

File Upload, Session, and Cookies

File Uploading

To enable the feature of uploading images instead of using image URLs in the inventory management app, the following steps should be followed:

Install Multer

- Multer is a Node.js middleware used for handling multipart/form-data, primarily for file uploads.
- Install Multer by running the command: `npm i multer`

Changes in the new-product view and update-product view

- Replace the input field of type "text" for the imageUrl with an input field of type "file" to allow image uploads:

```
<input type="file" name="imageUrl" id="imageUrl" accept="images/*" />
```

- Set the form's enctype attribute to "multipart/form-data" since we are now uploading files along with other form data:

```
<form action="/" method="post" enctype="multipart/form-data">  
  . . .  
</form>
```

- Add new-product view:

Add New Product

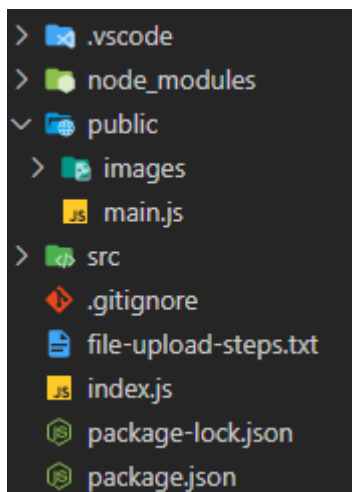
Product Name

Product Description

Price

Image URL No file chosen

Create an "images" folder in the public folder to store the uploaded images.



Implement a file-upload middleware to handle the rules and processing of uploaded images

- Create a file named "update-product.middleware.js" in the middleware folder. Use Multer to configure the storage and file naming:

```
import multer from 'multer';

const storageConfig = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, 'public/images/');
  },
  filename: (req, file, cb) => {
    const name = Date.now() + '-' + file.originalname;
```

```

        cb(null, name);
      },
    });

export const uploadFile = multer({
  storage: storageConfig,
});

```

Apply the file-upload middleware in the index.js file.

- Modify the code in index.js to include the file-upload middleware in the POST route for adding a product:

```

app.post('/', uploadFile.single('imageUrl'), validationMiddleware,
productsController.postAddProduct);

```

Update the product.controller.js file to handle the uploaded image.

- In the postAddProduct method, extract the name, desc, and price from the request body, and get the filename of the uploaded image.
- Modify the imageUrl to use the "images" folder and the filename of the uploaded image:

```

const imageUrl = 'images/' + req.file.filename;

```

- Update the ProductModel's add method to include the imageUrl parameter:

```

ProductModel.add(name, desc, price, imageUrl);

```

Adjust the imageUrl validation in validation.middleware.js to account for the changes in image handling:

Add a custom validation rule to check if the req.file object exists. If not, throw an error indicating that the image is required:

```

body('imageUrl').custom((value, { req }) => {
  if (!req.file) {
    throw new Error('Image is required');
  }
  return true;
});

```

Understanding Session

Session



- Session is a mechanism that maintains stateful communication between a client and a server.
- In a stateless protocol like HTTP, sessions help store and maintain information about a client.
- Sessions are used to manage authentication status, preferences, and other client-specific data.
- A unique session ID is generated by the server and sent to the client as a cookie.
- The client includes the session ID with each request, allowing the server to identify and access the client's session data.
- Sessions provide secure stateful communication by storing session data on the server, making it difficult for attackers to tamper with.
- Sessions play a crucial role in securing applications by allowing the server to maintain authentication status and other important user data.
- Sessions help prevent unauthorized access and protect sensitive information.
- Registration and login features can be implemented to create sessions for users.
- Sessions enhance application security and enable personalized user experiences.

Registration Page

To implement the user registration page we have to make controller, model and view for user.

