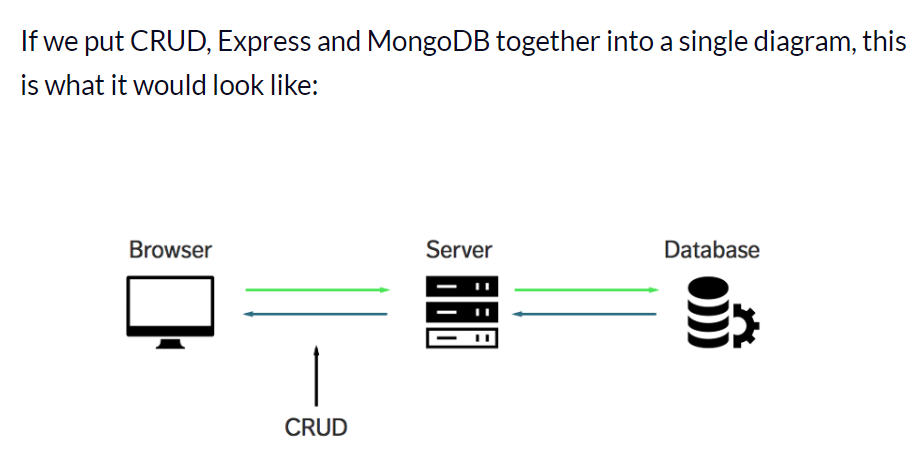
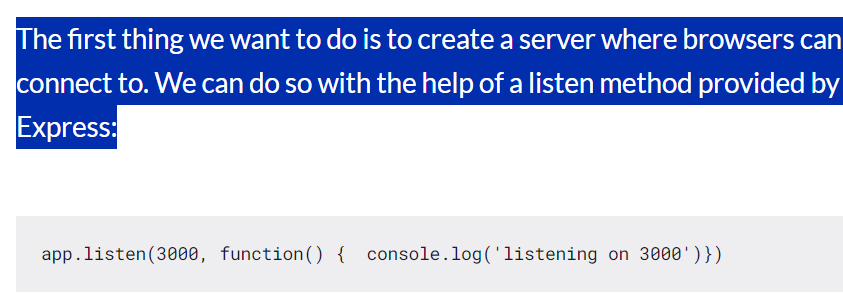
**416 to 425 CRUD operations**



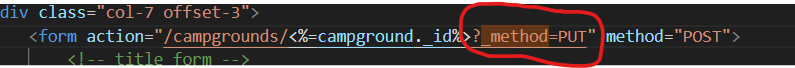
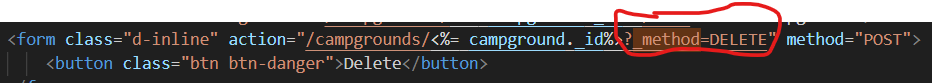


**\*methodOverride** is another middleware for Express that allows you to use HTTP verbs such as PUT or DELETE in HTML forms, which are traditionally limited to GET and POST. So first we import it



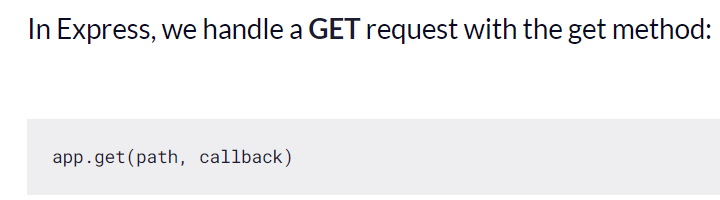
Then we enable it using following in App.js

The **\_method** parameter specifies the query parameter or form field name that will be used to override the HTTP method. So, if a form submission includes a field with the name **\_method** and a value of PUT or DELETE***, the methodOverride middleware will intercept the request and modify the HTTP method to PUT or DELETE, respectively.***

These are snips from show.ejs and edit.ejs******

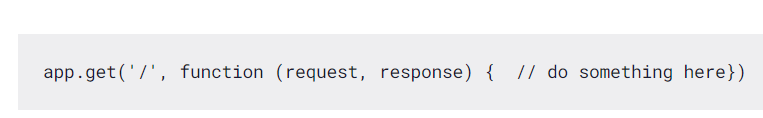
*No we’ve the CRUD operations :*

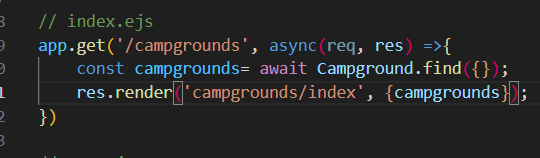
***READ: (get)***

* The **READ** operation is performed by browsers whenever you visit a webpage.
* Under the hood, browsers sends a **GET** request to the server to perform a READ operation.
* **The reason we see the “cannot get /”** error is because we have yet to send anything back to the browser from our server.
* 
* **The first argument, path**, is the path of the GET request. It’s anything that comes after your domain name.

When we’re visiting localhost:3000, our browsers are actually looking for localhost:3000/. The path argument in this case is /.

* ***The second argument is a callback function****that tells the server what to do when the path is matched. It takes in two arguments, a request object and a response object:*



YELPCAMPS read function 

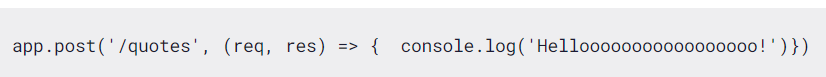
***Create: (post)***

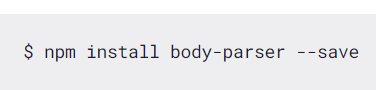
* The **CREATE** operation is performed only by the browser if a **POST** request is sent to the server. This POST request can triggered either with JavaScript or through a **<form> element.**
* **We used form to do it in yelpcamp**
* We need to have three things on this form element:

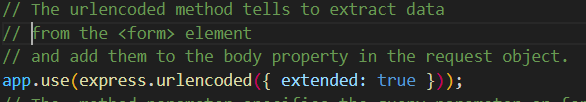
1. An action attribute
2. a method attribute
3. and name attributes on all <input> elements within the form

<form **action="/quotes" method="POST"**> <input type="text" placeholder="name" **name="name"**> <input type="text" placeholder="quote" **name="quote"**> <button type="submit">Submit</button></form>

1. The action attribute tells the browser where to navigate to in our Express app. In this case, we’re navigating to /quotes.
2. The method attribute tells the browser what to request to send. In this case, it’s a POST request.

* On our server, we can handle this POST request with a post method that Express provides. It takes the same arguments as the GET method: 
* \* Turns out, Express doesn’t handle reading data from the <form> element on it’s own. We have to add another package called ***body-parser*** to gain this functionality.



* Express allows us to add middlewares like body-parser to our application with the **use method.**
* **Middlewares are basically plugins that change the request or response object before they get handled by our application.**
* **Make sure you place body-parser before your CRUD handlers!**
* const bodyParser= require('body-parser')
* **Although we can use direct express for it extract data from the form**

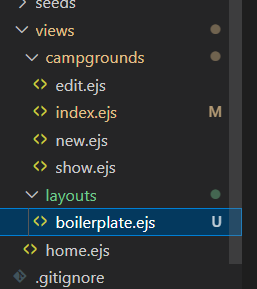
**434. A New EJS Tool For Layouts: EJS mate**

* **EJS Mate** is a middleware for Express that provides additional functionality to the EJS templating engine.
* It includes features such as partials, layouts, and helpers that make it easier to create and manage views in a CRUD project.
* With EJS Mate, you can modularize your views, reuse code, and create a consistent layout across pages.

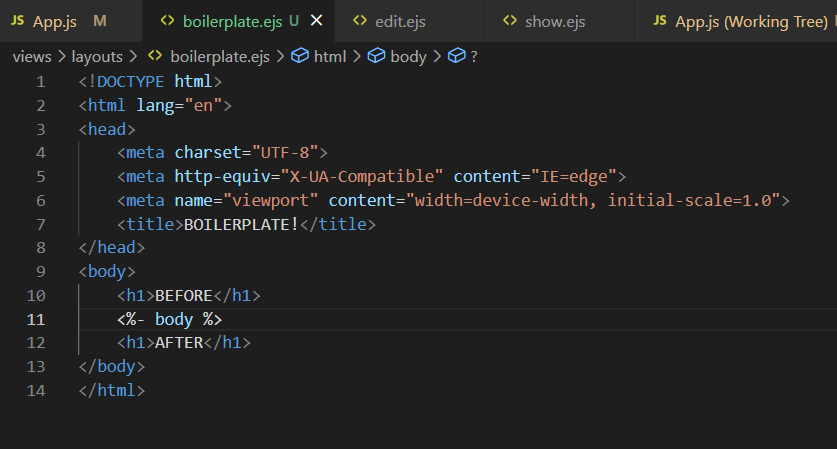
layout(view):

When called anywhere inside a template, requests that the output of the current template be passed to the given view as the body local.

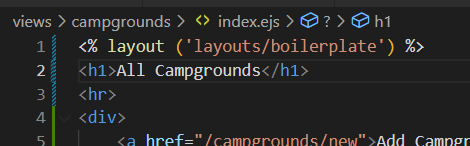
Inside views we created layout folder



Boilerplate.ejs contains boilerplate for ejs

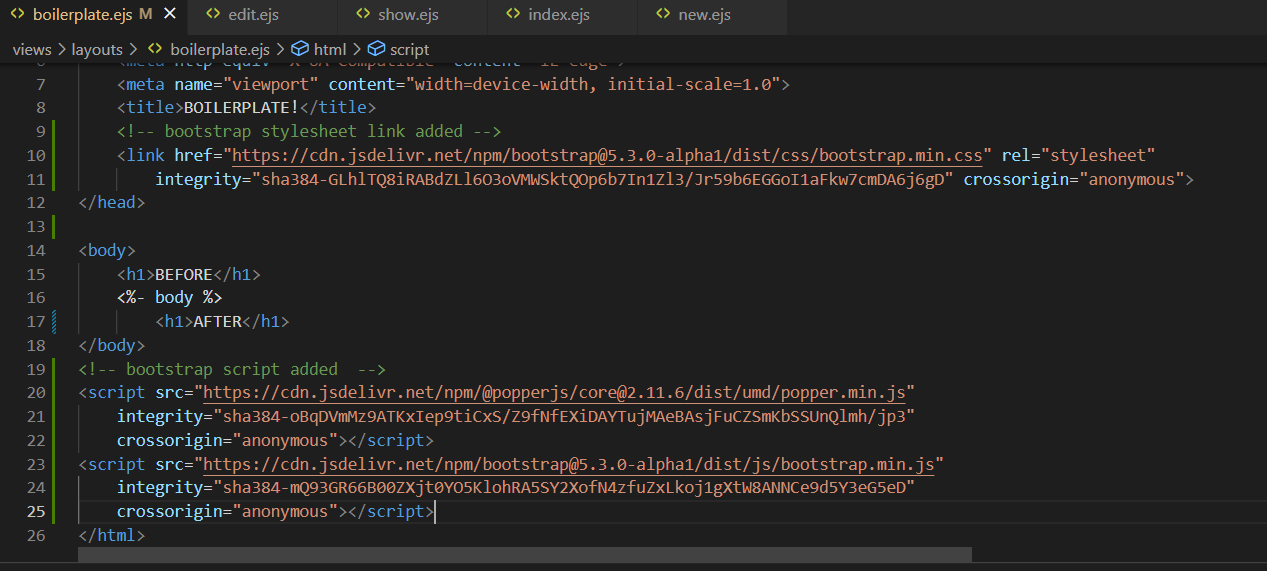


We just need to add ***“ <% layout ('layouts/boilerplate') %>”*** to access the boilerplate

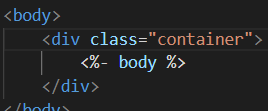


***435. Bootstrap5! Boilerplate***

Add link and script of bootstrap to the boilerplate.ejs file.



