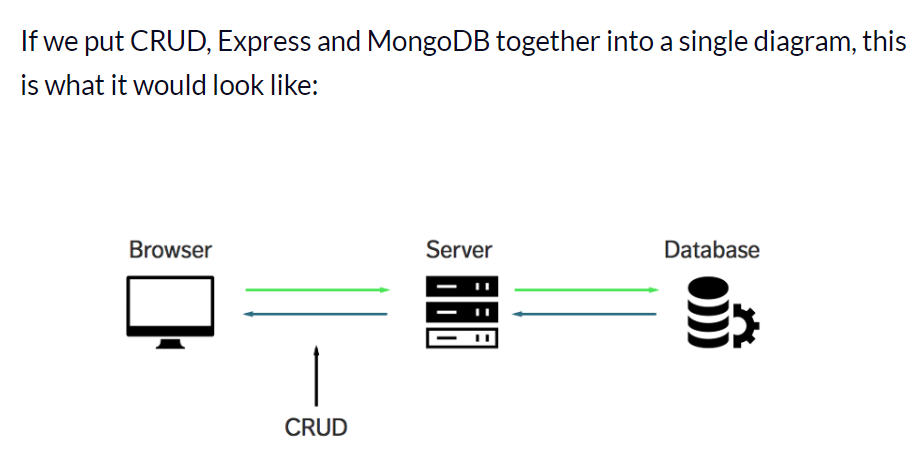
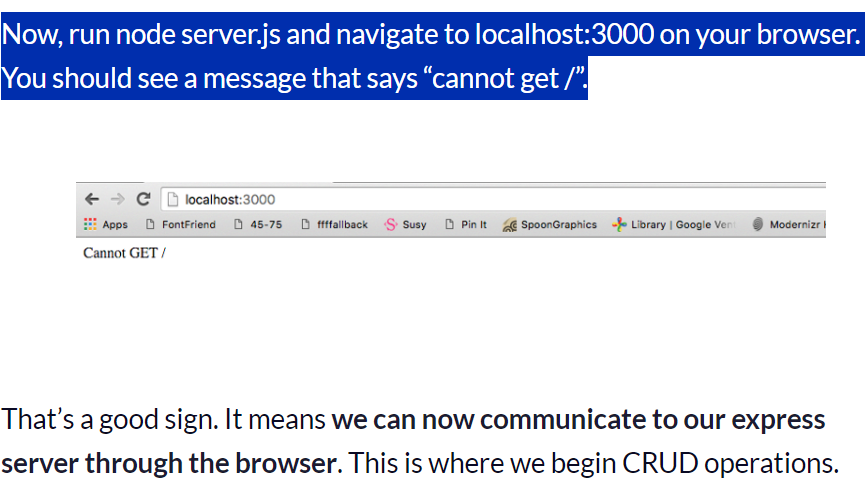
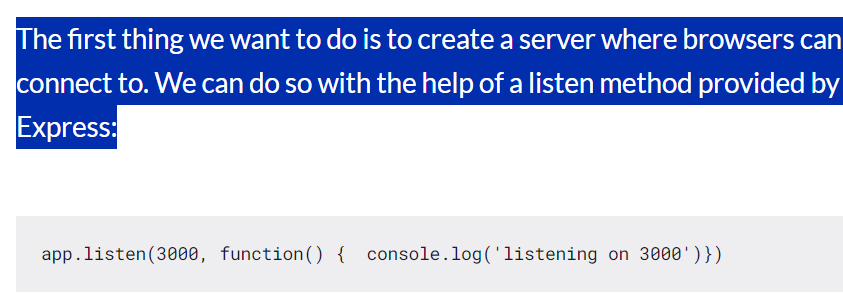
**416 to 425 CRUD operations**



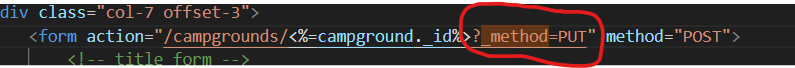
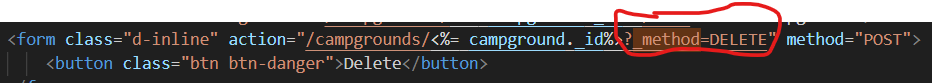


**\*methodOverride** is another middleware for Express that allows you to use HTTP verbs such as PUT or DELETE in HTML forms, which are traditionally limited to GET and POST. So first we import it



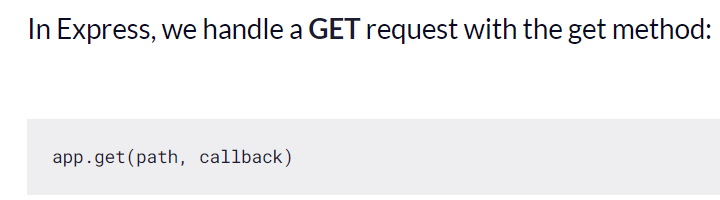
Then we enable it using following in App.js

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******

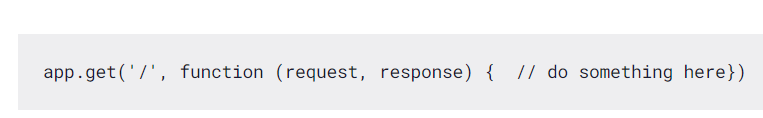
*No we’ve the CRUD operations :*

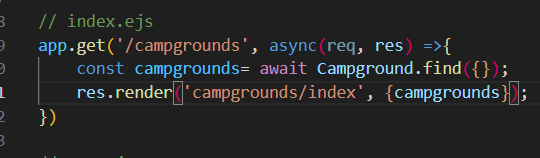
***READ: (get)***

* 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 

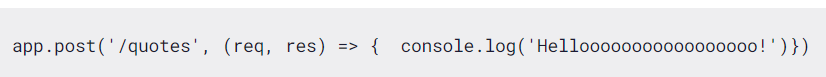
***Create: (post)***

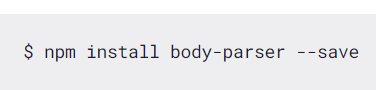
* The **CREATE** operation is performed only by the browser if a **POST** request is sent to the server. This POST request can triggered either with JavaScript or through a **<form> element.**
* **We used form to do it in yelpcamp**
* We need to have three things on this form element:

