Model-View-Controller - II

Working with Forms

There are many scenarios where forms can be used, Imagine you're running an online store, and you want to give your staff the ability to add new products easily. To do this, you need to create a form that lets them input product information and submit it to the server.

Creating the new-product.ejs view

To add a new product, we need to create a new view file called new-product.ejs. We will use Bootstrap to create the form in this view. Here is the code for the new-product.ejs file:

```html
<!-- views/add-product.ejs -->

<h1 class="mt-5 mb-4">Add New Product</h1>
<form action="/" method="post">
    <div class="mb-3">
        <label for="name" class="form-label">Product Name</label>
        <input type="text" class="form-control" id="name" name="name" required>
    </div>
    <div class="mb-3">
        <label for="desc" class="form-label">Product Description</label>
        <textarea class="form-control" id="desc" name="desc" rows="3" required></textarea>
    </div>
    <div class="mb-3">
        <label for="price" class="form-label">Price</label>
        <input type="number" class="form-control" id="price" name="price" step="0.01" min="0" required>
    </div>
    <div class="mb-3">
        <label for="imageUrl" class="form-label">Image URL</label>
        <input type="url" class="form-control" id="imageUrl" name="imageUrl" required>
    </div>
    <button type="submit" class="btn btn-primary">Add Product</button>
</form>
```

Updating the product.controller.js file

After creating the view, we need to update our product.controller.js file. We will add a **getAddForm** method to get the form, and a **postAddProduct** method to update the model with the form data after submission.

Here is the updated product.controller.js file:

```javascript
import path from 'path'
import ProductModel from '../models/product.model.js'

export default class ProductController {
  getProducts(req, res) {
    let products = ProductModel.getAll()
    console.log(products)
    res.render('products', { products: products })
  }


  getAddForm(req, res) {
    return res.render('new-product')
  }


  postAddProduct(req, res) {
    // access data from form.
    console.log(req.body)
    let products = ProductModel.getAll()
    return res.render('products', { products: products })
  }
}
```
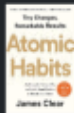
Updating the product.ejs file

Finally, we need to update the product.ejs file to add a "New Product" nav item that redirects to **/new**

Here is how updated layout looks:

Products view

New Product View

Note: req.body value will be undefined we will see how to overcome this problem next.

Parsing Data

Express's internal body parser **express.urlencoded** is used to parse form data. It is a middleware that helps us access form data in the request body.

This middleware is a built-in middleware function in Express. It parses incoming requests with JSON payloads and is based on body-parser. It returns middleware that only parses JSON and only looks at requests where the **Content-Type** header matches the type option. A new body object containing the parsed data is populated on the request object after the middleware (i.e. **req.body**), or an empty object ({}) if there was no body to parse, the Content-Type was not matched, or an error occurred.

To use the express.urlencoded middleware in the project, we need to add the following line of code in our **index.js file**:

```
import express from 'express'
import ProductController from './src/controllers/product.controller.js'
import ejsLayouts from 'express-ejs-layouts'
import path from 'path'


const server = express()

// setup view engine settings
server.set('view engine', 'ejs')
server.set('views', path.join(path.resolve(), 'src', 'views'))

server.use(ejsLayouts)
server.use(express.urlencoded({"extended": false}))

// create an instance of ProductController
const productController = new ProductController()
server.get('/', productController.getProducts)
server.get('/new', productController.getAddForm)
server.post('/', productController.postAddProduct)

server.use(express.static('src/views'))
server.listen(3400)
```

In the **product.controller.js** file, we can use req.body to get the form data that is submitted. To add this data to the products array in **products.model.js**, we need to add an **addProduct** method in the **ProductModel** class.

Here is the updated code for the **products.model.js** file:

```
export default class ProductModel {
  constructor(_id, _name, _desc, _price, _imageUrl) {
    this.id = _id
    this.name = _name
    this.desc = _desc
    this.price = _price
    this.imageUrl = _imageUrl
  }
  static getAll() {
```

```
    return products;
  }

  static addProduct(pObj){
      let newProduct = new ProductModel(products.length+1, pObj.name,
pObj.desc, pObj.price, pObj.imageUrl);
      products.push(newProduct)
  }
}

var products = [
  new ProductModel(
    1,
    'Product 1',
    'Description for Product 1',
    19.99,
'https://m.media-amazon.com/images/I/51-nXsSRfZL._SX328_BO1,204,203,200
_.jpg',
  )
]
```

In the **product.controller.js file**, add a post request on the **/new** endpoint to access the form data and call the addProduct method to add the product to the products array.