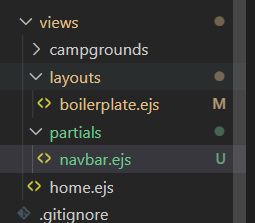
Added container class to the div enclosing the body tag of ejs mate



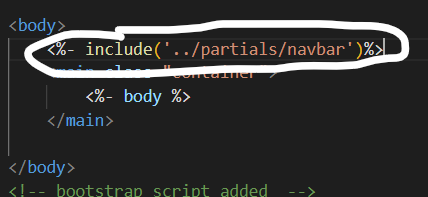
***436. Navbar Partial***

Created a new navbar.ejs file to store navbar from bootstrap which will be **included** in boilerplate.ejs.

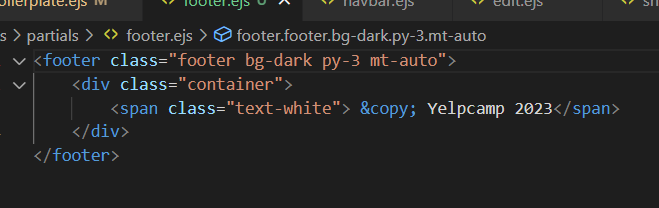
To do so we created a folder named **partials** inside view in which we stored **navbar.ejs**

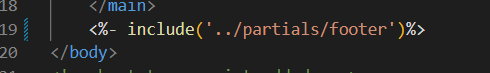


We include navbar partial in boilerplate using following code fragment

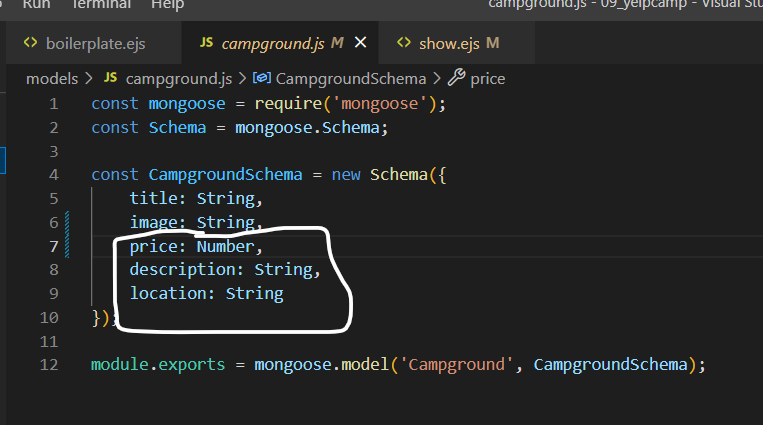


***437. Footer Partial***

We created a partial named **footer.ejs**  initially then added bootstrap style to the footer. 

Then included the partial footer.ejs file to the boilerplate.ejs

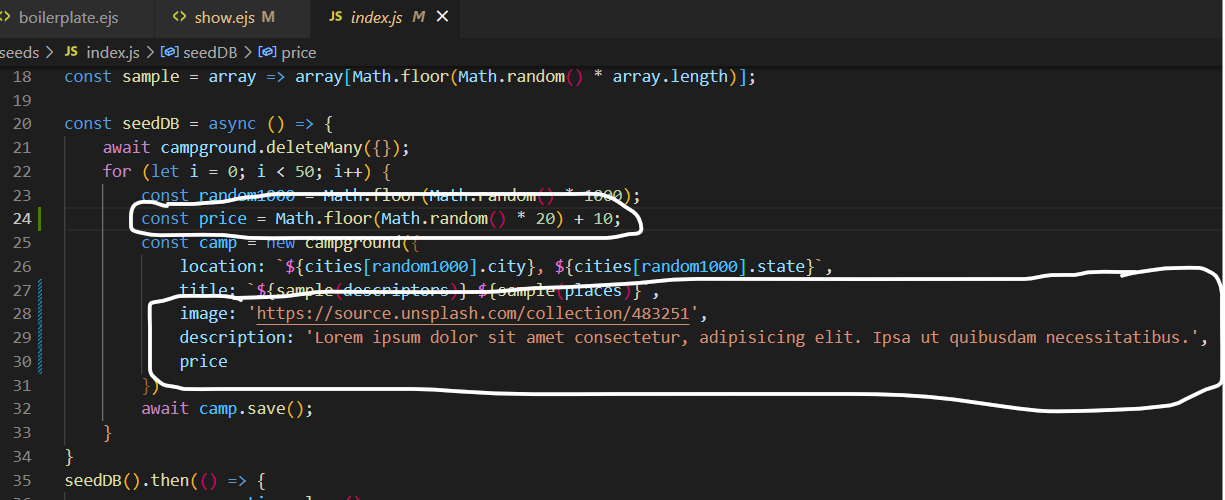
***438. Adding Images***

Added image, price and description to **models/campgrounds.js **

Added image to **seeds/index.js** where it seeded.

I’ve added unsplash collection url which shows random image every time it is seeded

And same for price for which we’ve used random number generator

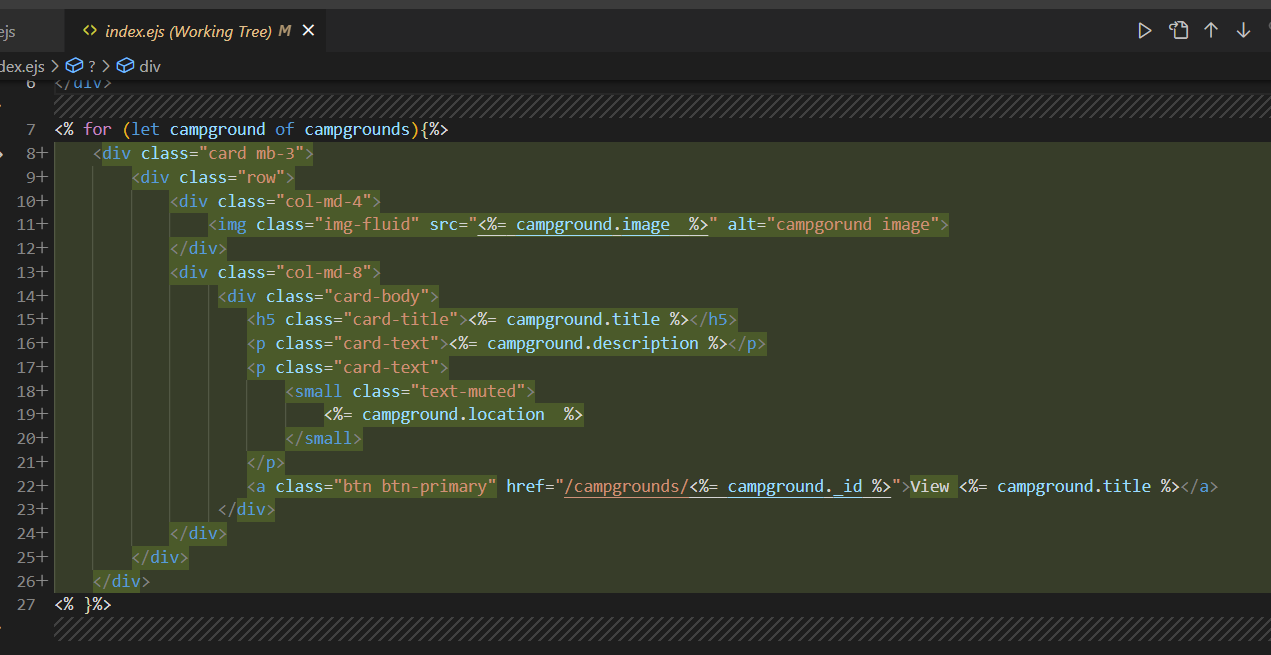


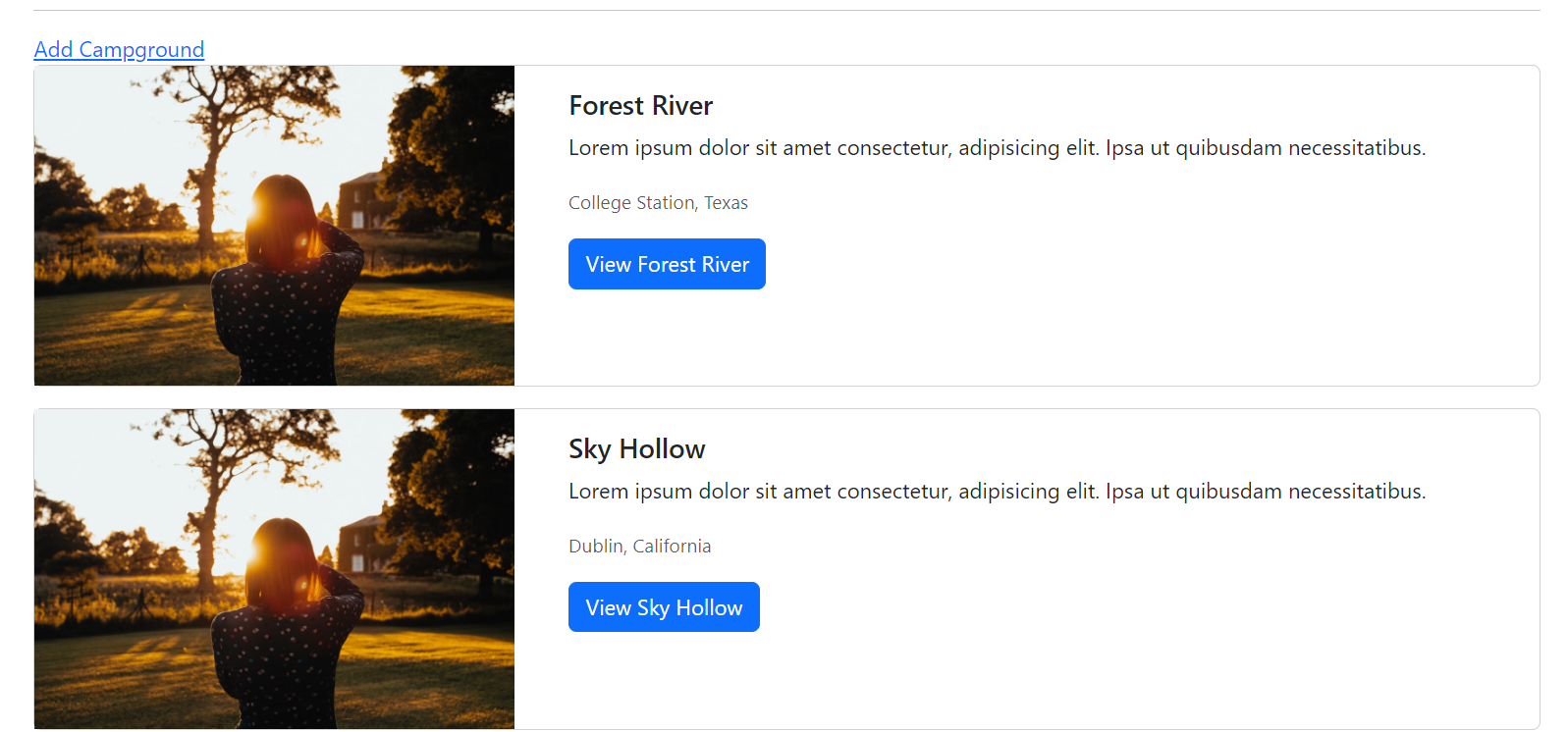
After that we’ve added price, description and image to **views/campgrounds/show.ejs**  so that it is displayed on show page.