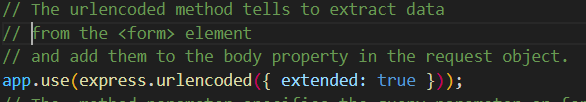
1. An action attribute
2. a method attribute
3. and name attributes on all <input> elements within the form

<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**

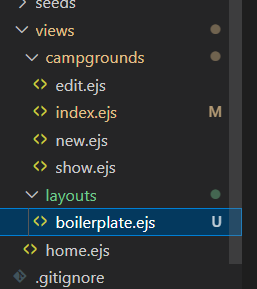
**434. A New EJS Tool For Layouts: EJS mate**

* **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.

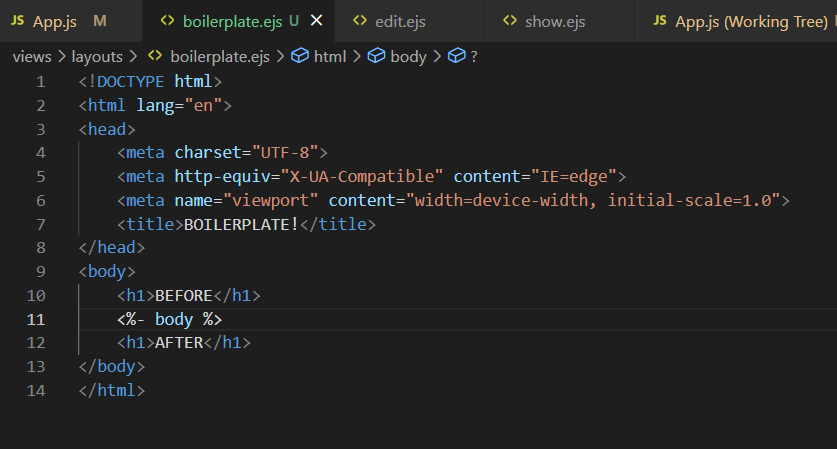
layout(view):

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

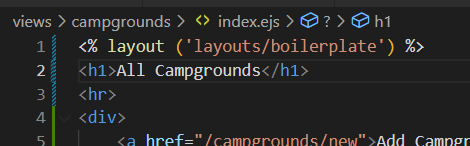
Inside views we created layout folder



Boilerplate.ejs contains boilerplate for ejs

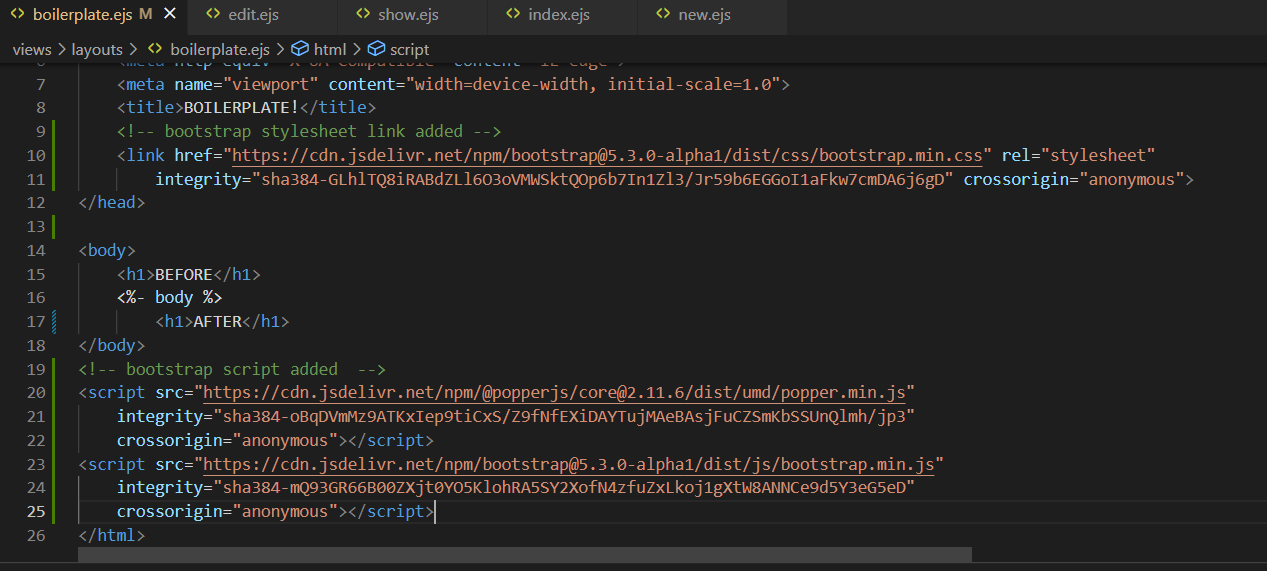


We just need to add ***“ <% layout ('layouts/boilerplate') %>”*** to access the boilerplate

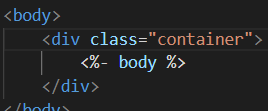


***435. Bootstrap5! Boilerplate***

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



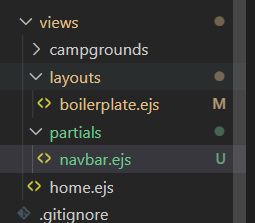
Added container class to the div enclosing the body tag of ejs mate