Here is the updated code for the product.controller.js file:

```
import path from 'path'
import ProductModel from '../models/product.model.js'

export default class ProductController {
  getProducts(req, res) {
    let products = ProductModel.getAll()
    console.log(products)
    res.render('products', { products: products })
  }


  getAddForm(req, res) {
```

```
    return res.render('new-product')
  }


  postAddProduct(req, res) {
    // access data from form
    ProductModel.addProduct(req.body)
    let products = ProductModel.getAll()
    console.log(products)
    res.render('products', { products: products })
  }
}
```

Validating Data

Data validation is an important process of ensuring that the input data is correct, accurate, and secure. In the context of web development, data validation is crucial to prevent potential security risks and to ensure that the application runs smoothly. Here is some information and code snippets about data validation in the product.controller.js and new-product.ejs files:

Changes in product.controller.js file

To validate the incoming form data, the conditions to check for errors are added in the **postAddProduct** method in product.controller.js file. The updated code adds a new variable called **validURL** which uses the URL constructor to check if the imageUrl provided in the form is a valid URL. If there are any errors, they are pushed into the **errors** array.

If the errors array is not empty, it means that there are errors in the form data, so the **new-product.ejs** view is rendered with the first error message from the errors array using the errorMsg variable. If there are no errors, the **index.ejs** view is rendered with the products variable containing the updated list of products.

Code for postAddProduct method:
```
postAddProduct(req, res) {
    // access data from form.
    ProductModel.addProduct(req.body)
```

```
let products = ProductModel.get()
console.log(products)

let { name, desc, price, imageUrl } = req.body
let errors = []
if (!name || name.trim() == '') {
    errors.push('Name is invalid')
}
if (price < 0) {
    errors.push('Price is invalid')
}
try {
    const validURL = new URL(imageUrl)
} catch (e) {
    errors.push('Invalid URL')
}
if (errors.length != 0) {
    res.render('new-product', { errorMsg: errors[0] })
} else {
    res.render('products', { products: products })
}
}
```
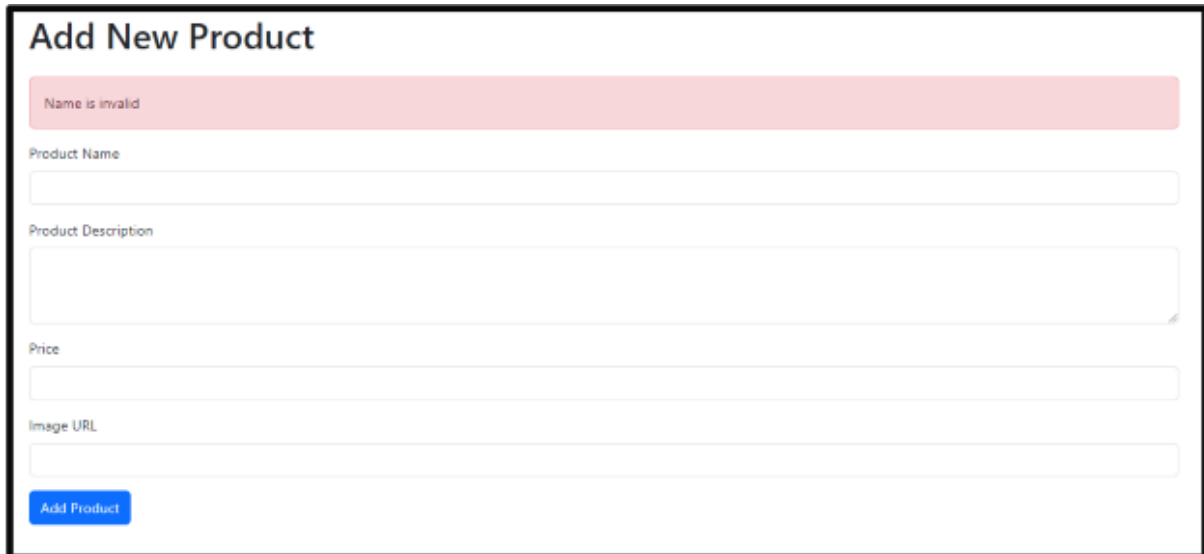
Changes in new-product.ejs file

The new-product.ejs file can incorporate a conditional statement to determine whether to display the the form view with error message or without error message based on the validity of the data. The following is the revised code snippet:

```
<% if(errorMsg){ %>
    <div class="alert alert-danger" role="alert">
      <%= errorMsg %>
    </div>
<%} %>
```