***439. Styling Campgrounds Index***

Added card logic to **index.ejs**  to better style index.ejs page

It now shows image, title, location and a show button which redirects to **show.ejs**

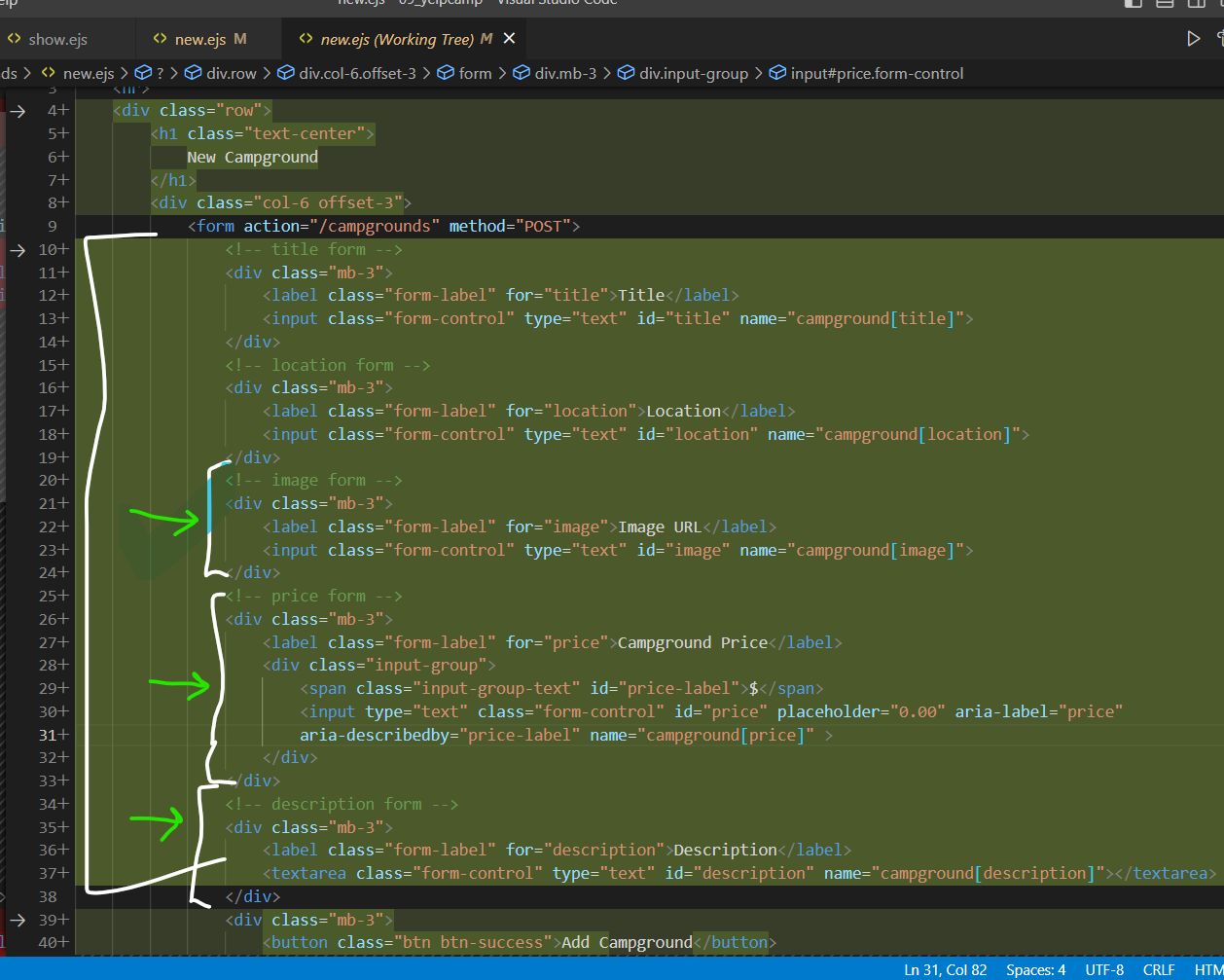
****

****

***440. Styling The New Form***

Added styles to the form using bootstrap in **new.ejs**  file to improvise the UI

Plus added form i.e label and input for **imageURL, price**  and **description**  as well to the folder.

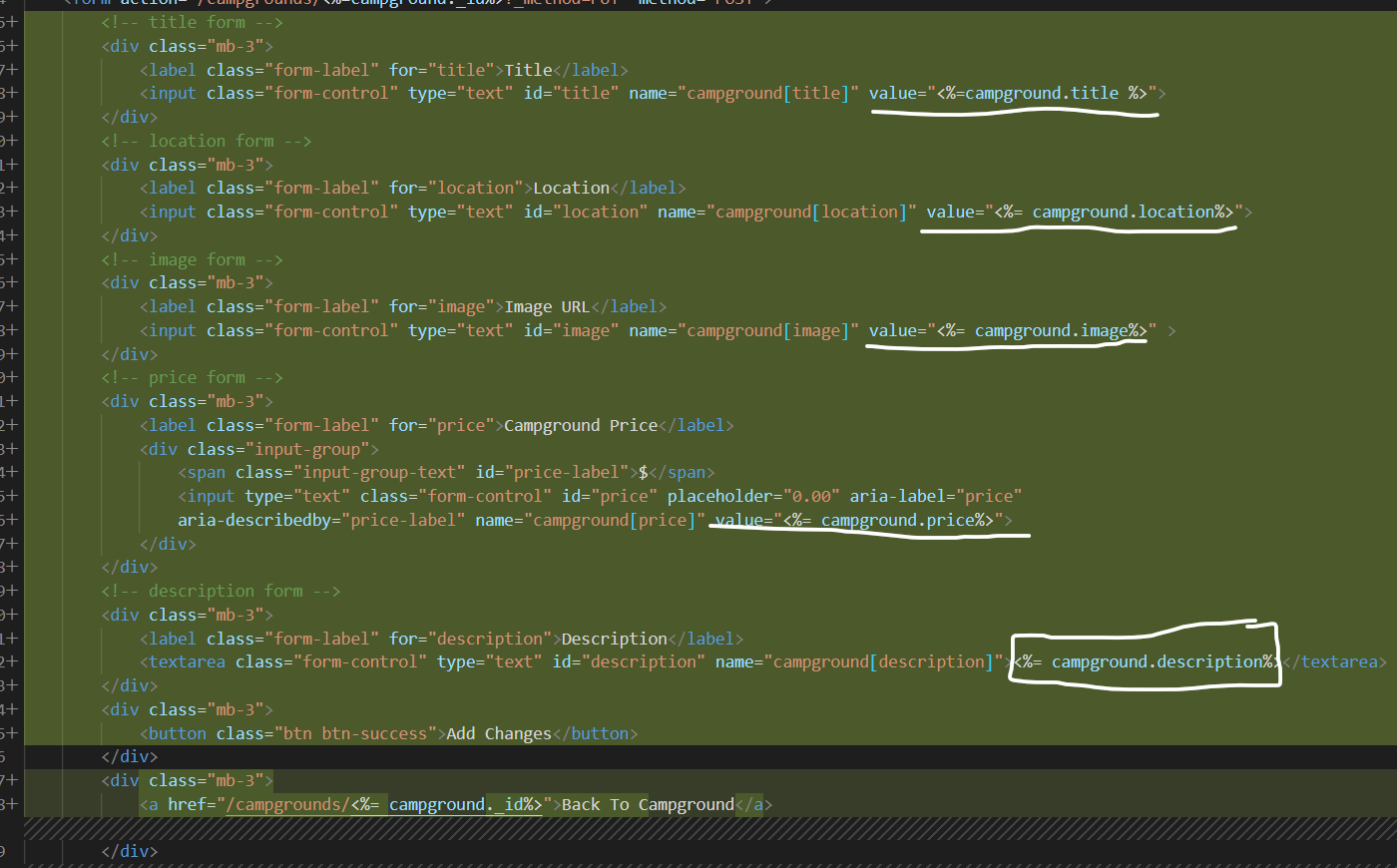


Finally it looks something like this



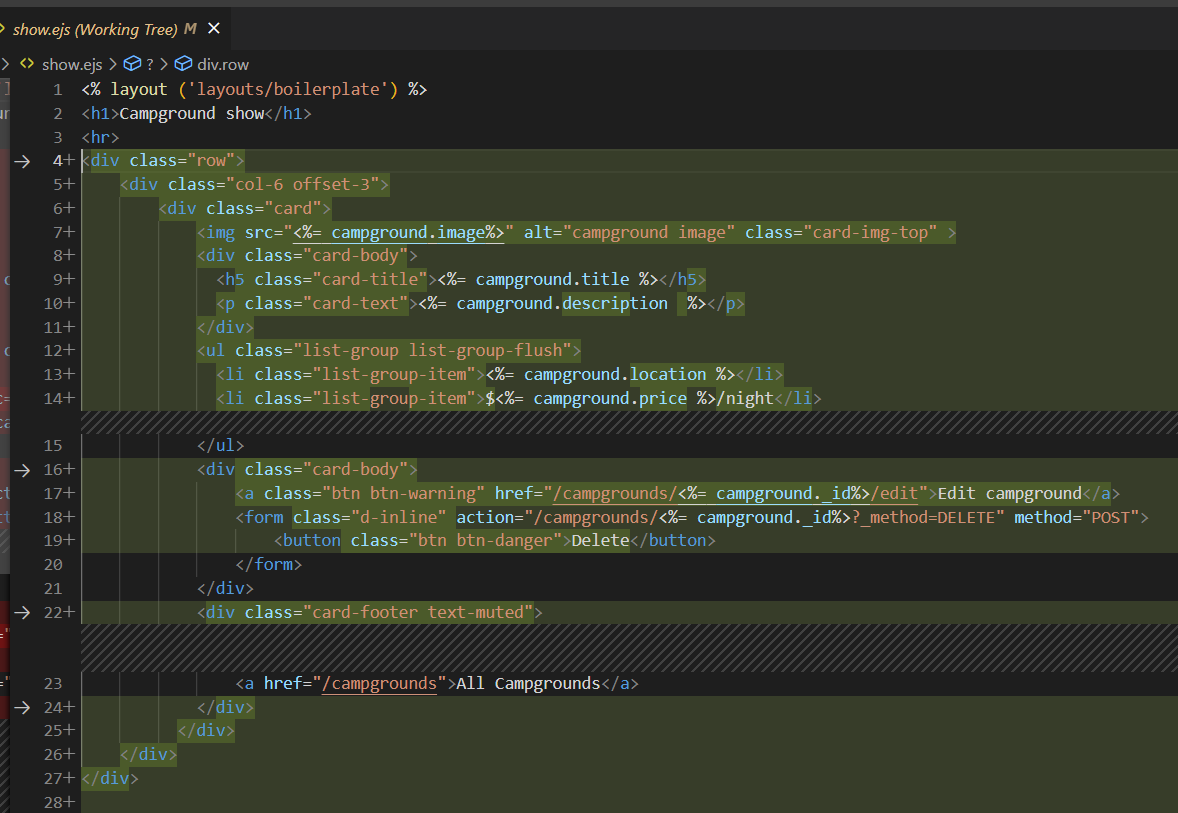
***441. Styling Edit Form***

In this part we just copied the form we created in **new.ejs**  and added it to the **edit.ejs** page with minor changes to neede to update the data which’ve been highlighted by with pen in below image.

Value can’t be added to textarea in description part of the form just like we added them into the input area of imageURL, price, location and title. It has to be just added as it is in between the textarea tags. 