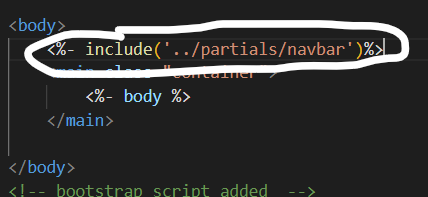
***436. Navbar Partial***

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

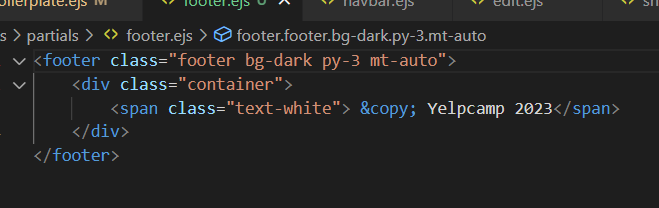
To do so we created a folder named **partials** inside view in which we stored **navbar.ejs**

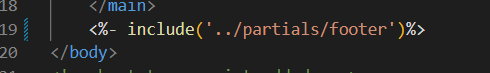


We include navbar partial in boilerplate using following code fragment

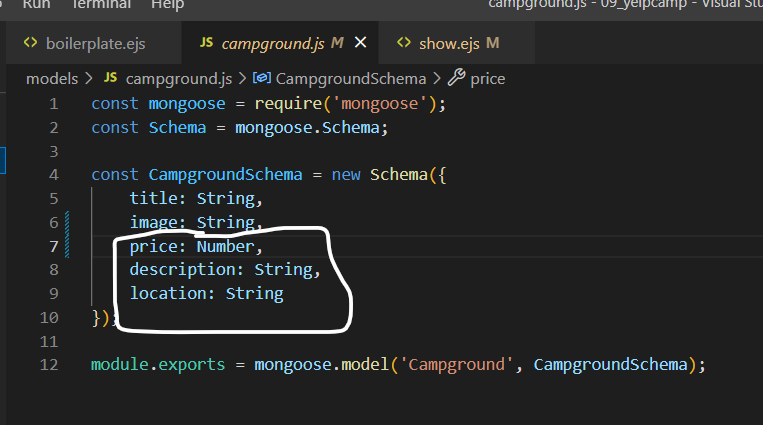


