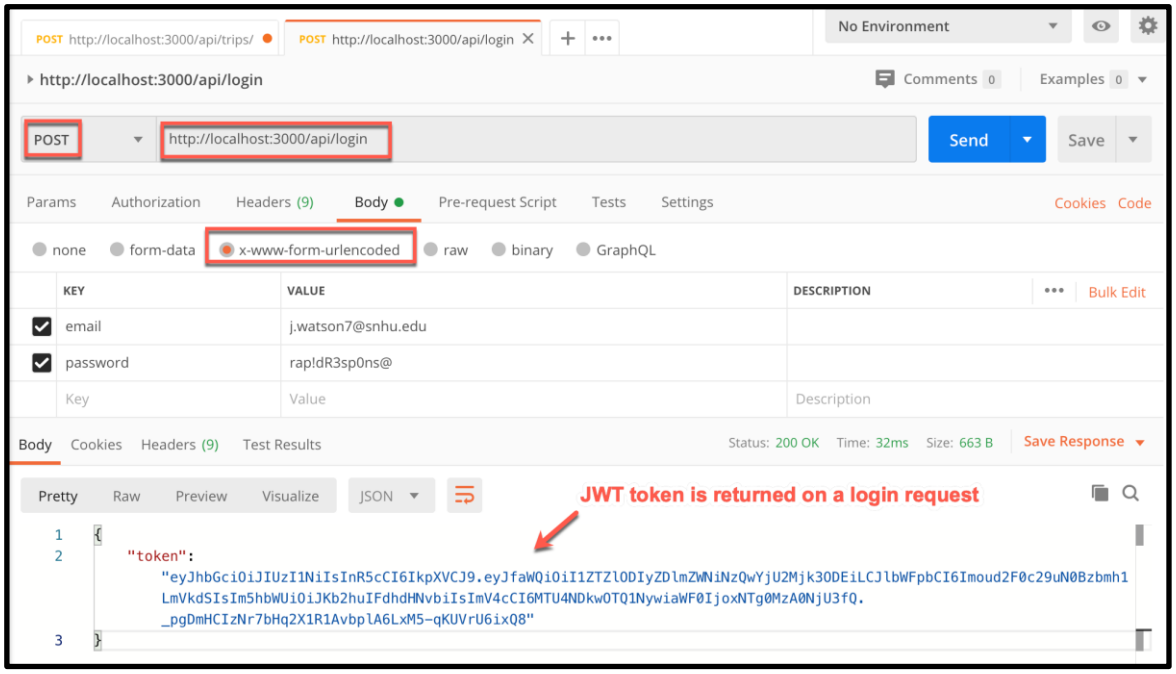
```
const tripsUpdateTrip = async (req, res) => {
   getUser(req, res,
       (req, res) => {
           Trip
               .findOneAndUpdate({'code': req.params.tripCode },{
                   code: req.body.code,
                   name: req.body.name,
```

```
                    length: req.body.length,
                    start: req.body.start,
                    resort: req.body.resort,
                    perPerson: req.body.perPerson,
                    image: req.body.image,
                    description: req.body.description
                }, { new: true })
                .then(trip => {
                    if (!trip) {
                        return res
                            .status(404)
                            .send({
                                message: "Trip not found with code
" + req.params.tripCode
                            });
                    }
                    res.send(trip);
                }).catch(err => {
                    if (err.kind === 'ObjectId') {
                        return res
                            .status(404)
                            .send({
                                message: "Trip not found with code
" + req.params.tripCode
                            });
                    }
                    return res
                        .status(500) // server error
                        .json(err);
                });
        }
    );
}
```