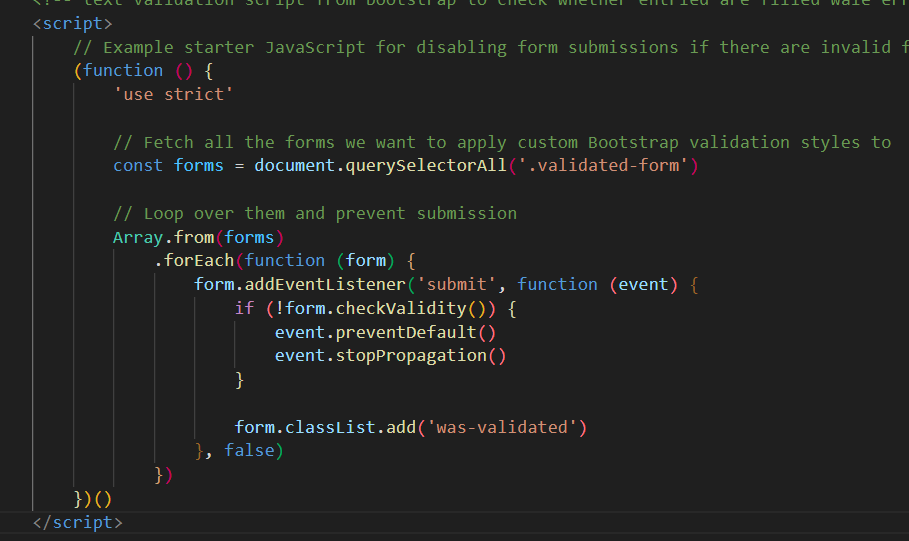
***442. Styling Show Page***

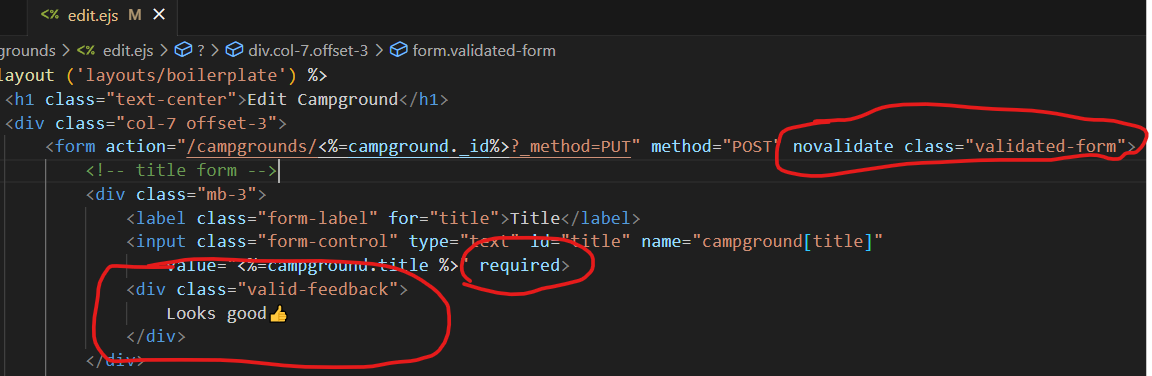
Added better styles using bootstrap to **show.ejs**  file. This was made using card template from bootstrap in which we showcased image the title, description, location and price along with two buttons dedicated to edit and delete the particular campground.

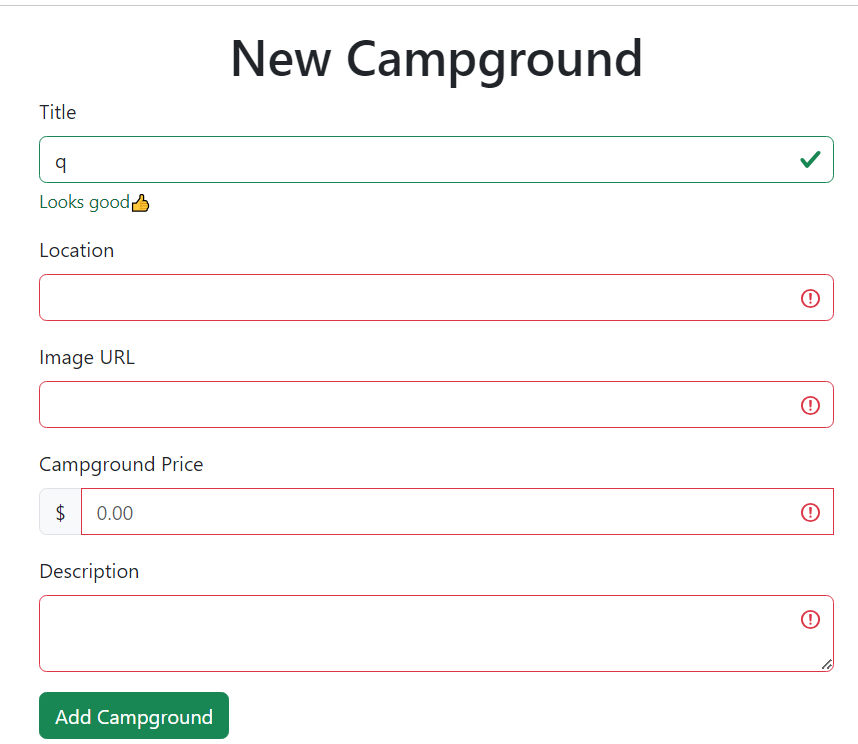


**Section43: Yelpcamp: Errors and Validating data**

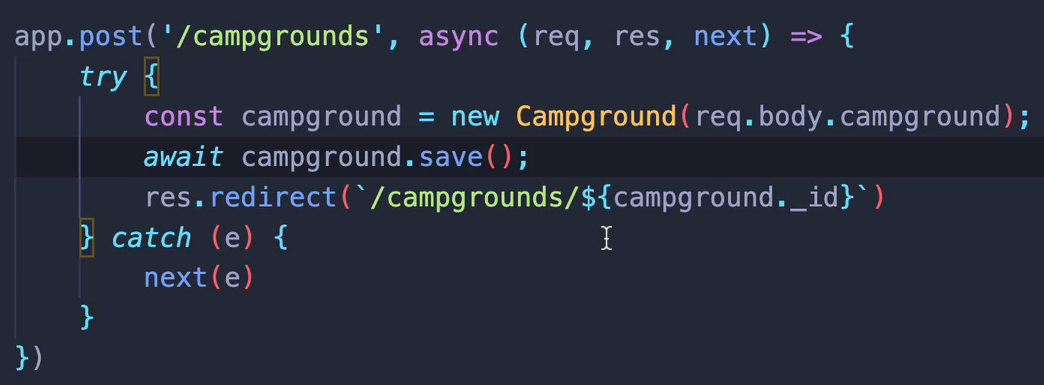
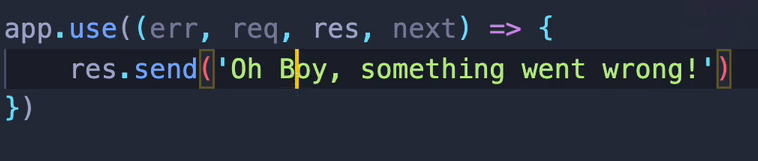
452. Client-Side Form Validations

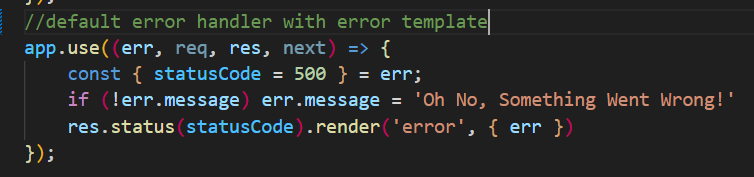
Added this following script to the boilerplate.ejs file which is taken from bootstrap website’s validation page.





453. Basic Error Handler

We did some basic error handling using try and catch in app.js

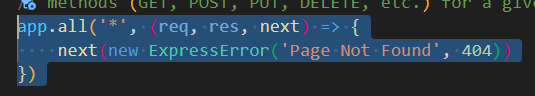
And at last we created a default error handler 

**454. Defining ExpressError Class**

We started with creating a utils folder for utilities function.

In web development, the term "**utils**" refers to **utility functions or modules** that are ***created to perform frequently used and often generic tasks that are not specific to a particular feature or functionality of the web application***. These can include functions that handle tasks such as data validation, error handling, formatting data, and other commonly required tasks.

The **main reason for creating utils** in web apps **is to improve code reusability, maintainability, and readability which futher can help reduce the amount of code duplication, which in turn helps to minimize the risk of errors and bugs in the codebase.**



Above given is a middleware function in an Express.js web application **that handles all HTTP requests for routes that do not exist.**

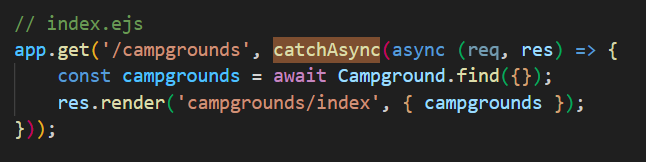
**\*HERE app.all()** method, is **used to handle all HTTP methods (GET, POST, PUT, DELETE, etc.) for a given route**. In this case, the **route is specified as '\*'**, which is a **wildcard that matches all routes**. The middleware function takes **three arguments**: **req (the request object), res (the response object), and next (a function that is used to pass control to the next middleware function in the chain)**.

In summary, the code **app.all('\*', (req, res, next) => { next(new ExpressError('Page Not Found', 404)) })** is a middleware function that **handles all HTTP requests for routes that do not exist in an Express.js web application**. It creates a new ExpressError object with a message of ***'Page Not Found'*** and **a status code of *404***, and passes it to the next function to be handled by the error-handling middleware function.

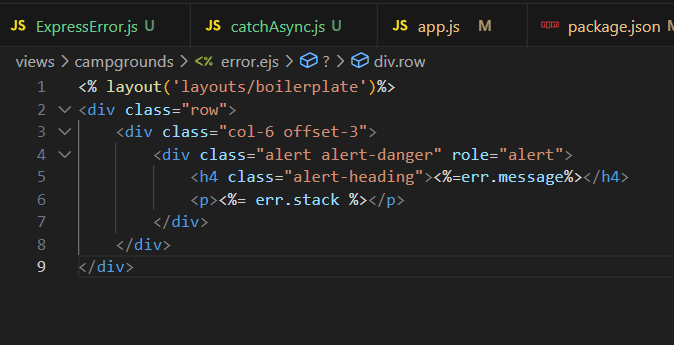
**455. More Errors**

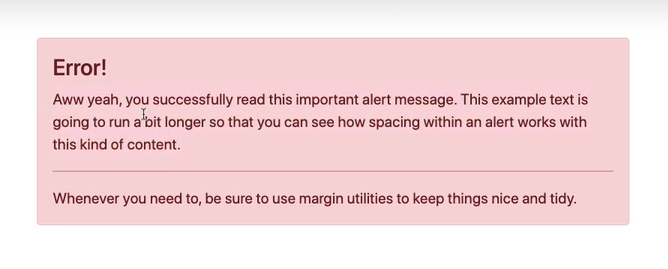
In this part **we created catchAsync** and **ExpressError JS files** inside **utils**

We wrapped all our CRUD endpoints inside **catchAsync** as it is basically a **function we create and use as a wrapper to catch any asynchronous errors when our route functions are executed.**



**456. Defining Error Template**

In this part we created error.ejs template

It will show something like this

**457. JOI Schema Validations and 458. JOI Validation Middleware**

In part we’re gonna add [JOI validations](https://joi.dev/api/?v=17.9.1)