***437. Footer Partial***

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

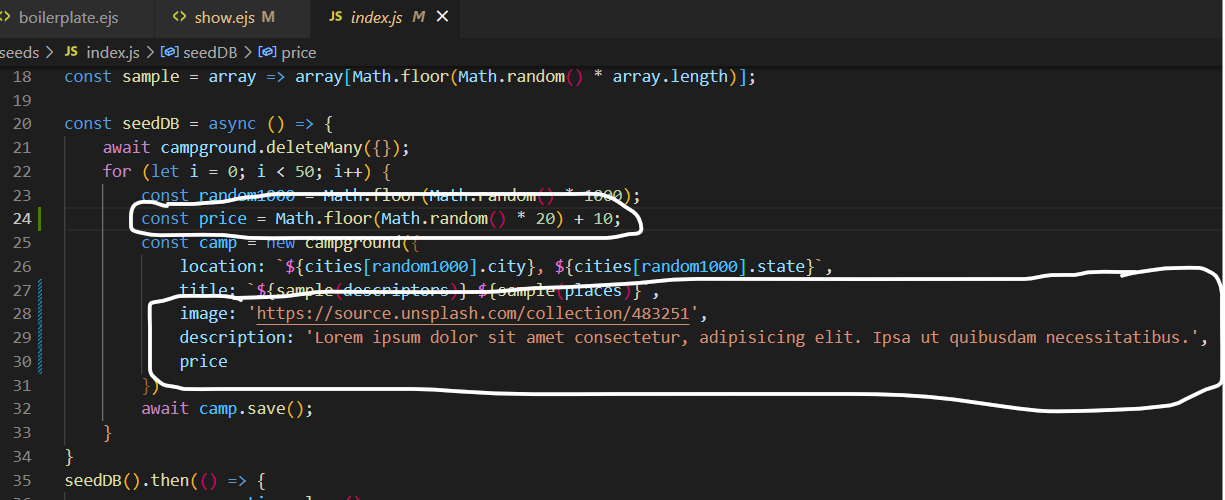
***438. Adding Images***

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

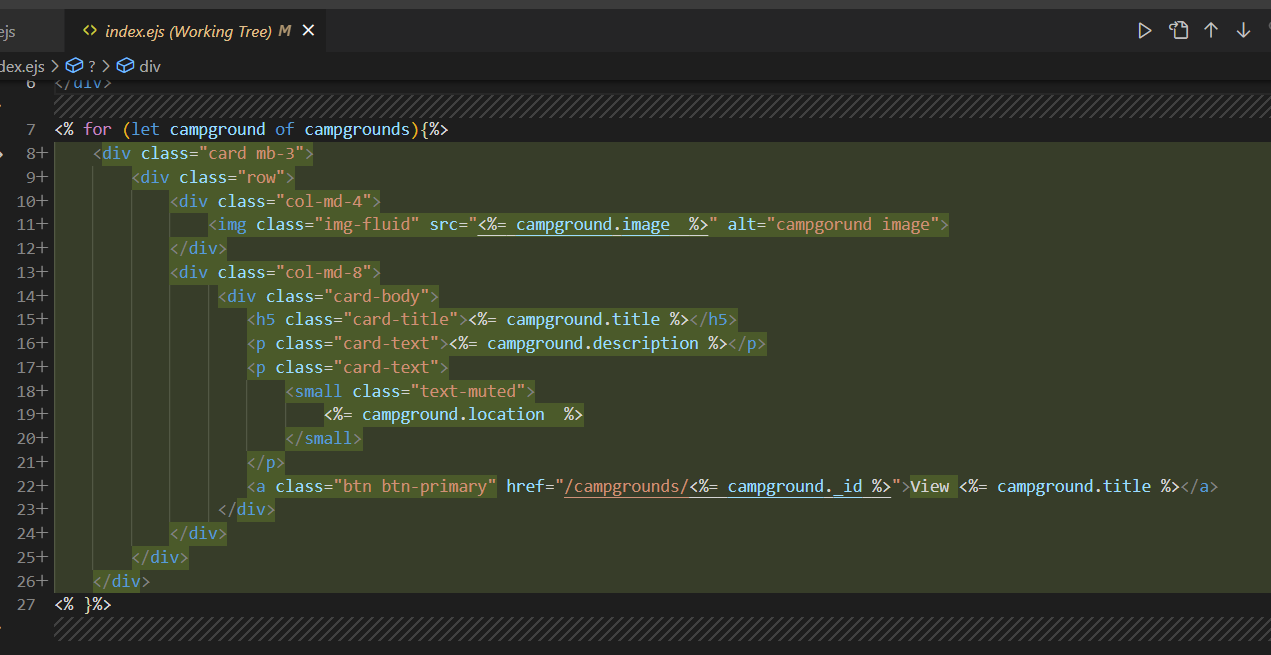


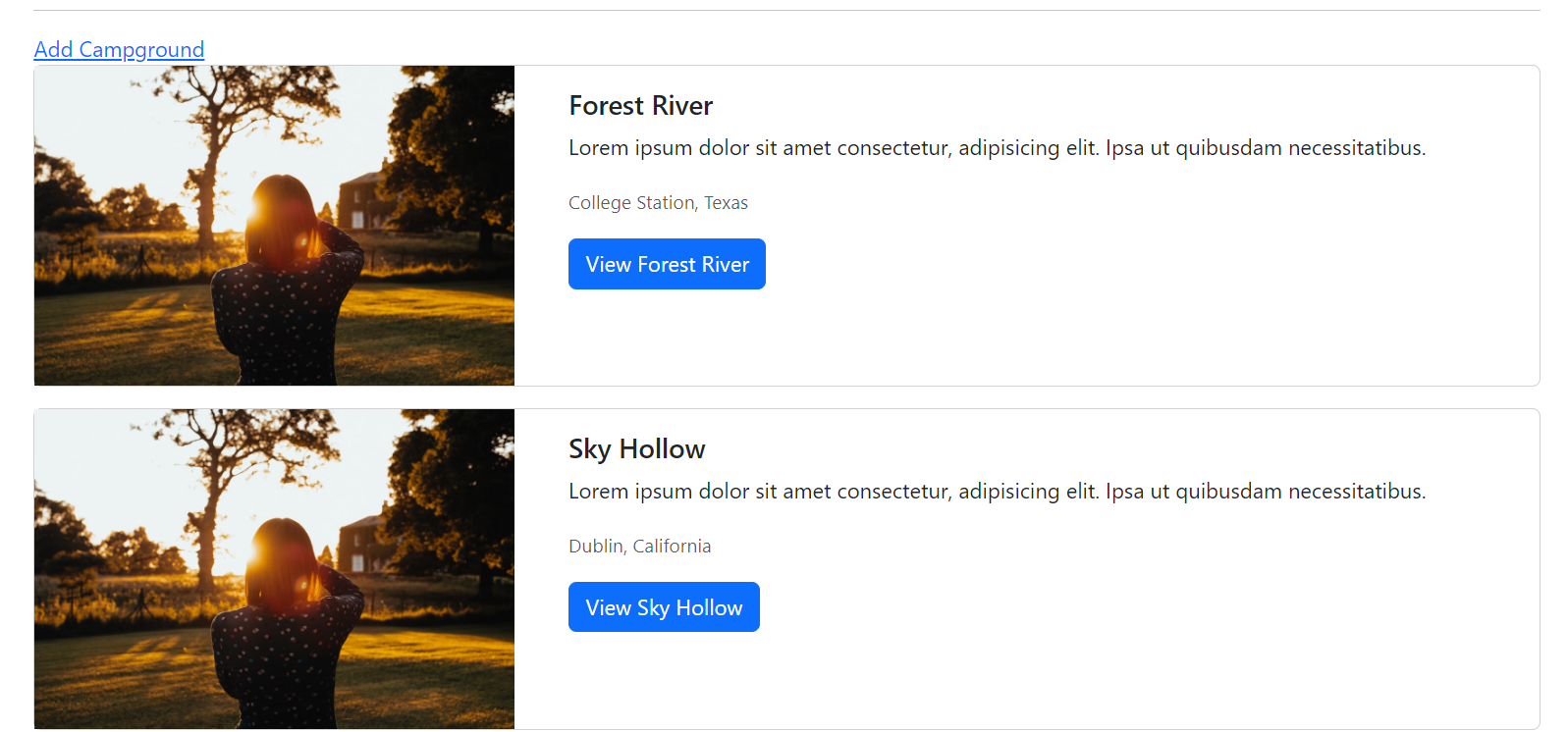
After that we’ve added price, description and image to **views/campgrounds/show.ejs**  so that it is displayed on show page.