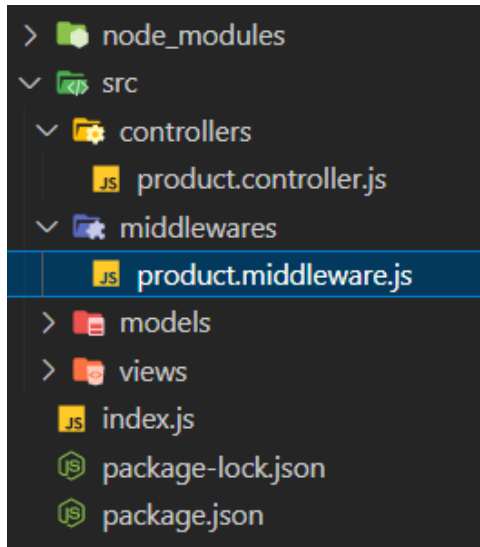Here is what form view with invalid data input looks like:

Validation Middleware

In the current implementation, the validation code for form data is placed in the **postAddProduct** method in **product.controller.js**. This is problematic as it violates the **Single Responsibility Principle**. We can also add the validation code in a separate method and call it form postAddProduct but this lead to the problem of **tight coupling**. To overcome this issue, we can create a separate middleware function for validation.

Here are the steps to implement Validation Middleware:

1. Create a new folder named **middleware** in the src folder.

2. Create a file named **validation.middleware.js** in the middleware folder and add a function named **validateRequest** and export it. Move the validation code from the postAddProduct method to this function.

Here is the validation.middleware.js file:

```js
const validateRequest = (req, res, next) => {
  let {name, price, imageUrl} = req.body
  let errors = []


  if(!name || name.trim() == '')
    errors.push('Name is invalid')


  if(price < 0)
    errors.push('Price is invalid')


  try{
    const validURL = new URL(imageUrl)
  }catch(e){
    errors.push(e)
  }
  if(errors.length != 0)
    res.render('new-product', { errorMsg: errors[0] })
  else
    next()
}
export default validateRequest
```

3. In the **index.js** file, import the **validateRequest** function from validation.middleware.js and pass it as middleware in the **/** route of the **server.post** method.

```javascript
import express from 'express'
import ProductController from
'./src/controllers/product.controller.js'
import ejsLayouts from 'express-ejs-layouts'
import path from 'path'
import validateRequest from
'./src/middleware/validation.middleware.js'

const server = express()

// setup view engine settings
server.set('view engine', 'ejs')
server.set('views', path.join(path.resolve(), 'src', 'views'))

server.use(ejsLayouts)
server.use(express.urlencoded({"extended": false}))

// create an instance of ProductController
const productController = new ProductController()
server.get('/', productController.getProducts)
server.get('/new', productController.getAddForm)
server.post('/', validateRequest,
productController.postAddProduct)

server.use(express.static('src/views'))

server.listen(3400)
```

By implementing the Validation Middleware, we have separated the validation logic from the controller, and achieved a better adherence to the Single Responsibility Principle.

Using Express Valdiator

Express-validator is a middleware used for data validation in Node.js applications built with the Express framework. It is used to validate data coming from HTTP requests, such as form data, query parameters, or JSON payloads.

To use express-validator, we need to follow three steps:

1. Setup rules for validation
2. Run those rules
3. Check if there was any validation error

The code in the **validateRequest** middleware has been updated to use express-validator for data validation.

```
import {body, validationResult} from 'express-validator'

const validateRequest = async (req, res, next) => {
  //Step 1: Setup rules for validation
  const rules = [
   body('name').isLength({min : 1}).withMessage('Name is required'),
   body('imageUrl').isURL().withMessage('Image URL is missing'),
   body('price').isInt({min : 1}).withMessage('Price should be greater
than 1')
  ]
  //Step 2: Run the rules
  await Promise.all(rules.map((rule) => rule.run(req)))
  let validationErrors = validationResult(req)

 console.log(validationErrors)
//Check if there was any validation error
 if(!validationErrors.isEmpty()){
 res.render('new-product', { errorMsg: validationErrors.array()[0].msg
        })
 }else{
   next()
 }
}
export default validateRequest
```

The following methods and imports are used in the code:

- **body**: This method is used to extract the value of a field from the request body for validation.
- **isLength**: This method is used to check the length of a field.
- withMessage: This method is used to specify an error message if the validation fails.
- **isURL**: This method is used to check if a field value is a valid URL.
- **isInt**: This method is used to check if a field value is an integer.
- **validationResult**: This method is used to check if there were any validation errors after running the validation rules.