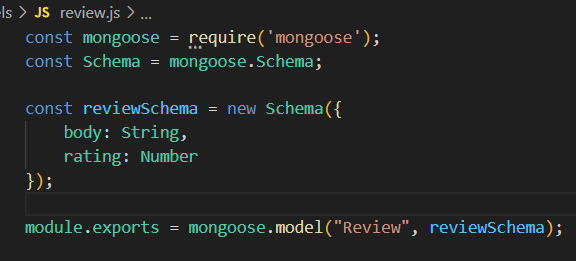
***\*JOI schema validation ensures data submitted to an application meets specific rules, preventing errors and inconsistencies. It defines rules for data validation, including type, format, and required fields.***

To do so we created a **schema.js** in the root folder and added JOI validation code to it 

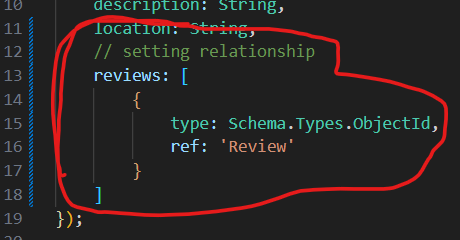
**Section 46: Yelpcamp: Adding the Reviews Model**

**476. Defining The Review Model**

We started by creating new model **review.js** to have reviews part in it i.e. **body(comment)** and **rating.**

****

Then we setup relationship campground by giving reference to the review model

****

**477. Adding The Review Form**

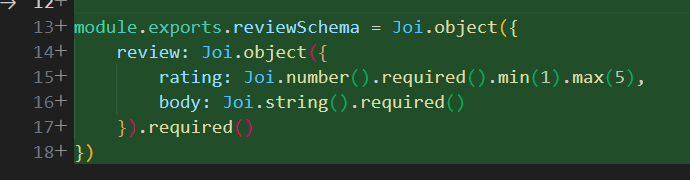
In this part we’re gonna add a form in **show.ejs** to take reviews from the user.

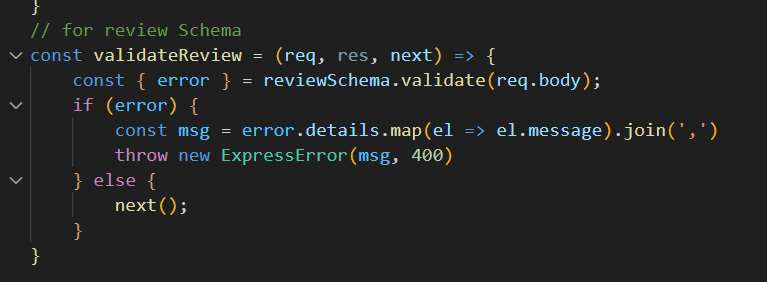


**478. Creating Reviews**

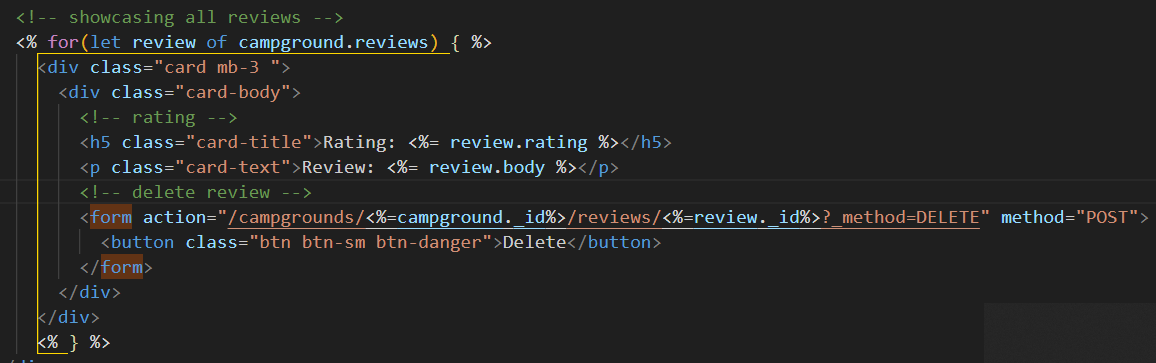
In this lecture we created routes for ccreating reviews 

**479. Validating Reviews**

In this lecture we created validations using JOI in **schemas.js** ****

And in **app.js** it’s **validateReview** function 

**480. Displaying Reviews**

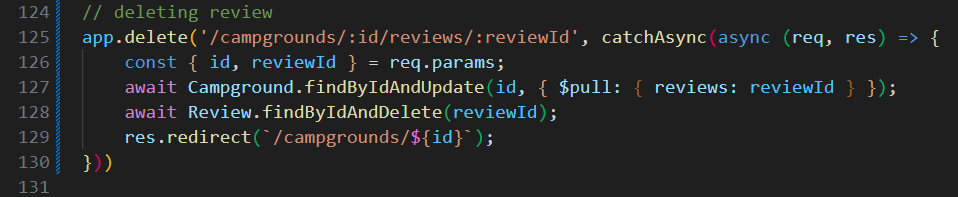
Showimg reviews in **show.ejs** 

**481. Styling Reviews**

Basic styling is done to shift review part to the side of show panel

**482. Deleting Reviews Play**

Created delete route for the review part



**483. Campground Delete Middleware**

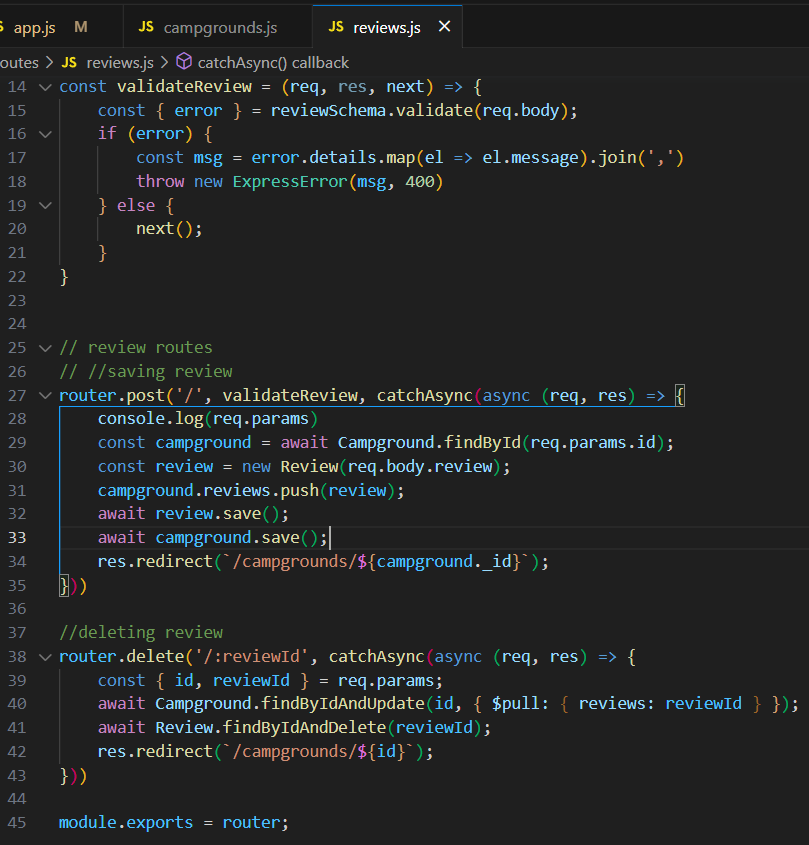
\*Suppose a case where campground is deleted so we’ve to make sure we also delete it’s reviews as well from review table for which we create delete middleware in campground.js schema



**Section 49: Yelpcamp restructuring and flash**

**498. Breaking Out Campground Routes and 499. Breaking Out Review Routes**

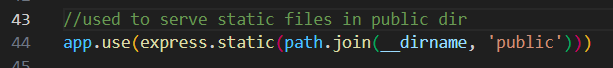
We create a folder route in which we added **campground routes and review routes.**

Here it is 

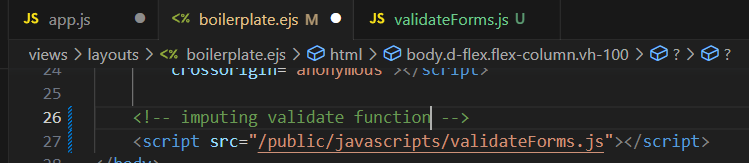
**500. Serving Static Assets**

We added the following fragment of code which **is used to serve static files (such as HTML, CSS, images, and scripts) from the "public" directory in an Express.js application.**

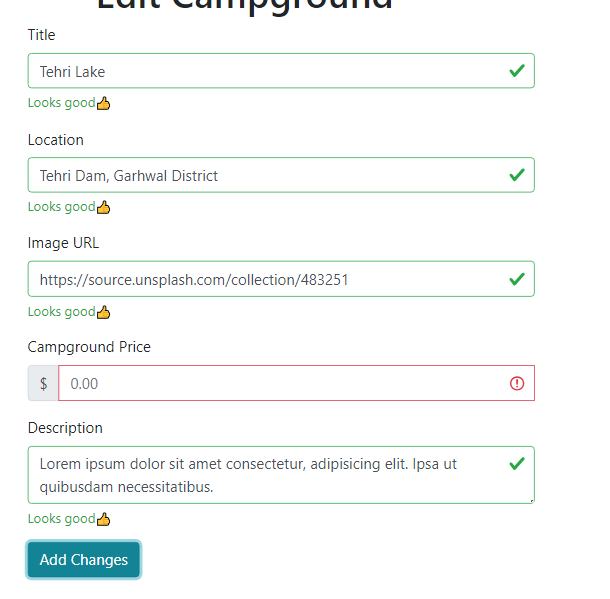
**It sets up a middleware that intercepts incoming requests for static files and responds with the appropriate file if it exists in the "public" directory.**

****

Then we created public/javascripts/validateForms.js

And using that public file we use it in boilerplate 

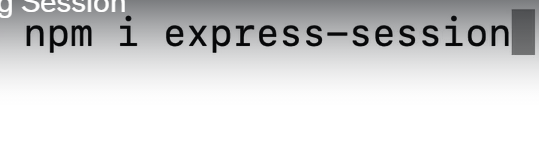
Humne validate form ka function public main dala and usko fir hum kahin bhi access kar ske hain just like humne usse boilerplate main use kiya!



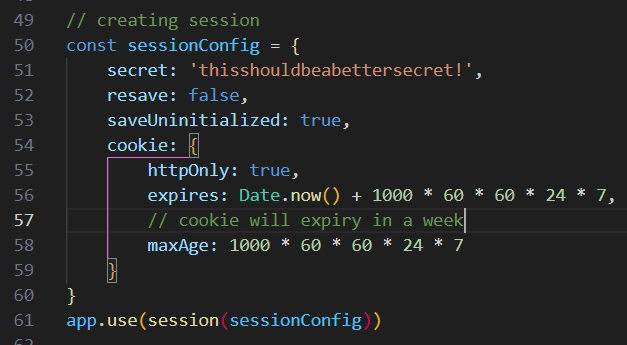
**501. Configuring Session**

In this lec we’re gonna add express session.