***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**

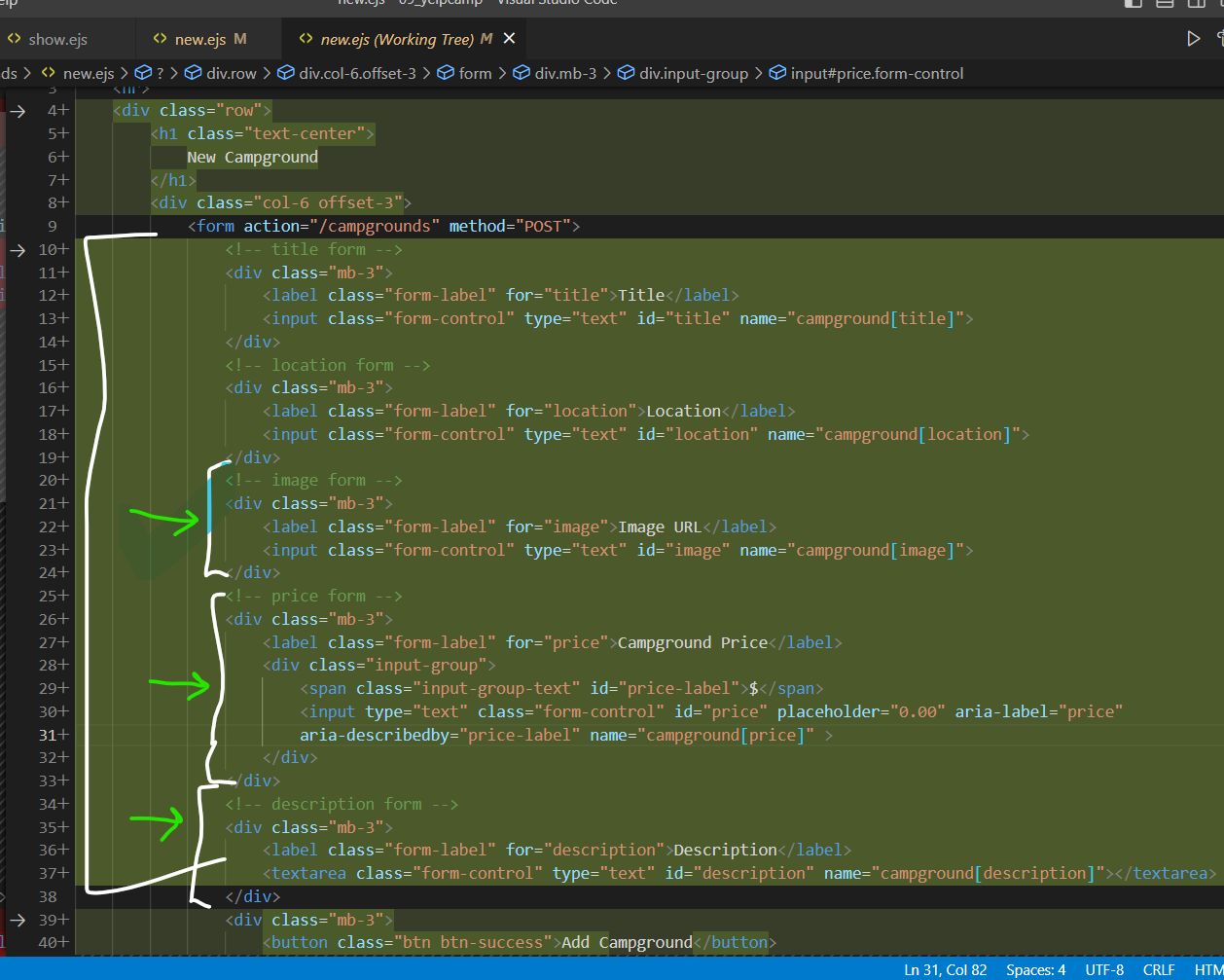
****

****

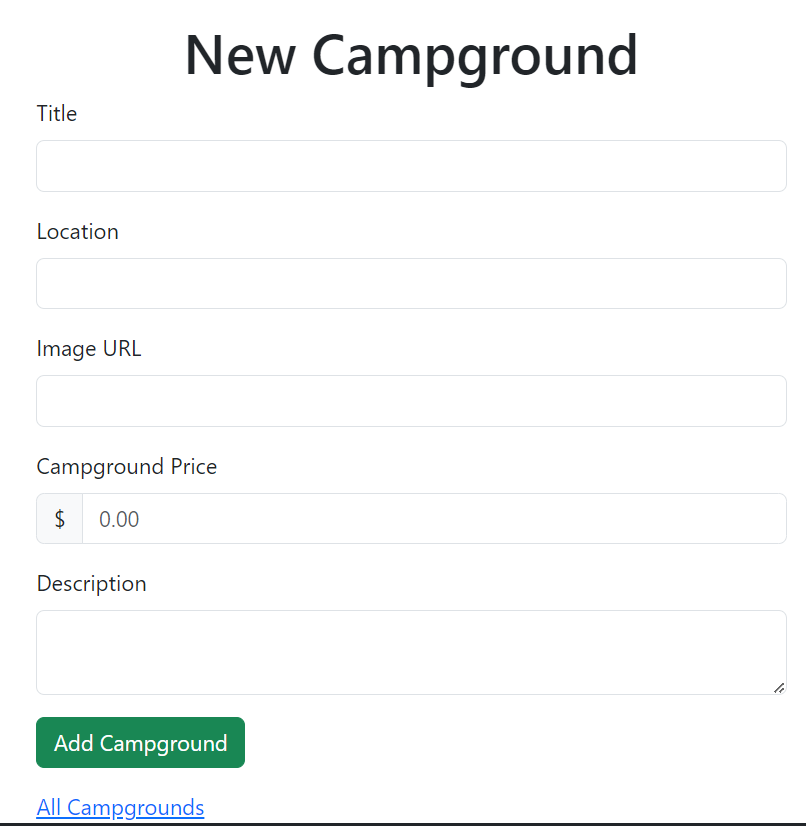
***440. Styling The New Form***

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.