1. Create a user.model.js file in the models folder to define the UserModel class with id, name, email, and password properties.

```
export default class UserModel {
  constructor(id, name, email, password) {
    this.id = id;
    this.name = name;
    this.email = email;
    this.password = password;
  }
}
```

2. Create a register.ejs view in the views folder with a form for user registration.

```
<h1 class="mt-5 mb-4">Register</h1>

<form action="/register" method="post">
  <div class="mb-3">
    <label for="name" class="form-label">Name</label>
    <input type="text" class="form-control" id="name" name="name"
required>
  </div>
  <div class="mb-3">
    <label for="email" class="form-label">Email address</label>
    <input type="email" class="form-control" id="email"
name="email" required>
  </div>
  <div class="mb-3">
    <label for="password" class="form-label">Password</label>
    <input type="password" class="form-control" id="password"
name="password" required>
  </div>
  <button type="submit" class="btn btn-primary">Register</button>
</form>
```

3. Create a user.controller.js file in the controllers folder with a UserController class that handles the get request for the registration page.

```
export default class UserController {
  getRegister(req, res) {
    res.render('register');
  }
}
```

```
}  
}
```

4. In index.js, instantiate a UserController object and add a get request for /register that renders the register.ejs view.

```
const usersController = new UserController();  
app.get('/register', usersController.getRegister);
```

Login Page

To implement the user login page we have to make controller, model and view for user.

1. Create a login.ejs view in the views folder with a form for user login and an optional error message display.

```
<h1 class="mt-5 mb-4">Login</h1>  
  
<%if(errorMessage){ %>  
  <div class="alert alert-danger" role="alert">  
    <%= errorMessage %>  
  </div>  
<%}%>  
  
<form action="/login" method="post">  
  
  <div class="mb-3">  
    <label for="email" class="form-label">Email address</label>  
    <input type="email" class="form-control" id="email"  
name="email" required>  
  </div>  
  <div class="mb-3">  
    <label for="password" class="form-label">Password</label>  
    <input type="password" class="form-control" id="password"  
name="password" required>  
  </div>  
  <button type="submit" class="btn btn-primary">Login</button>  
</form>
```

2. Update the UserModel class in user.model.js to include static methods for adding a user and validating user credentials. Use an array (users) to store the user data.

```
export default class UserModel {
  constructor(id, name, email, password) {
    this.id = id;
    this.name = name;
    this.email = email;
    this.password = password;
  }

  static add(name, email, password) {
    const newUser = new UserModel(
      users.length + 1,
      name,
      email,
      password
    );
    users.push(newUser);
  }

  static isValidUser(email, password) {
    const result = users.find(
      (u) =>
        u.email == email && u.password == password
    );
    return result;
  }
}

var users = [];
```

3. In user.controller.js, add the necessary methods for handling user registration and login:

```
import UserModel from '../models/user.model.js';
import ProductModel from '../models/product.model.js';

export default class UserController {
  getRegister(req, res) {
```

```

    res.render('register');
  }

  getLogin(req, res) {
    res.render('login', { errorMessage: null });
  }

  postRegister(req, res) {
    const { name, email, password } = req.body;
    UserModel.add(name, email, password);
    res.render('login', { errorMessage: null });
  }

  postLogin(req, res) {
    const { email, password } = req.body;
    const user = UserModel.isValidUser(
      email,
      password
    );
    if (!user) {
      return res.render('login', {
        errorMessage: 'Invalid Credentials',
      });
    }
    var products = ProductModel.getAll();
    res.render('index', { products });
  }
}

```

4. In index.js, add a get request for /login that renders the login.ejs view.

```
app.get('/login', usersController.getLogin);
```

5. In index.js, handle post requests for /register and /login by calling the corresponding methods in the UserController:

```

app.post('/login', usersController.postLogin);
app.post(
  '/register',
  usersController.postRegister

```



```
);
```

Securing Application

When a user visits your application, a unique session ID is generated and stored in a cookie on the user's browser. This session ID is sent with every request the user makes to your application, allowing the server to identify the user and access their session data.

To implement the session in our project we need to follow these steps:

1. To implement session authentication, we need to install and use the express-session package by running `npm i express-session`

2. Import express-session:

```
import session from 'express-session';
```

3. In the index.js file, configure the session using the app.use() method and provide the necessary options.

```
app.use(  
  session({  
    secret: 'SecretKey',  
    resave: false,  
    saveUninitialized: true,  
    cookie: { secure: false },  
  })  
);
```

Here is what each option means:

- **secret**: A string used to sign the session ID cookie to prevent tampering.
 - **resave**: Specifies whether the session should be saved back to the session store on each request.
 - **saveUninitialized**: Determines if an uninitialized session should be saved to the session store.
 - **cookie**: Configures the session cookie options.
4. In the postLogin method of the UserController in user.controller.js, after validating the user's credentials, store the user's email in the session.

```
postLogin(req, res) {  
  const { email, password } = req.body;  
  const user = UserModel.isValidUser(  
    email,  
    password,  
    req.session
```