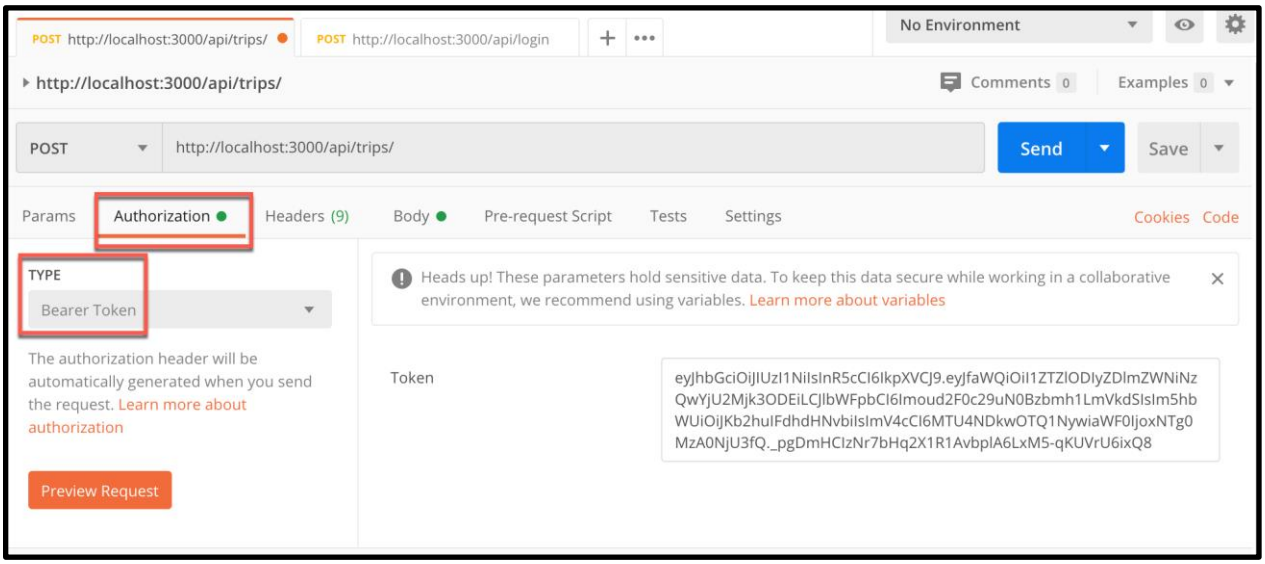
14. Use Postman to attempt to add a trip without first authenticating and you should receive an HTTP status 401–Unauthorized.



15. Using Postman to call the new **/login** endpoint with an email address and password will return the JWT token.



16. With the token just retrieved, you can use a built-in feature of Postman to pass that token in the header on a call to add a new trip and it will be successful. On the **Authorization** tab, choose **Bearer Token** from the **Type** drop-down menu and paste the token value in.



The backend Express application is now secured against unauthorized use.

Add Authentication to Angular SPA Frontend

Chapter 12 in the textbook covers the aspects of adding security to support authenticating users and ensuring only an authenticated user can perform certain actions. This section will summarize the steps specific to the Travlr Getaways scenario.

17. Create the storage class in the **/src/app** folder.

```
  ng generate class storage
```

```
import { InjectionToken } from '@angular/core';
export const BROWSER_STORAGE = new
InjectionToken<Storage>('Browser Storage', {
 providedIn: 'root',
 factory: () => localStorage
});
```

18. Create the user class in the **/src/app/models** folder.

```
  ng generate class user
```

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

19. Create the authresponse class in the **/src/app/models** folder.

```
  ng generate class authresponse
```

```
export class AuthResponse {
 token: string;
}
```

20. Create the authentication service in the **/src/app/services** folder.

```
  ng generate service authentication
```

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

@Injectable({
 providedIn: 'root'
})
export class AuthenticationService {

 constructor(
```

```
    @Inject(BROWSER_STORAGE) private storage: Storage,
    private tripDataService: TripDataService
  ) { }

  public getToken(): string {
    return this.storage.getItem('travlr-token');
  }

  public saveToken(token: string): void {
    this.storage.setItem('travlr-token', token);
  }

  public login(user: User): Promise<any> {
    return this.tripDataService.login(user)
      .then((authResp: AuthResponse) =>
this.saveToken(authResp.token));
  }

  public register(user: User): Promise<any> {
    return this.tripDataService.register(user)
      .then((authResp: AuthResponse) =>
this.saveToken(authResp.token));
  }

  public logout(): void {
    this.storage.removeItem('travlr-token');
  }

  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;
    }
  }

  public getCurrentUser(): User {
    if (this.isLoggedIn()) {
      const token: string = this.getToken();
      const { email, name } =
JSON.parse(atob(token.split('.')[1]));
      return { email, name } as User;
    }
  }
}
```

21. Update the **/src/app/services/trip-data-service.ts** component to include authentication calls to the new API endpoints.
    a. Include imports and inject the browser storage component.


```
import { User } from '../models/user';
import { AuthResponse } from '../models/authresponse';
import { BROWSER_STORAGE } from '../storage';

@Inject(BROWSER_STORAGE) private storage: Storage
```

b. Add the **login()**, **register()**, and **makeAuthApi()** functions at the bottom of the file.