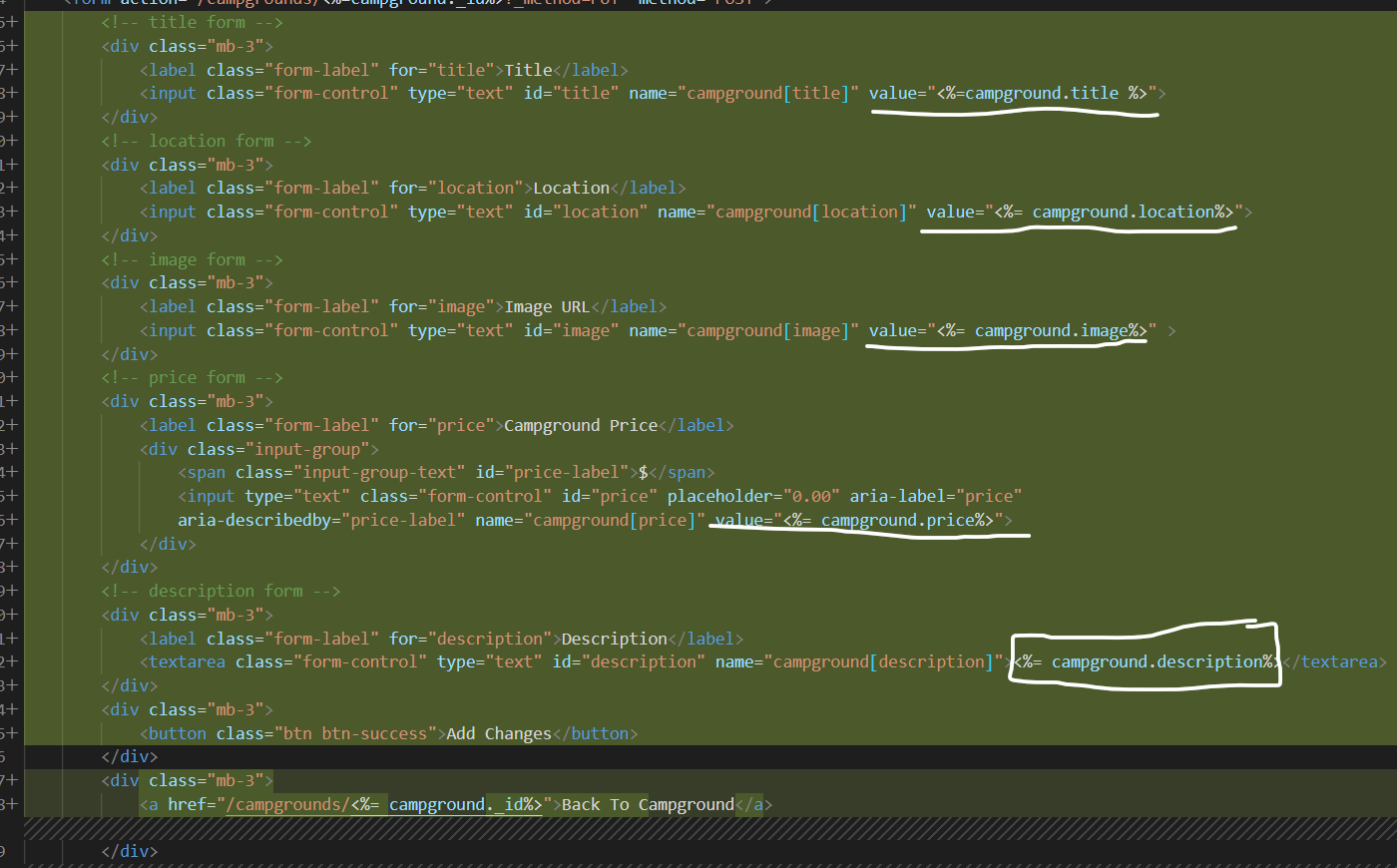


Finally it looks something like this



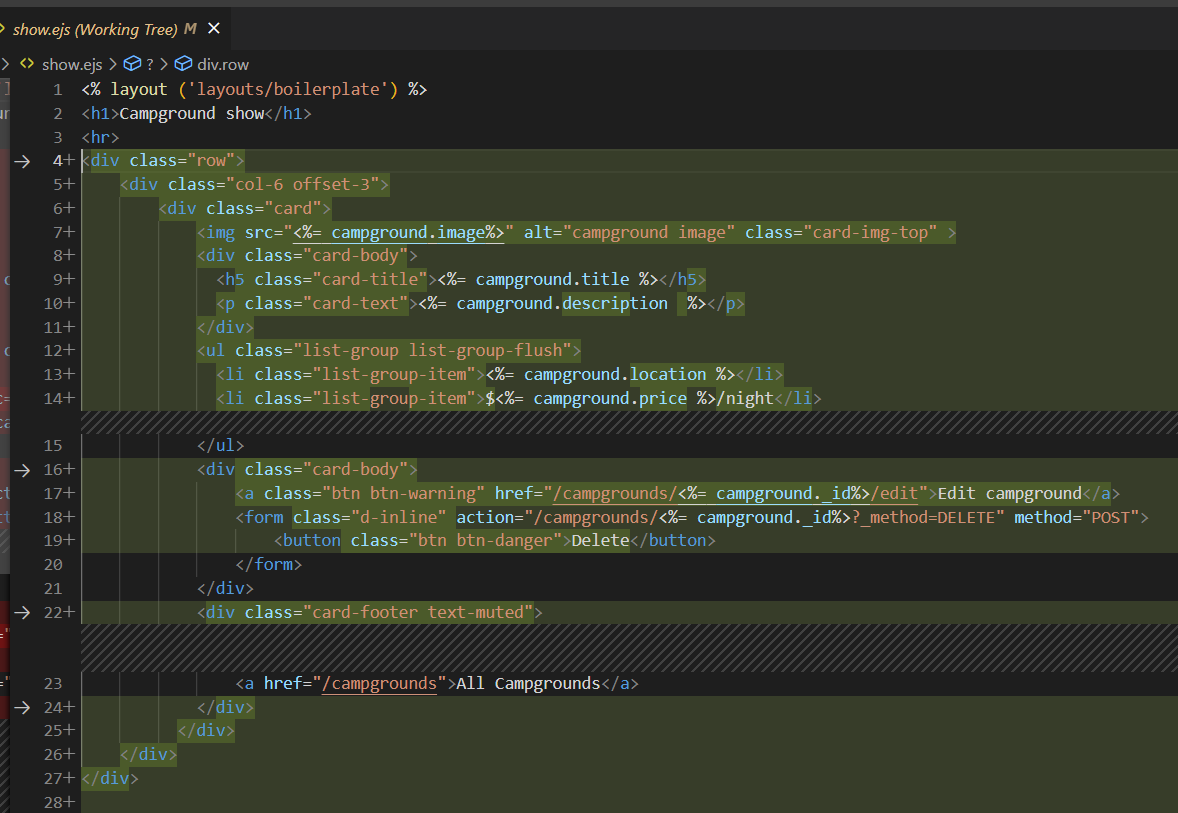
***441. Styling Edit Form***

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. 

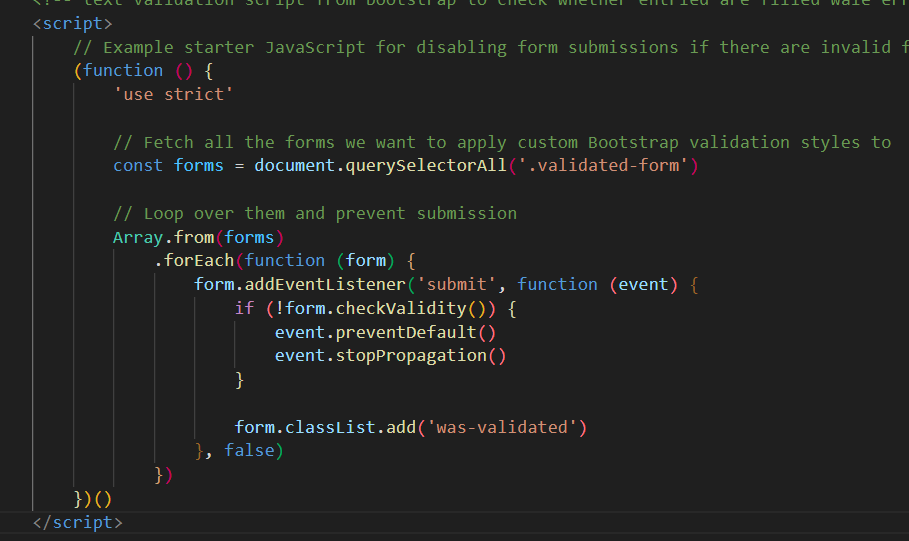
***442. Styling Show Page***

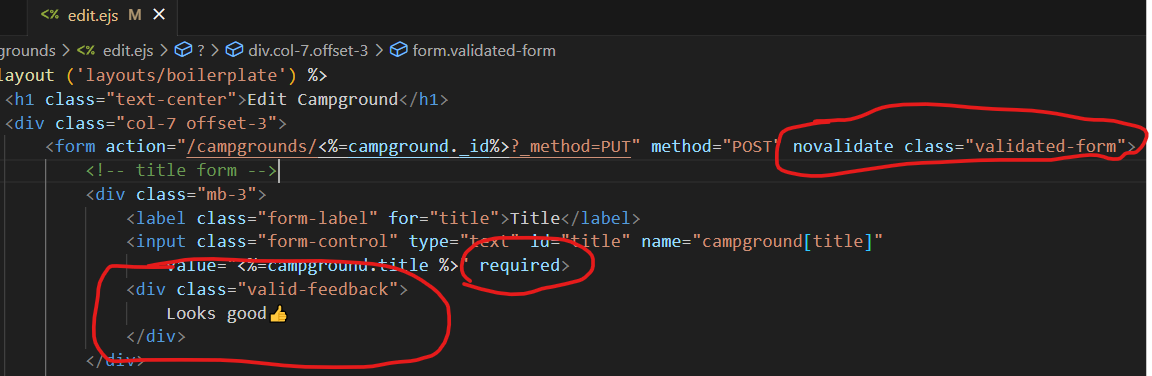
Added better styles using bootstrap to **show.ejs**  file. This was made using card template from bootstrap in which we showcased image the title, description, location and price along with two buttons dedicated to edit and delete the particular campground.