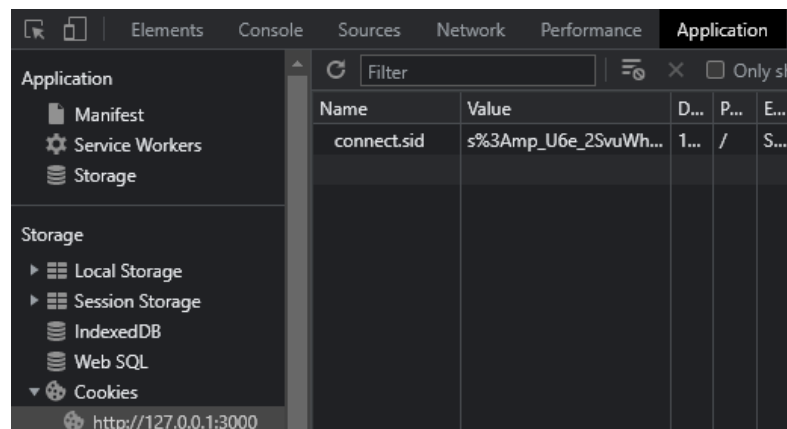
```

    email,
    password
  );
  if (!user) {
    return res.render('login', {
      errorMessage: 'Invalid Credentials',
    });
  }
  req.session.userEmail = email;
  var products = ProductModel.getAll();
  res.render('index', { products });
}

```

5. To verify that the session ID is stored in a cookie, follow these steps:

- Start the project server using `node index.js`
- Register with an email and password, and then log in.
- Inspect the browser and go to the "Application" tab.
- Under the "Cookies" section, you will find the sid (session ID) cookie and its value.



6. To restrict access to certain routes only to logged-in users, create an `auth.middleware.js` file in the `middlewares` folder.

```

export const auth = (req, res, next) => {
  if (req.session.userEmail) {
    next();
  } else {
    res.redirect('/login');
  }
};

```

7. In index.js, apply the auth middleware to the routes that require authentication.

```
app.get(
  '/',
  auth,
  productsController.getProducts
);

app.get(
  '/add-product',
  auth,
  productsController.getAddProduct
);

app.get(
  '/update-product/:id',
  auth,
  productsController.getUpdateProductView
);

app.post(
  '/delete-product/:id',
  auth,
  productsController.deleteProduct
);

app.post(
  '/',
  auth,
  uploadFile.single('imageUrl'),
  validationMiddleware,
  productsController.postAddProduct
);

app.post(
  '/update-product',
  auth,
  productsController.postUpdateProduct
);
```

8. The auth middleware ensures that only logged-in users can access these routes. If the user is not logged in, they will be redirected to the login page.

Logout and Clearing Session

When a user is logged in, we need to remove the login and register links from the layout and add a logout link.

1. In the layout.ejs file, modify the navigation section to conditionally render the links based on the presence of the userEmail in the session. If userEmail exists, render the logout link; otherwise, render the register and login links.

```
<% if(locals.userEmail) { %>
  <li class="nav-item">
    <a class="nav-link active" aria-current="page"
href="/logout">Logout</a>
  </li>
<% } else { %>
  <li class="nav-item">
    <a class="nav-link active" aria-current="page"
href="/register">Register</a>
  </li>
  <li class="nav-item">
    <a class="nav-link active" aria-current="page"
href="/login">Login</a>
  </li>
<% } %>
```

2. In the ProductController in product.controller.js, when rendering views that require the user's information, pass the userEmail value to the view.

```
getAddProduct(req, res, next) {
  res.render('new-product', {
    errorMessage: null,
    userEmail: req.session.userEmail,
  });
}
```

Similarly, in other methods such as `postAddProduct`, `getUpdateProductView`, `postUpdateProduct`, and `deleteProduct`, pass the `userEmail` value to the corresponding views.