For it we install



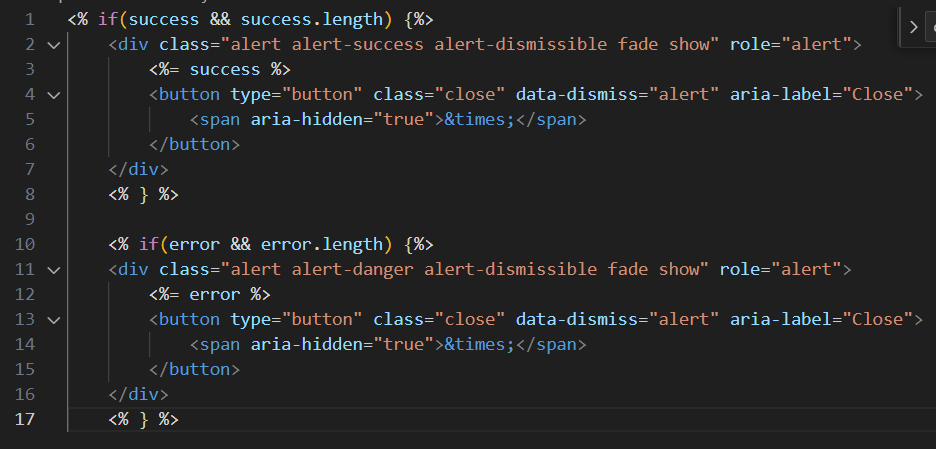
Then require it 

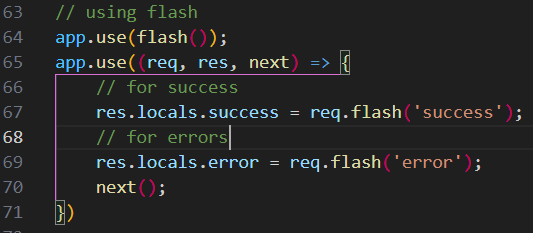


**502. Setting Up Flash, 503. Flash Success Partial, 504. Flash error Partial**

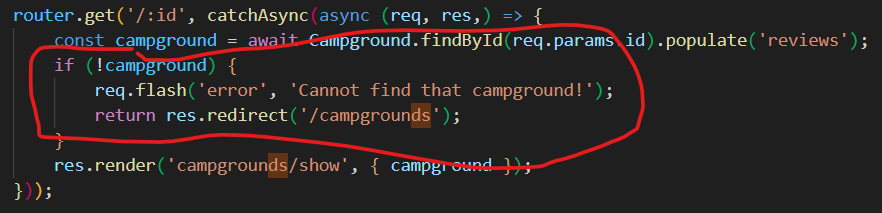
Install 



then we create partial for flash and then add the following code to app.js



then add error and success flash to routes.



**Section 51: Yelpcamp Adding in Authentication**

**518. Introduction to Passport**

**Passport is a Node.js middleware used for user authentication.**

It provides a simple way to authenticate users using various strategies such as local, OAuth, or OpenID and it  can be easily integrated into Node.js applications to protect routes and manage user sessions.

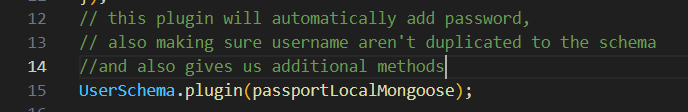
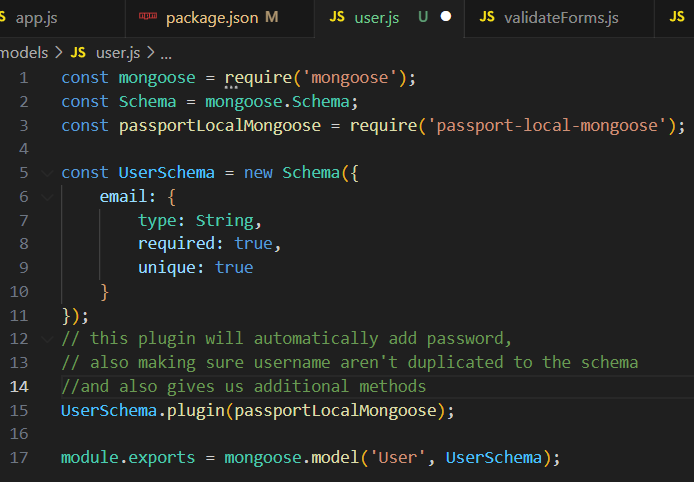
Here we’ll also be using **passport local and passport mongoose** as well.

**1-Passport-Local** is a **strategy that allows you to authenticate users with a username and password that are stored locally in your application.** It provides a simple way to authenticate users without relying on a third-party service.

**2-Passport-Mongoose** is a **strategy that allows you to authenticate users with a username and password that are stored in a MongoDB database.**

**519. Creating Our User Model**

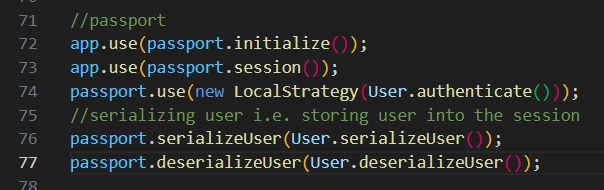
We started with installing **passport, passport-local** and  **passport-mongoose.** ****

We’ll now make new model **usersthis the new model file**

**520. Configuring Passport**

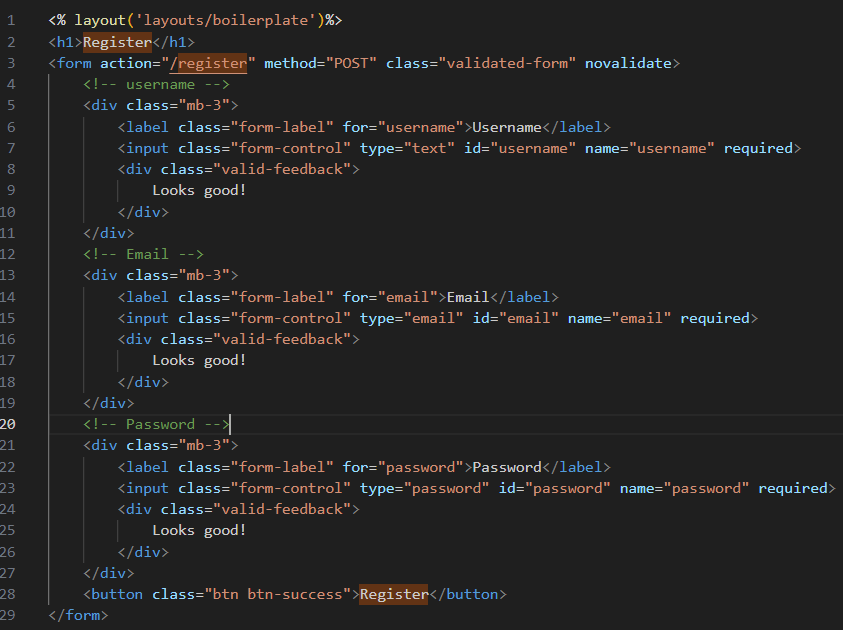
We require all the three packages

For login session we’re gonna use passport JS.

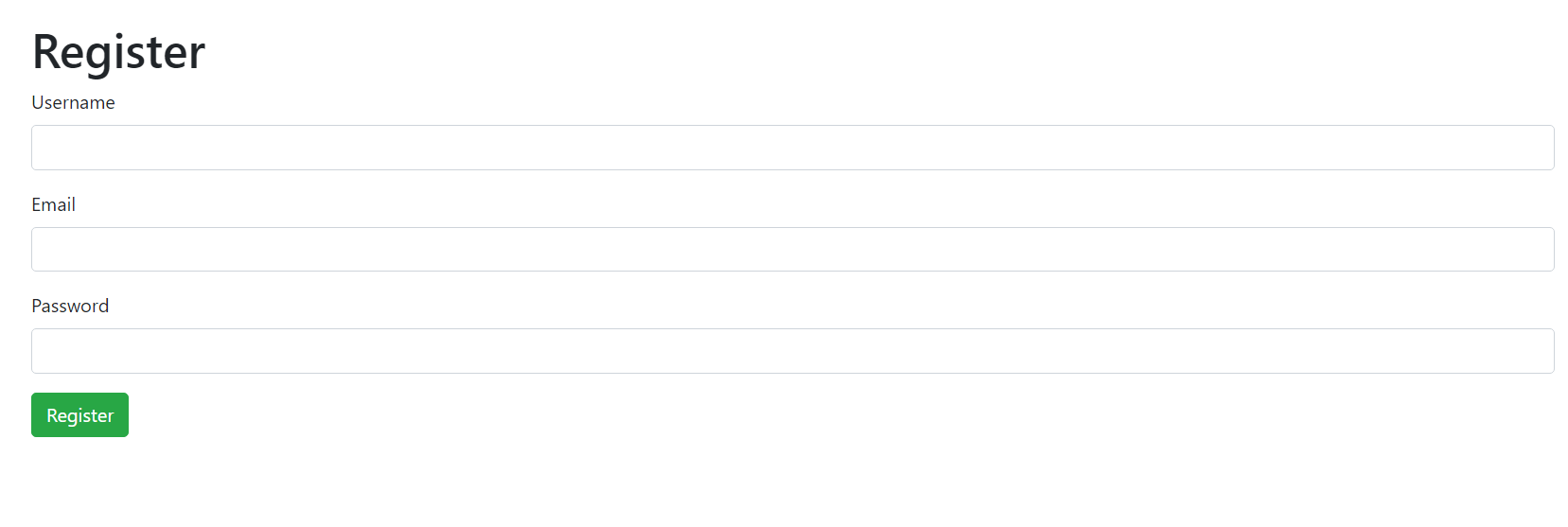


**521. Register Form**

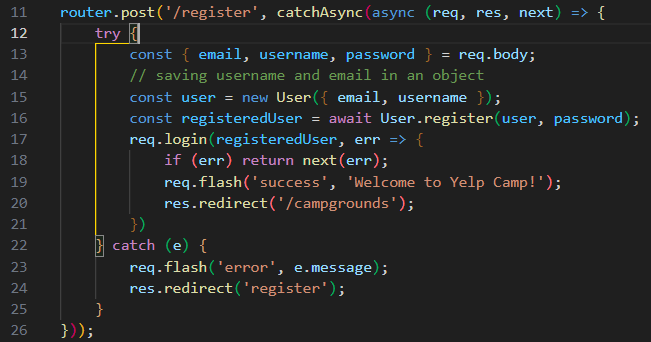
In this part we created a **register user form**



output:

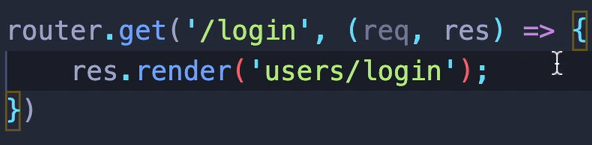


**522. Register Route Logic**

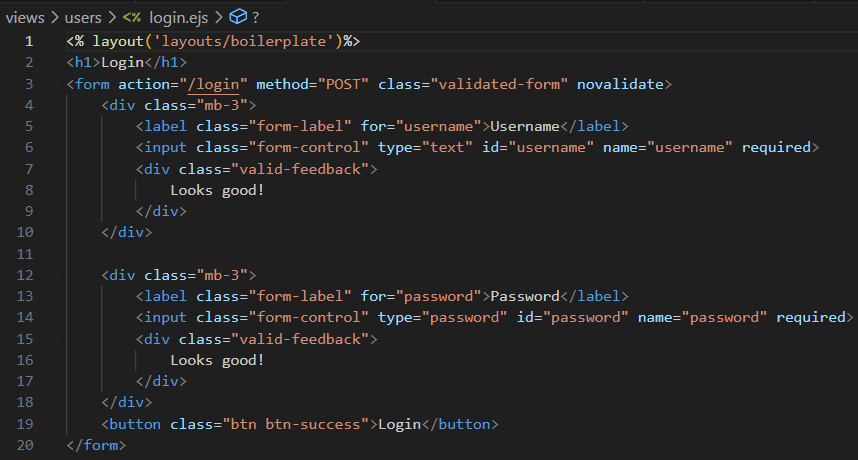
In this lecture we’ll create routes for register page 