The updated code uses the **body** method to extract the name, price, and imageUrl fields from the request body. The rules array is then set up using the isLength, isURL, and isInt methods. The **Promise.all** method is used to run all the validation rules, and the validationResult method is used to check if there were any validation errors.

If there are any validation errors, the code renders the new-product view with the first error message. If there are no errors, the code calls the next function to proceed to the next middleware.

Updating Product

To implement the functionality of updating a product's details, the following steps need to be taken:

1. Add an "Update Product" link in the **index.ejs** view to enable updating the product information.

2. In the **ProductController** class of the **product.controller.js** file, add the **getUpdateProductView** method. This method retrieves the product details based on the provided ID and renders the update-product view. If the product is not found, it sends a "Product Not Found" message.

```
getUpdateProductView(req, res){
    const id = req.params.id;
    let productFound = ProductModel.getByID(id);
```

```
    if(productFound){
        res.render('update-product', {
            product: productFound,
            errorMsg: null
        });
    }else{
        res.send("Product Not Found");
    }
}
```

3. Add the **postUpdateProduct** method in the ProductController class. This method receives the updated product details through the request body and sends them to the ProductModel for updating. Finally, it renders the **index.ejs** view to display the updated product list.

```
postUpdateProduct(req, res) {
    ProductModel.update(req.body);
    const products = ProductModel.getAll();
    res.render('products', { products: products});
}
```

4. Create a new file named **update-product.ejs** in the views folder. Use the product object sent from the **getUpdateProductView** method to autofill the update-product view with the existing product details.

Update Product

Product Name

Product 1

Product Description

Description for Product 1

Price

19.99

Image URL

https://m.media-amazon.com/images/I/51-nXsSRfZL._SX328_BO1,204,203,200_.jpg

Add Product

5. In the index.js file, make the following changes to the routing:

```
server.post('/update-product',
  productController.postUpdateProduct);
server.get('/update-product/:id',
  productController.getUpdateProductView);
```

`server.post('/update-product', productController.postUpdateProduct)`: This route is responsible for handling the POST request to update the product. It calls the postUpdateProduct method of the ProductController when this route is accessed.

`server.get('/update-product/:id', productController.getUpdateProductView)`: This route handles the GET request to retrieve the update-product view. It includes a route parameter :id, representing the ID of the product to be updated. It calls the getUpdateProductView method of the ProductController to render the update-product view.

6. In the products.model.js file, add the following changes to the ProductModel class:

```
static update(productObj) {
```

```
    const index = products.findIndex((p) => p.id ==
productObj.id);
    products[index] = productObj;
}


static getByID(id) {
    return products.find((p) => p.id == id);
}
```

getByID(id): This method is added to retrieve a product by its ID. It takes the id parameter, representing the ID of the product to be retrieved. The method performs the following steps:
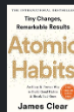- It uses the find function on the products array to find the product that matches the provided id.
- If a matching product is found, it is returned by the method.

update(productObj): This method is added to update the product details. It takes the productObj parameter, which represents the updated product object containing the new details. The method performs the following steps:
- It uses the findIndex function on the products array to locate the index of the product that matches the provided id.
- Once the index is found, the method replaces the existing product at that index with the updated productObj, effectively updating the details of the product in the array.

Deleting Product

To add the delete product feature, a delete link needs to be added to each product in the index.ejs file. This link should redirect to the URL "/delete-product/" followed by the product's ID.

ID	Name	Description	Price	Image	Operation
1	Product 1	Description for Product 1	19.99	Atomic Habits James Clear	Update Delete
2	Product 2	Description for Product 2	29.99	IKIGAI	Update Delete

1. In the product.controller.js file, the following changes are made to add the deleteProduct() middleware:
```
deleteProduct(req, res) {
    const id = req.params.id;
```

```
    ProductModel.delete(id);
    var products = ProductModel.getAll();
    res.render('products', { products });
}
```

- **deleteProduct(req, res)**: This middleware is responsible for handling the deletion of a product. It extracts the id parameter from the request's URL parameters using **req.params.id**, representing the ID of the product to be deleted.
- It calls the **delete()** method of **ProductModel**, passing the id parameter to delete the product from the model.
- After deleting the product, it retrieves the updated list of products using the getAll() method of ProductModel.
- Finally, it renders the products view, passing the updated list of products to be displayed.