```
public login(user: User): Promise<AuthResponse> {
```

```
    return this.makeAuthApiCall('login', user);
  }

  public register(user: User): Promise<AuthResponse> {
    return this.makeAuthApiCall('register', user);
  }

  private makeAuthApiCall(urlPath: string, user: User):
 Promise<AuthResponse> {
    const url: string = `${this.apiBaseUrl}/${urlPath}`;
    return this.http
      .post(url, user)
      .toPromise()
      .then(response => response.json() as AuthResponse)
      .catch(this.handleError);
  }
```

22. Create a navigation bar component to hold the "Log in" link.

```
ng generate component navbar
```

navbar.component.html:

```
<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-item nav-link active" routerLink="list-
trips">Trips<span class=" sr-only ">(current)</span></a>
       </div>
   </div>
   <div class="navbar-end ">
       <a class="navbar-item" routerLink="login"
*ngIf="!isLoggedIn()">
           <span class="has-icon-left">Log In</span>
       </a>
       <a class="navbar-item" (click)="onLogout()"
*ngIf="isLoggedIn()">
           <span class="has-icon-left ">Log Out</span>
       </a>
   </div>
</nav>
```

navbar.component.ts:

```
import { Component, OnInit } from '@angular/core';
import { AuthenticationService } from
'../services/authentication.service';

@Component({
 selector: 'app-navbar',
 templateUrl: './navbar.component.html',
 styleUrls: ['./navbar.component.css']
})
export class NavbarComponent implements OnInit {

 constructor(
   private authenticationService: AuthenticationService
 ) { }
 ngOnInit() { }
 public isLoggedIn(): boolean {
   return this.authenticationService.isLoggedIn();
 }
 private onLogout(): void {
   return this.authenticationService.logout();
 }
}
```

23. Create a new homepage with a message prompting the user to log in.

```
  ng generate component home
```

```
<div class="container" *ngIf="!isLoggedIn()">Please login to
continue</div>
```

24. Replace the existing static navbar HTML in **/src/app/app.component.html** with an Angular selector to bring in the new navigation component.

```
<app-navbar></app-navbar>
```

25. Create a new login component.

```
  ng generate component login
```

login.component.html:

```
<div class="row">
   <div class="col-12 col-md-8">
       <h2>Login</h2>
       <form (ngSubmit)="onLoginSubmit(evt)">
           <div role="alert" *ngIf="formErrors" class="alert
alert-danger">{{ formError }}</div>
           <div class="form-group">
               <label for="email">Email Address</label>
```

```
                <input type="email" class="form-control"
id="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>
```

Login.component.ts:

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

@Component({
 selector: 'app-login',
 templateUrl: './login.component.html',
 styleUrls: ['./login.component.css']
})
export class LoginComponent implements OnInit {

 public formError: string = '';

 public credentials = {
   name: '',
   email: '',
   password: ''
 };

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

 ngOnInit() {}

 public onLoginSubmit(): void {
   this.formError = '';
   if (!this.credentials.email || !this.credentials.password) {
```

```
      this.formError = 'All fields are required, please try
again';
    } else {
      this.doLogin();
    }
  }

  private doLogin(): void {
    this.authenticationService.login(this.credentials)
      .then(() => this.router.navigateByUrl('#'))
      .catch((message) => this.formError = message);
  }
}
```

26. Add new routes to **/src/app/app-routing.component.ts**.

```
const routes: Routes = [
    { path: 'add-trip', component: AddTripComponent },
    { path: 'edit-trip', component: EditTripComponent },
    { path: 'login', component: LoginComponent },
    { path: 'list-trips', component: TripListingComponent },
    { path: '', component: HomeComponent, pathMatch: 'full' }
]
```

Notice that the default catch-all path at the end of the list has been changed to route to the new **home** component.