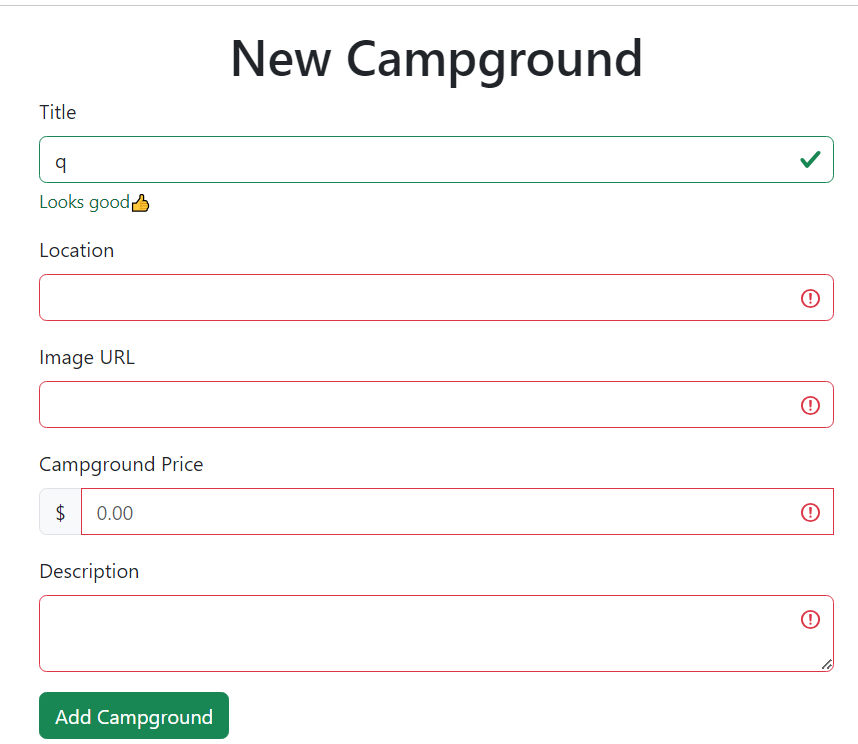


**Section43: Yelpcamp: Errors and Validating data**

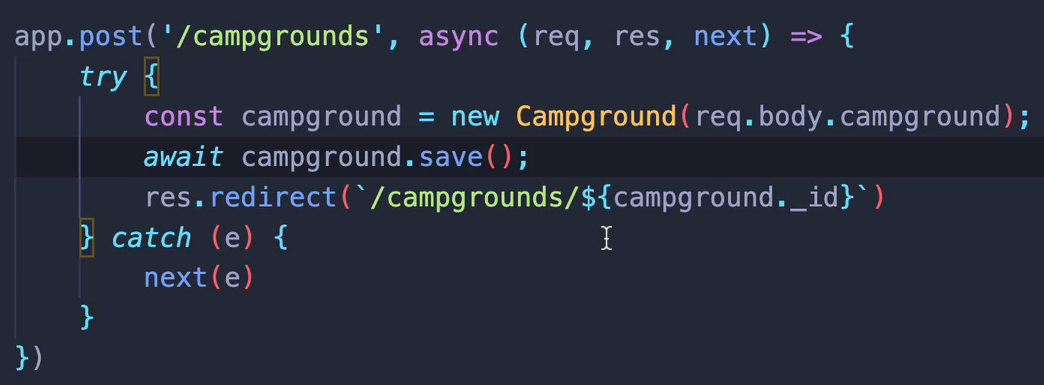
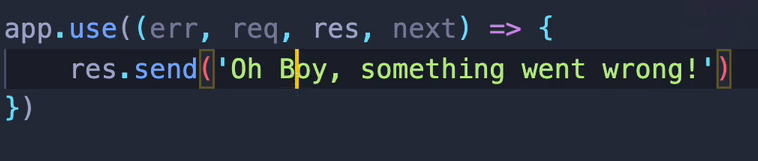
452. Client-Side Form Validations

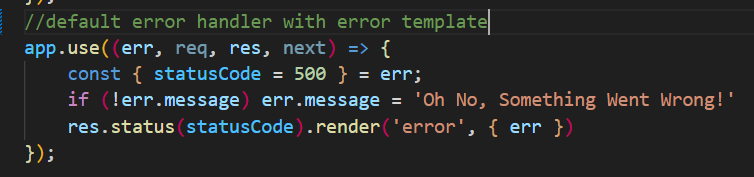
Added this following script to the boilerplate.ejs file which is taken from bootstrap website’s validation page.





453. Basic Error Handler

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

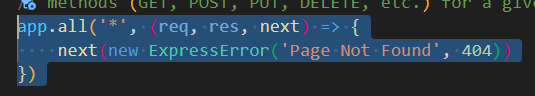
And at last we created a default error handler 

**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.**

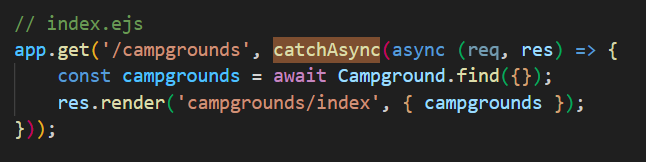
**\*HERE app.all()** method, is **used to handle all HTTP methods (GET, POST, PUT, DELETE, etc.) for a given route**. In this case, the **route is specified as '\*'**, which is a **wildcard that matches all routes**. The middleware function takes **three arguments**: **req (the request object), res (the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

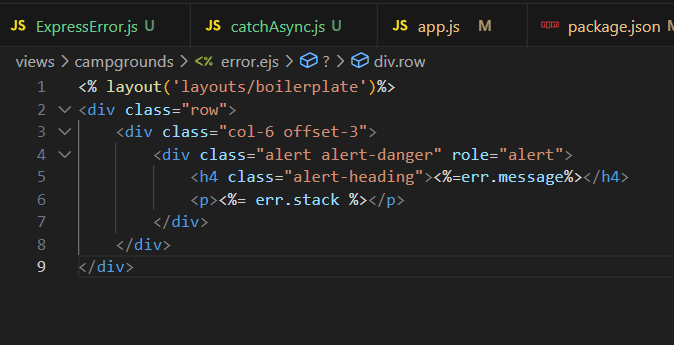
**455. More Errors**

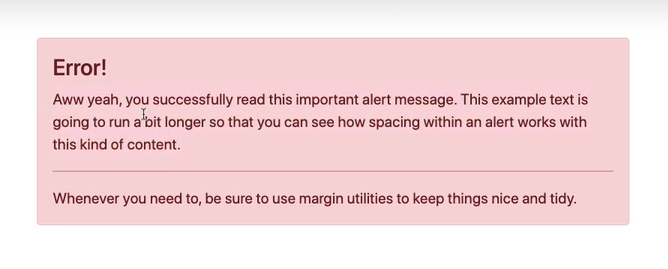
In this part **we created catchAsync** and **ExpressError JS files** inside **utils**

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 