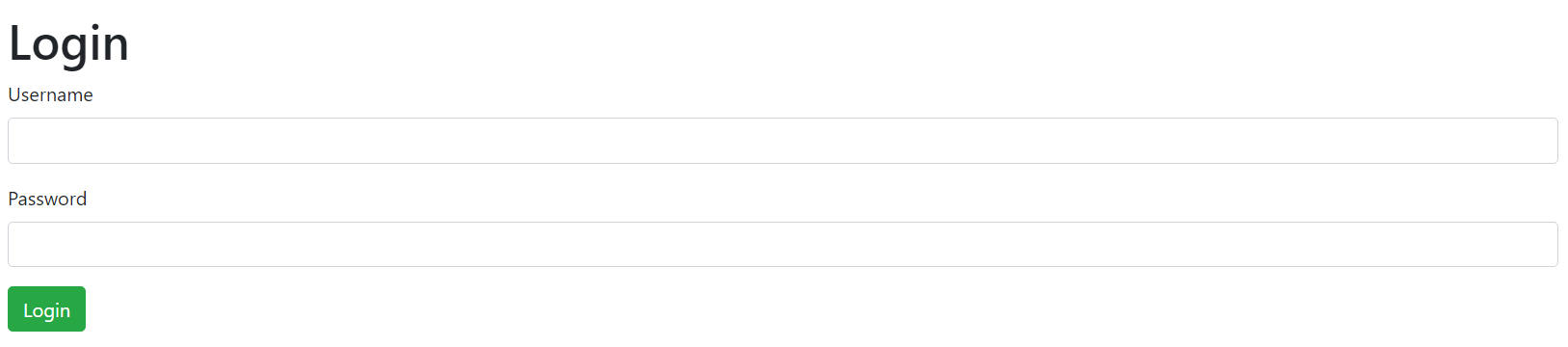
**523. Login routes**

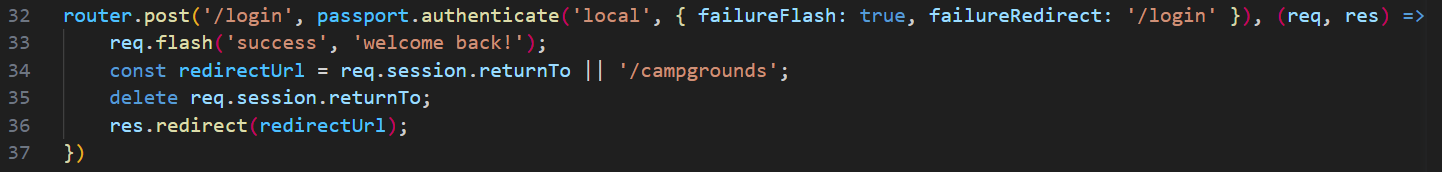
In this video we’re gonna create **login view and route(user.js).**

here’s what we write for rendering the login page initially in user.js route 

then we create view



Output 

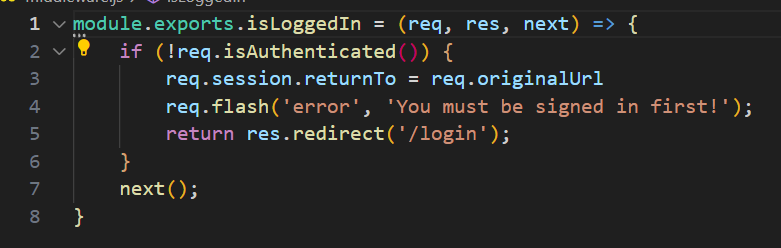
Route is given below for the login page 

**524. isLoggedIn Middleware**

In this video we’ll add **We can’t add new campground if we’re not loggedin.**

It is done using **isAuthenticated** methodfor **passportJS.**

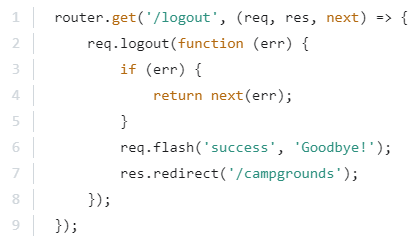
For it we create **middleware.js** file and add **isAuthenticated** method inside **isLoggedin method** to use that in required routes.

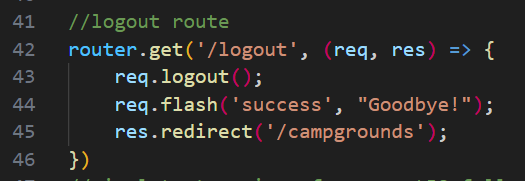


*Then import it from the middleware*

 The we can use it in required route while requesting the endpoints 

**526. Adding Logout and 525. IMPORTANT: Fixing Logout**

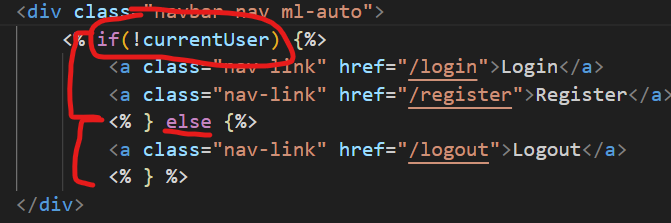
***In newer version use the following syntax for logout route ***

As we’re using the earlier version we’ll use this method 

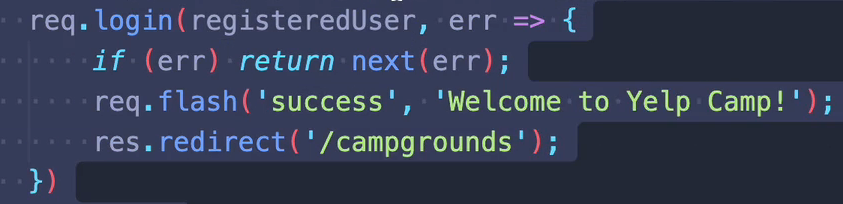
**527. currentUser Helper**

In this video we’ll change **the logout and register** buttons in **navbar**  to **Logout** button once user is loggen in.

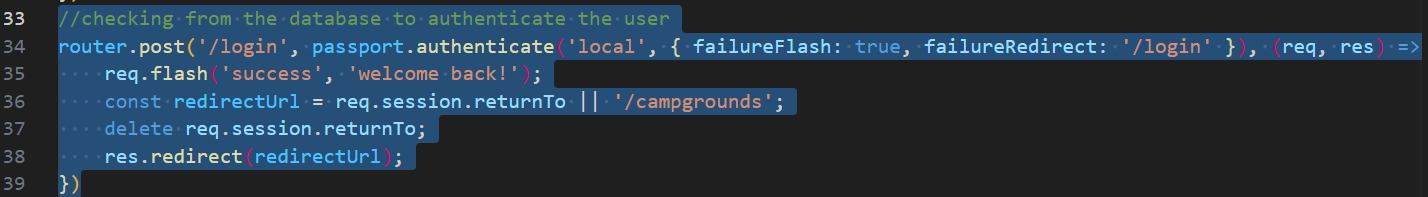
 

We use conditionals in navbar view to check user loggedin or not 

**528. Fixing Register Route**

Then we check in the register route by sending login route info about the new registered user to log him in 

**530. ReturnTo Behavior**

****

**In line 36**, a function is **defined to handle the successful authentication**.

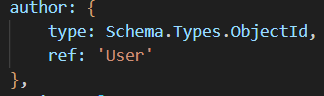
This function first checks if there is a returnTo property in the **req.session object**, **which is used to store the URL that the user was trying to access before they were redirected to the login page**. If there is a returnTo property**, its value is stored in a redirectUrl variable. Otherwise, the default value of /campgrounds is used.**

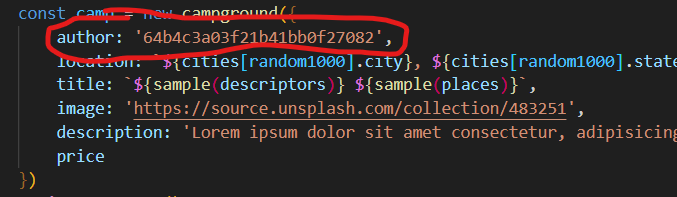
**On line 37, the returnTo property is deleted from the req.session object to prevent it from being used again in future requests.**

**On line 38, the user is redirected to the redirectUrl using the res.redirect method.**

**Section 51: Yelpcamp: Basic Authorisation**

**531. Adding an Author to Campground**

We’ll add a **author** field in campground model. 

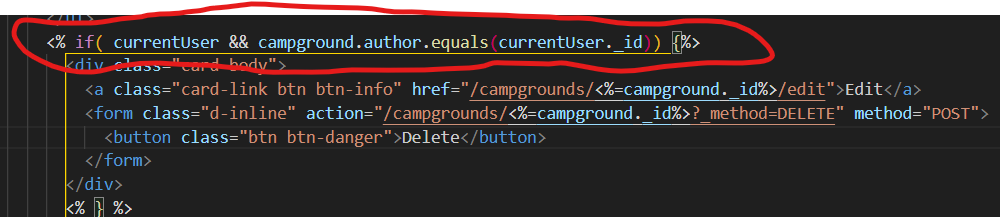
Then we update our seeds.js and user who added it. 

Then we added it to the show.ejs field as well.



**532. Showing and Hiding Edit/Delete**

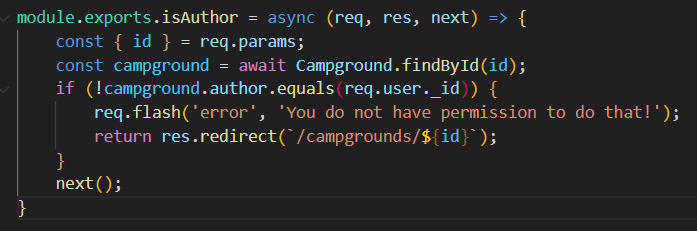
This lec shows edit and delete function based on **authorisation** by using conditionals in show template.

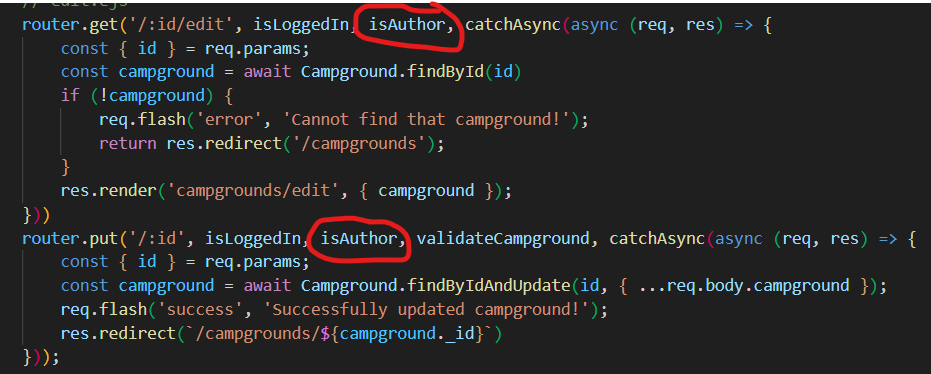


**533. Campground Permissions and 534. Authorization Middleware**

In this video **we’ll protect rou**tes and will allow only **to edit and delete an campground** if it is an **author of the campground.**

**This part is written in middleware.js**



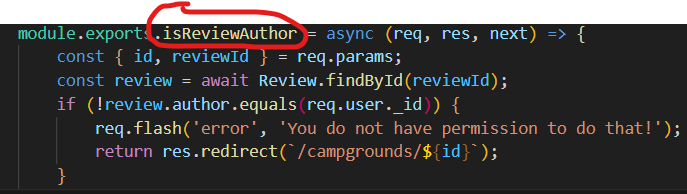
And then we use it in edit and delete routes 

Now if we try to edit the campground we didn’t write we’ll be redirected to campground page without any changes.