27. Modify the trip-card component to add conditional logic controlling the output of the "Edit" button.
    a. Add *ngIf="isLoggedIn()" to the ** ** tag containing the button.

trip-card.component.html

app_admin > src > app > trip-card > trip-card.component.html >

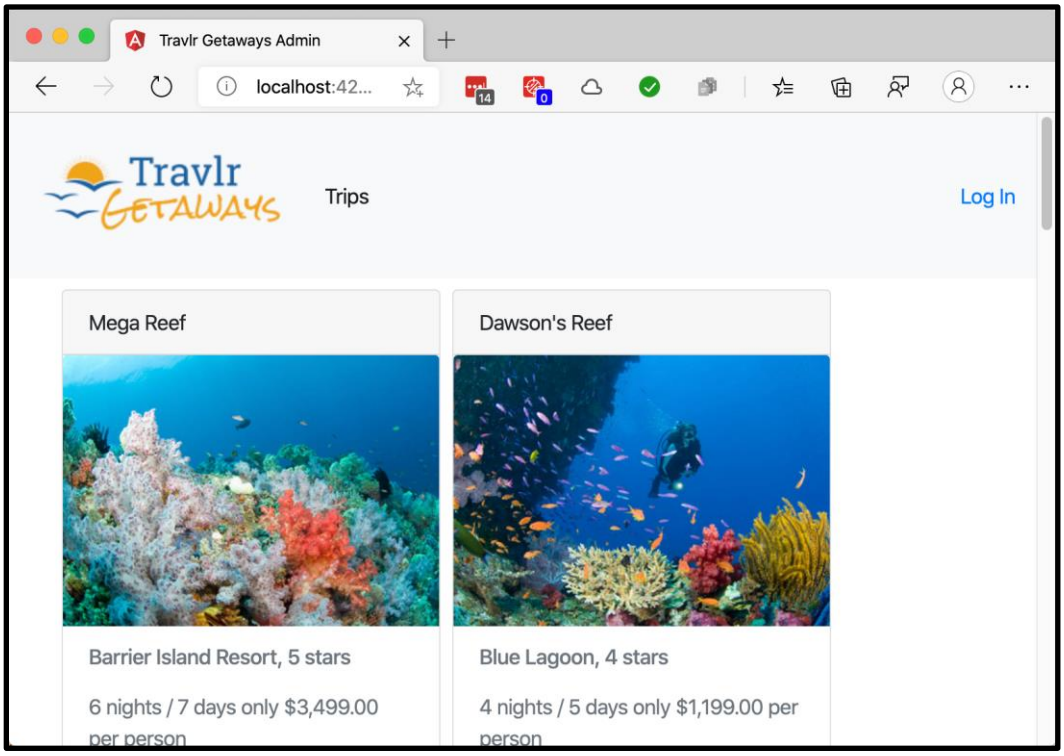
```
1   <div class="card ml-4" style="width: 18rem;">
2       <div class="card-header">{{ trip.name }}</div>
3       <img src="assets/images/{{ trip.image }}" alt="
4       <div class="card-body">
5           <h6 class="card-subtitle mb-2 text-muted">{
6           <p class="card-subtitle mt-3 mb-3 text-mute
7           <p class="card-text" [innerHTML]="trip.desc
8           <div *ngIf="isLoggedIn()">
9               <button (click)="editTrip(trip)" class=
10          </div>
11      </div>
12  </div>
```

b. Add the **isLoggedIn()** function to the backing component class.