3. In the `UserController` in `user.controller.js`, add a `logout` method to handle the logout functionality. In this method, destroy the session using `req.session.destroy()` and redirect the user to the login page.

```
logout(req, res) {  
  // on logout, destroy the session  
  req.session.destroy((err) => {  
    if (err) {  
      console.log(err);  
    } else {  
      res.redirect('/login');  
    }  
  });  
}
```

4. Finally, in the `index.js` file, map the `/logout` route to the `logout` method in the `UserController`.

```
app.get('/logout', usersController.logout);
```

Understanding Cookie

- When a client connects to a server, a session is created to maintain the state and context of the client's interaction with the server.
- The server generates a unique session ID for the session and sends it to the client.
- The client typically stores this session ID in a cookie. Cookies are small pieces of data stored on the client's browser.
- With each subsequent request sent by the client to the server, the session ID is included in the request, usually through the cookie.
- The server uses the session ID to identify the specific session associated with the client and retrieves the corresponding session data.
- In the context of the inventory management project, an example of utilizing session data could be tracking the last visited products by a user.
- For example, you can store the IDs or details of the last explored products in the user's session data.

Creating Cookie

In the inventory management project, cookies will be used to store the last visit date and time of the user.

To handle cookies in the project, the cookie-parser package will be used. It can be installed using the command `npm i cookie-parser`

1. In the index.js file, import the cookie-parser package to use it in the project:

```
import cookieParser from 'cookie-parser';
```

2. For every request made by the user, the last visited time will be updated. This functionality will be implemented using a middleware called setLastVisit, which will be defined in the lastVisit.middleware.js file within the middlewares folder.

```
export const setLastVisit = (req, res, next) => {  
  // 1. if cookie is set, then add a local variable with last visit  
  time data.  
  
  if (req.cookies.lastVisit) {  
    res.locals.lastVisit = new Date(  
      req.cookies.lastVisit  
    ).toLocaleString();  
  }  
  res.cookie(  
    'lastVisit',  
    new Date().toISOString(),  
    {  
      maxAge: 2 * 24 * 60 * 60 * 1000,  
    }  
  );  
  next();  
};
```

The logic for the setLastVisit middleware is as follows:

- If the lastVisit cookie is set, retrieve its value and add a local variable lastVisit with the formatted date and time.
- Set the lastVisit cookie with the current date and time using res.cookie
- The maxAge option in res.cookie specifies the maximum age of the cookie in milliseconds. In this case, it is set to 2 days.
- Finally, call the next function to continue processing the request.

3. Conditional rendering will be added to show the last visited time only if it is available. The changes will be made in the index.ejs file.

In the index.ejs file, add the following code snippet to display the last visited time:

```
<% if(locals.lastVisit){ %>
<div class="alert alert-primary" role="alert">
  Your Last Visit was on : <%= locals.lastVisit %>
</div>
<% } %>
```

4. In the index.js file, the two middlewares (cookieParser and setLastVisit) will be used to handle cookies and set the last visit time for each request.

Include the following code in index.js:

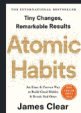

```
app.use(cookieParser());
app.use(setLastVisit);
```

5. View of the application:

[Inventory](#) [Products](#) [New Product](#) [Logout](#)