2. In the product.model.js file, the following changes are made to the ProductModel class:

```
static delete(id) {
    const index = products.findIndex(p => p.id == id);
    products.splice(index, 1);
}
```

- **static delete(id):** This static method is added to delete a product from the model based on its ID. It takes the id as a parameter.
- Inside the method, it uses the findIndex() method on the products array to locate the index of the product with the matching ID (id).
- If a matching product is found (index is not -1), it uses the splice() method to remove the product from the products array at the identified index.

Confirmation before deletion

To add a confirmation message before deleting a product, the anchor tag used for deletion in the index.ejs file is replaced with a delete button. Modify the code in index.ejs as follows:

```
<button class="btn btn-danger" onClick="deleteProduct('<%= product.id %>')">Delete</button>
<!-- <a href="/delete-product/<%= product.id %>" class="btn btn-danger">Delete</a> -->
```

In the main.js file, add the following code for the **deleteProduct** method:
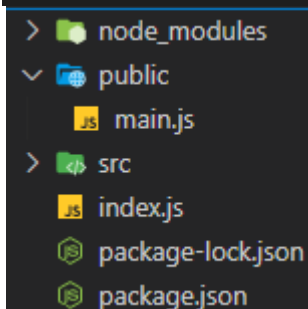
```
function deleteProduct(id) {
    const result = confirm('Are you sure you want to delete this
product?');

    if (result) {
        fetch('/delete-product/' + id, {
            method: 'POST'
        }).then((res) => {
            if (res.ok) {
                location.reload();
            }
        });
    }
}
```

Create a public folder in the root directory of the project, which will contain a main.js file. This file will handle the frontend JavaScript code for the delete functionality.

```
server.use(express.static('public'));
```

```
> node_modules
∨ public
    main.js
> src
    index.js
    package-lock.json
    package.json
```
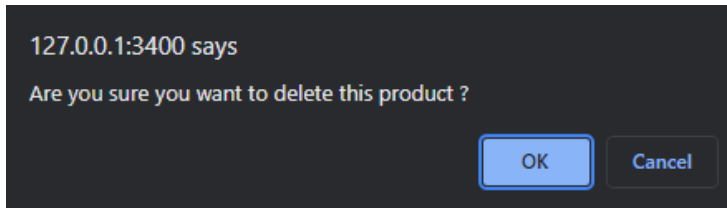
Add the main.js file in script tag in layout.ejs view.

```
<script src="main.js"></script>
```

Lastly, update the route in index.js to handle the POST request for deleting a product. Add the following line of code:

```
server.post("/delete-product/:id", productController.deleteProduct);
```

This route maps to the **deleteProduct** method in the **productController**, which handles the deletion of the product.
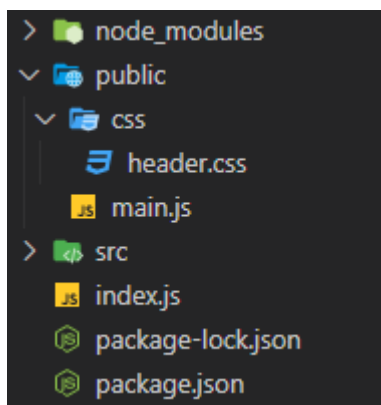
By implementing these changes, a confirmation message will be displayed before deleting a product, and the deletion will be handled by the deleteProduct method in the productController.

Styling Views

By following these steps, you can create a separate CSS file, link it in the layout.ejs file, and apply specific styles to the header elements in the new-product.ejs, update-product.ejs, and index.ejs views.

1. Create a css folder inside the public folder. This folder will hold your CSS files.
2. Inside the css folder, create a header.css file that will contain the CSS styles for the header.

3. In the layout.ejs file, add the following line inside the <head> tag to link the header.css file:

```
<link rel="stylesheet" href="/css/header.css">
```

4. Apply the header class to the new-product.ejs, update-product.ejs, and index.ejs files. In these files, you can add the header class to the appropriate elements that represent the header of each view. For example:

```
<div class="header">
    <!-- Header content -->
</div>
```

By applying the header class, the corresponding styles defined in header.css will be applied to the header elements in these views.

Summarising it

Let's summarise what we have learned in this module:

- **Learned to create and submit forms using POST requests.**
- **Added a new feature to add products and implemented manual data validation.**
- **Set up validation middleware and used Express Validator for data validation.**
- **Added features to update and delete products.**
- **Explored styling views in an Express application.**

Some Additional Resources:

- [**What does express.urlencoded do anyway?**](#)

- [**Form Data Validation in Node.js with express-validator**](#)
- [**Express Validator Tutorial**](#)
- [**Express + EJS: Styles and Partials**](#)