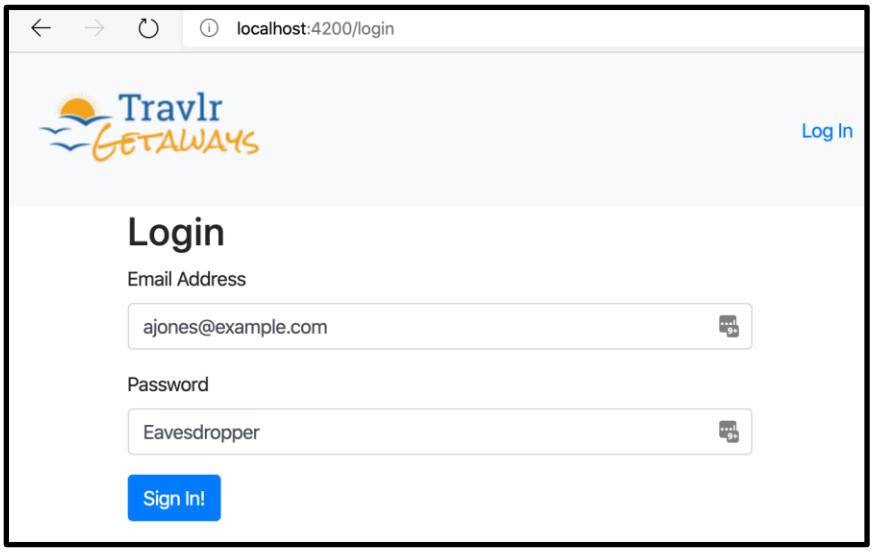

```
import { AuthenticationService }
    from '../services/authentication.service';

private authenticationService: AuthenticationService

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

28. Repeat this process to conditionality render the **Add Trip** button in the trip-list component.

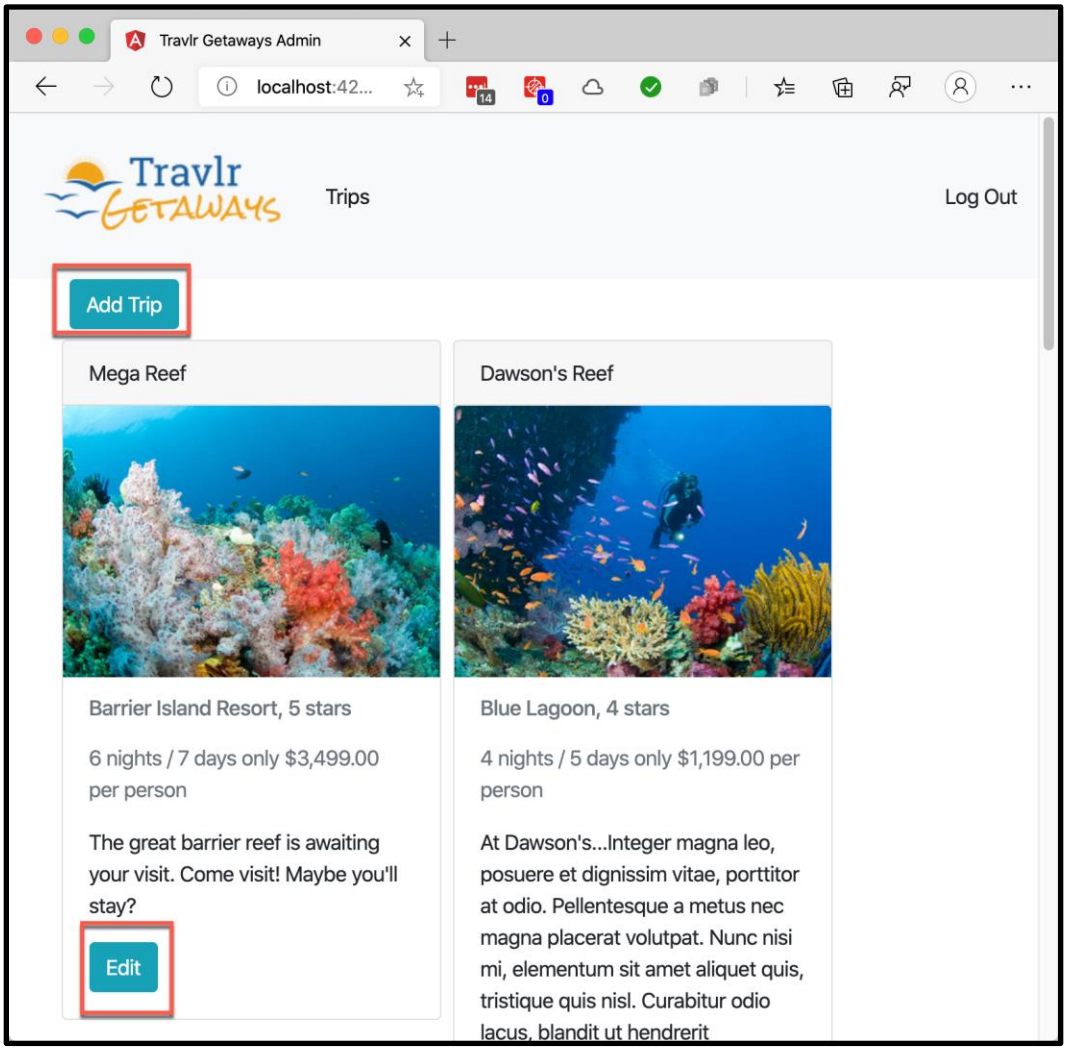
29. If you run the application, you will see the following:



30. Clicking the **Log In** link in the navbar will navigate the SPA to the login page.



31. After logging in, you will see that the **Add Trip** and **Edit** buttons are now visible and enabled.



Notice that the **Add Trip** and **Edit** buttons are now visible.