Your Last Visit was on : 29/5/2023, 6:15:36 pm

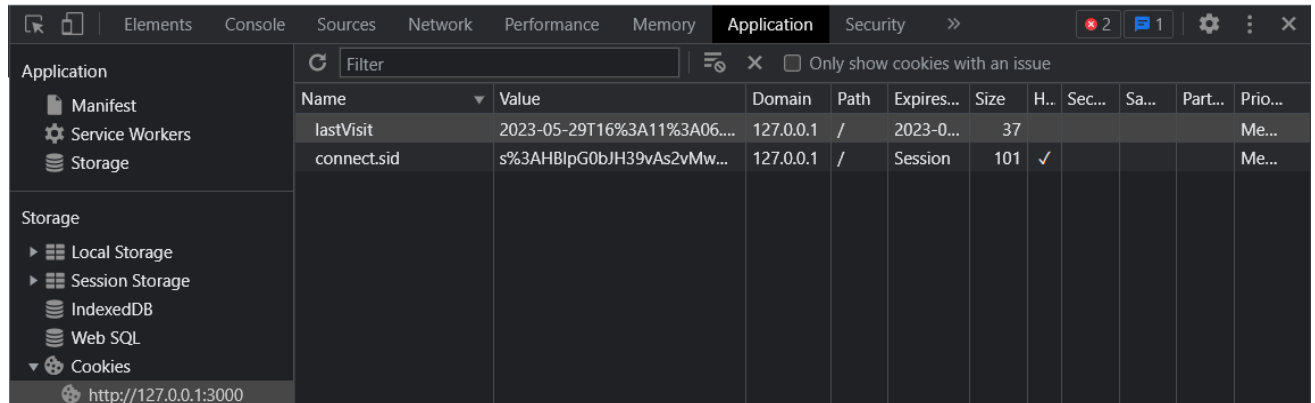
Products

ID	Name	Description	Price	Image	Actions
1	Product 1	Description for Product 10	19.99		<button>Update</button> <button>Delete</button>
2	Product 2	Description for Product 2	29.99		<button>Update</button> <button>Delete</button>

6. To see the cookies stored on the client in our application, we can follow these steps:

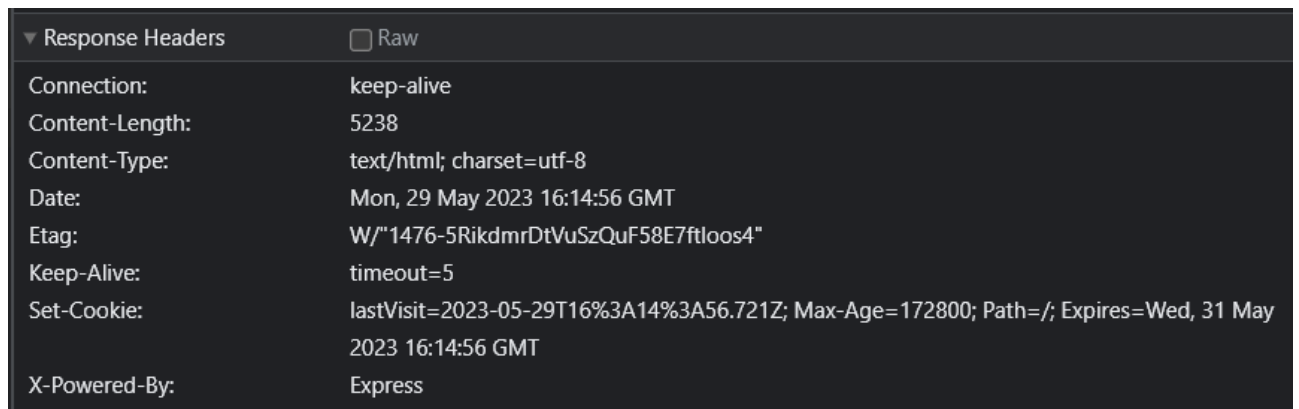
1. Open the web browser and navigate to the URL where the application is running.
2. Right-click on the page and select "Inspect" or "Inspect Element" from the context menu. This will open the browser's developer tools.
3. In the developer tools, locate the "Application" tab. Click on it to switch to the Application panel.
4. In the left sidebar of the Application panel, expand the "Cookies" section. Here, you will find a list of cookies associated with the current URL.
5. Look for the specific cookie you want to inspect, such as "lastVisit".

By clicking on the cookie, you can see its details, including its name, value, domain, path, expiration date, and other attributes.

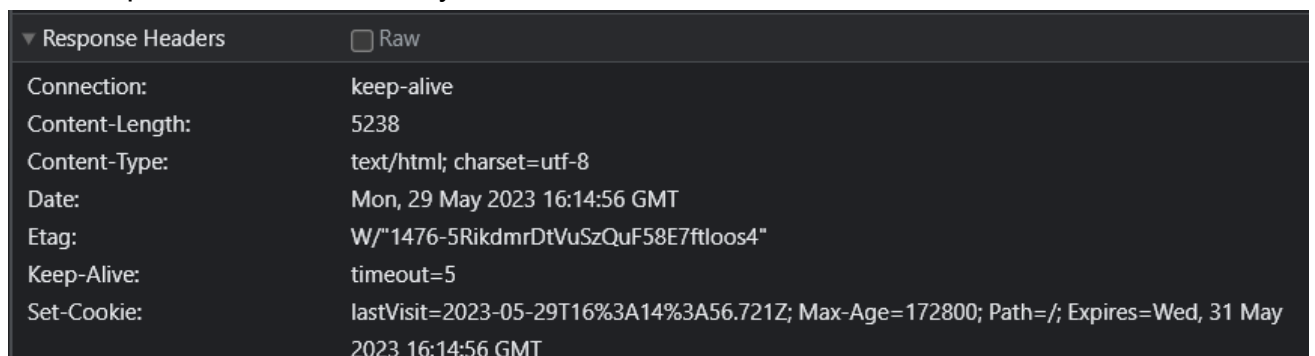


Additionally, you can inspect cookies in the Network tab of the developer tools.