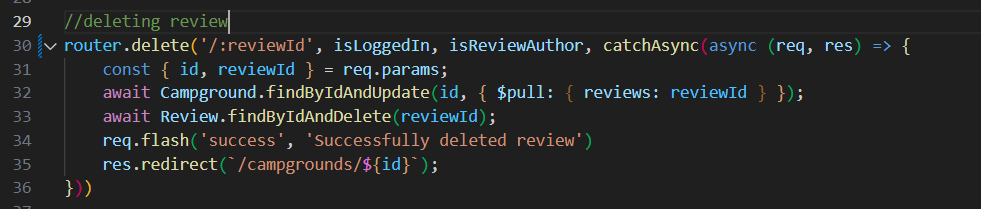
**535. Reviews Permissions and 536. More Reviews Authorisation**

In this video we’ll **protect the routes to ensure that the user who wrote the original review can only edit/delete reviews written on a campground**

This below given middleware is written in **middleware.js**



Then add it in edit/delete route of reviews.js



**Section 51: Yelpcamp: Controllers and star ratings**

**537. Refactoring To Campgrounds Controller and 538. Adding a Reviews Controller**

MVC stands for Model-View-Controller, which is a popular architecture pattern used in software development.

In this **MVC** ­we’re gonna create **controllers** in this lecture.

We’ll create **3 controllers** i.e. **campgrounds, reviews and users.**

**And add all the functionality we had in routes to them**

After creating controllers just use them in routes



**539. A Fancy Way To Restructure Routes**

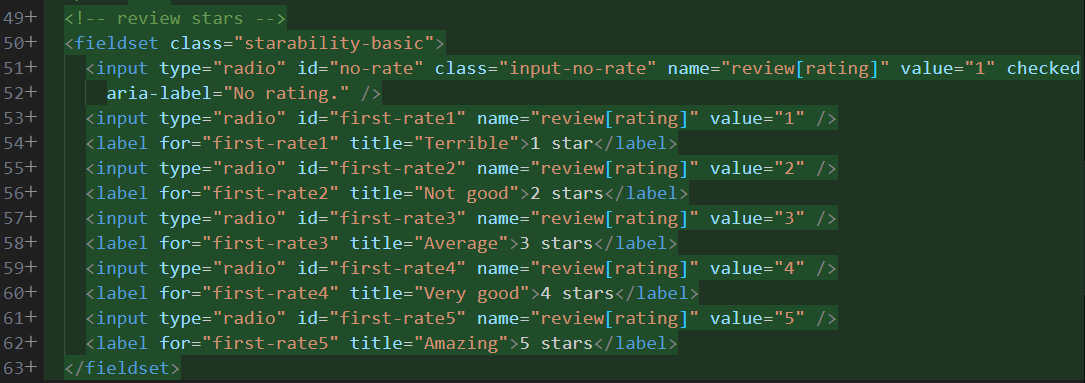
This way we club same kind of routes together and make it more clean.

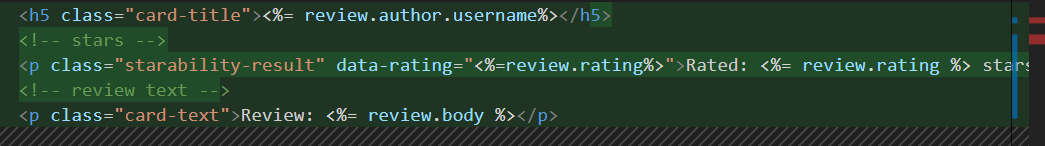
**540. Displaying Star Ratings and 541. Star Rating Form**

In this video we’ll be changing the review slider to stars.

We’ll be using this [css](https://github.com/LunarLogic/starability) for the adding stars functionality.

Then we added this part in show.ejs



And this part to showcase 

**Section 54: Yelpcamp: Image Upload**

**542. Intro To Image Upload Process**

\*We don’t put images in Mongo because images can be very large and there’s lot of BSON document size limit of 16Mb

So we’re gonna use [**cloudinary**](https://cloudinary.com/)where we can store images on cloud.

\*Once our campground form is submitted using the enpoint we’ll send data to cloudinary for storing which will send us URL of those images and we’re gonna store those URL in our mongo database.

**543. The Multer Middleware**

Every form we’ve created are URLencoded form

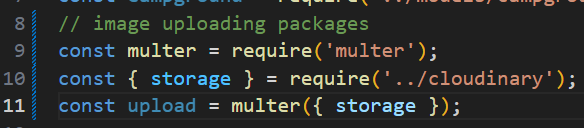
So if we want to submit large files in form we’e to add 

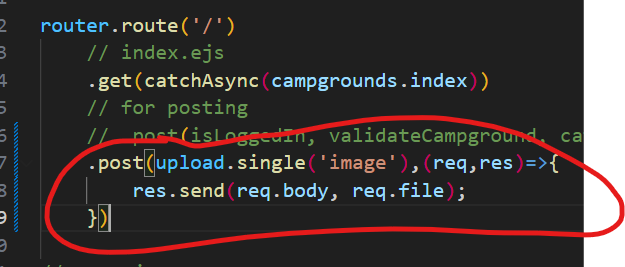
***And in order to parse multipart/forms in order to parse that we need other middleware named Multer.***

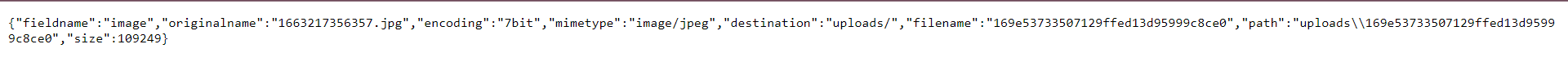
***Multer is a middleware for handling file uploads in Node.js. It allows you to easily handle multipart/form-data, which is commonly used for uploading files, including images. Multer can be used to store images on the server, validate file types, and set file size limits, among other things.***

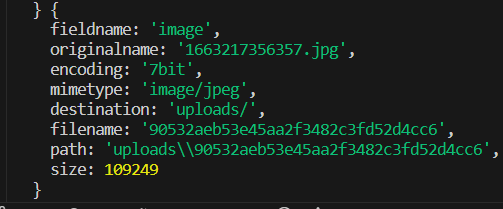
To install use 

In campground.js routes we imprt



This code imports the **Multer middleware** and **the storage configuration** from a file named **'cloudinary.js'**. It **then creates a Multer instance called 'upload'** and **passes the 'storage' configuration object to it.** ****

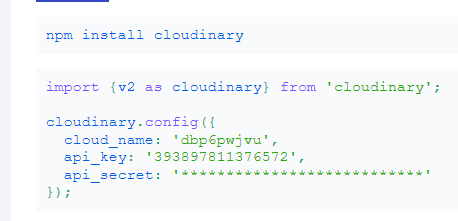




What we want to do is to get them on **cloudinary** or some other service and then we use there stored URL to put them in our database.

**544. Cloudinary Registration**

Login to cloudinary and find your API key and other important info

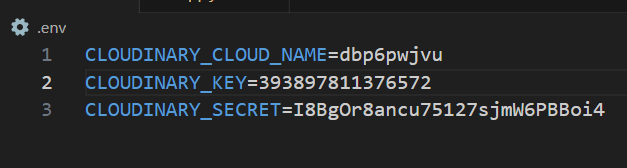


Copy them in your **dotenv file.**

**545. Environment Variables with dotenv**

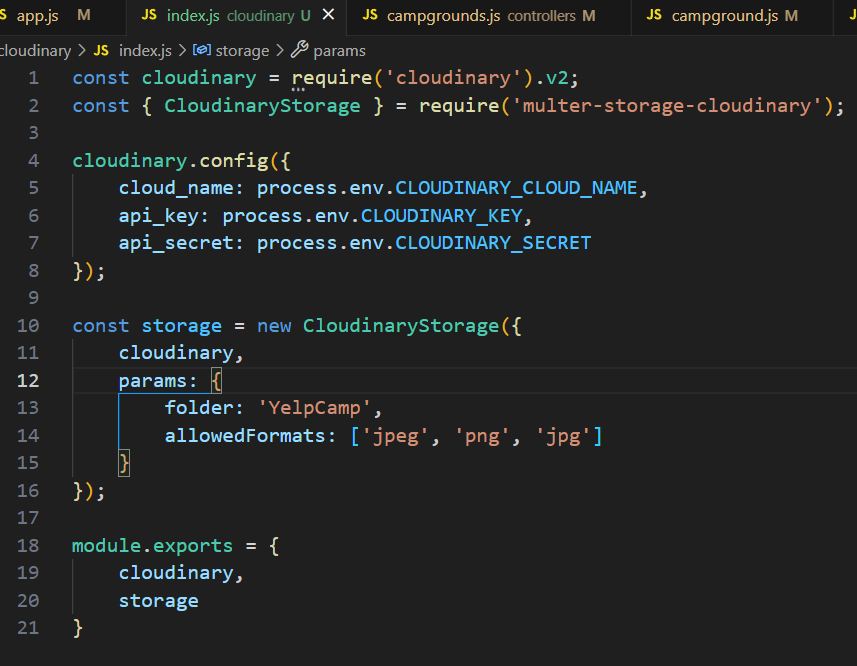
We don’t embed any API key inside usual folders, instead we save them in dotenv file. Which is added to .gitignore

To install 

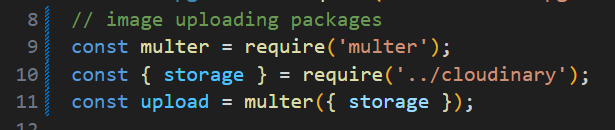
**ADD THESE** ****

**546. Uploading To Cloudinary Basics**

We created cloudinary folder inside which is index.js

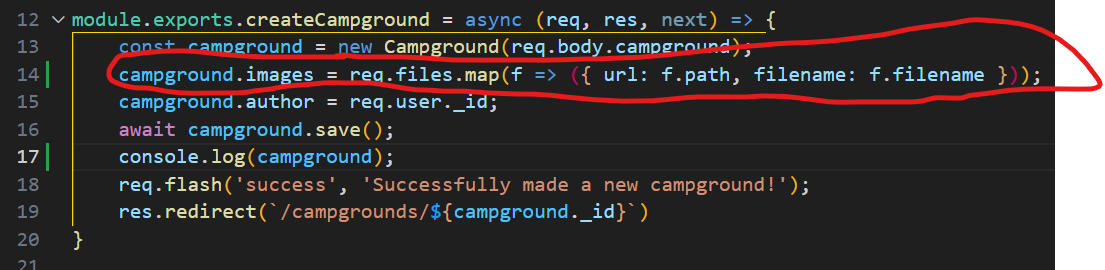


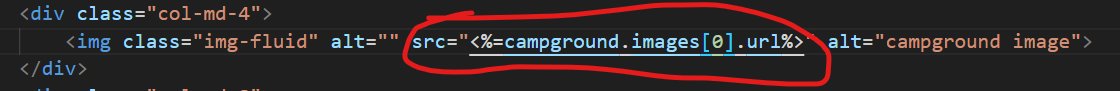
This part allows us to manipulate type of files we can insert in cloudinary 

Then in routes campgrounds.js we specify the cloudinary storage by importing it. 

**547. Storing Uploaded Image Links In Mongo**

Whenever we add a image it sent to cloudinary which returns an object from which we extract out **URL of the image** and **file\_name**  and store It in **mongo.**

****

**And then show the url in show.ejs and new.ejs file **