In the request headers section, look for the "Cookie" header. It contains the cookies sent by the client to the server.



In the response headers section, you can find the "Set-Cookie" header, which includes any new or updated cookies sent by the server to the client.



Deleting Cookie

To delete cookies in our application, we can follow these steps:

1. Open the file `user.controller.js` and locate the `logout` function.
2. Inside the `logout` function, after destroying the session using `req.session.destroy()`, add the following code to delete the specific cookie:

```
logout(req, res) {  
  // on logout, destroy the session  
  req.session.destroy((err) => {  
    if (err) {  
      console.log(err);  
    } else {  
      res.redirect('/login');  
    }  
  });  
  res.clearCookie('lastVisit');  
}
```

The `clearCookie` function is called on the response object `res` and takes the name of the cookie to be deleted as an argument. In this case, we are deleting the `"lastVisit"` cookie.

This code ensures that when the user logs out, the `"lastVisit"` cookie is cleared from the client's browser.

Additionally, to avoid setting the `"lastVisit"` cookie on the `logout` request, we can make the following changes in the `index.js` file:

1. Locate the root route (`"/"`) where the `getProducts` function is called.
2. Add the `setLastVisit` middleware before the `auth` middleware to ensure that the `"lastVisit"` cookie is set only on root route GET requests:

```
//app.use(setLastVisit)  
  
app.get(  
  '/',  
  setLastVisit,  
  auth,  
  productsController.getProducts  
);
```

By adding `setLastVisit` as a middleware only for the root route, we can prevent the `"lastVisit"` cookie from being set when the `logout` request is made.

These changes will ensure that the "lastVisit" cookie is deleted when the user logs out and that it is not set again during the logout request.

Best Practices

When working with MVC (Model-View-Controller) applications, it is important to follow certain best practices to ensure maintainability, readability, and scalability. Here are some best practices:

1. **Separation of Concerns:** Follow the principle of separating different concerns into distinct components. The model handles data and business logic, the view handles presentation and user interface, and the controller handles the interaction between the model and the view. This promotes modularity and makes the code easier to understand and maintain.
2. **DRY (Don't Repeat Yourself):** Avoid duplicating code by using reusable components and modularizing the application. For example, use of a layout page that can be shared across multiple views to avoid repeating common elements.
3. **Use of Middlewares:** Middlewares provide a way to decouple complex logic from the controller and allow for reusable and configurable components. Middleware can be used for various purposes such as authentication, logging, error handling, and more. They help in keeping the codebase clean and organized.
4. **Modularize:** Divide your application into smaller modules or components based on their functionality. This promotes code reusability, maintainability, and scalability. Each module should have a clear responsibility, such as a user module, order module, or product module.
5. **Naming Conventions:** Use meaningful and descriptive names for functions, classes, variables, and files. Follow a consistent naming convention that is easily understandable by other developers. This improves code readability and makes it easier to collaborate and maintain the codebase.
6. **Security Implementation:** Implement proper security measures such as authentication and authorization. Ensure that user access is authenticated and authorized based on their roles and permissions.

Summarising it

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

- Learned how to handle file uploads, specifically for images, using the multer library.
- Explored the concept of sessions and how they can be used to maintain user authentication state in an application.
- Implemented a registration and login functionality to allow users to create accounts and authenticate themselves.
- Secured the application by implementing session-based authentication and ensuring that only authenticated users can access protected routes.
- Added a logout feature that destroys the session and clears user-related data. This ensures proper session management and user privacy.
- Learned about cookies and used the cookie-parser library to create and manipulate cookies in the application.
- Implemented the deletion of cookies, specifically the "lastVisit" cookie, when the user logs out to maintain data privacy.
- Discussed best practices that were followed while developing the inventory management application.

Some Additional Resources:

- [How to Upload Image Using Multer in Node.js?](#)
- [How session works in express-session?](#)
- [Understanding Cookies and Implementing them in Node.js](#)
- [Node.js Best Practices](#)