Dharmsinh Desai University

Department of MCA

Semester – II

Seminar Report

on

# Express Js

By

**Name : <u>Nikhil R. Lathiya</u>** **Roll No : <u>MCA024</u>**

## INDEX

## 1. Introduction To Express Js

Express.js, or simply Express, is a back end web application framework for building RESTful APIs with Node.js, released as free and open-source software under the MIT License. It is designed for building web applications and APIs. It has been called the de facto standard server framework for Node.js.

Express.js is a powerful framework for building web applications with Node.js. It provides a minimalist and flexible approach to web development, allowing developers to create robust and scalable applications quickly and efficiently.

At its core, Express.js simplifies the process of handling HTTP requests and responses. With its intuitive routing system, developers can define routes for different URLs and HTTP methods, making it easy to handle various types of requests. Additionally, Express.js provides middleware, which are functions that can intercept and modify incoming requests or outgoing responses. This allows for common functionalities like authentication, logging, and error handling to be easily implemented and reused across different parts of the application.

One of the key advantages of Express.js is its simplicity and minimalistic design. Unlike other frameworks that come with a lot of built-in features and conventions, Express.js gives developers the freedom to structure their applications according to their specific needs. This flexibility makes it ideal for both small projects and large-scale applications.

Express.js also has a vibrant ecosystem with a wide range of plugins and modules available through npm, the Node.js package manager. These modules can extend the functionality of Express.js, adding features such as template engines, database integration, and authentication mechanisms, further enhancing the capabilities of your web applications.

## 1.2 Features of  Express.Js

**Built for Node.js:**

- Express.js is specifically designed to work seamlessly with Node.js, taking advantage of its asynchronous, event-driven architecture.

**Routing Simplified:**

- It simplifies the definition of routes, making it easy to handle different HTTP methods (GET, POST, etc.) and create modular and organized applications.

**Middleware Functionality:**

- Express.js utilizes middleware functions, allowing developers to enhance the functionality of their applications by adding modular components that can handle specific tasks during the request-response cycle.

**Template Engine Integration:**

- It seamlessly integrates with template engines like EJS, Pug, or Handlebars, enabling dynamic content rendering on the server-side.

**RESTful API Support:**

- It facilitates the creation of RESTful APIs, allowing developers to build scalable and maintainable backend services for their applications.

**Error Handling:**

- Express.js offers mechanisms for handling errors in a straightforward manner, making it easier to identify and address issues during development.

## 1.3 Setup and Installation

Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. Follow these steps to install Express.js and set up your development environment.

**Steps to install Enpress js :**

- **Prerequisites**: Before installing Express.js, ensure that you have Node.js and npm (Node Package Manager) installed on your system. You can download and install Node.js from the official website: nodejs.org.

- **Create a New Directory:** Create a new directory for your Express.js project. You can do this via the command line using the mkdir command.

  *For example: mkdir my-express-app*

- **Navigate to the Project Directory**: Move into the newly created directory using the cd command:

  ***npm init -y***

This command creates a package.json file with default values in your project directory.

- **Install Express.js:** Install Express.js as a dependency for your project using npm.

  Run the following command: ***npm install express***

This command installs Express.js and adds it to your project's package.json file under the dependencies section.

- **Create a Simple Express App:** Create a new JavaScript file (e.g., app.js) in your project directory and open it in a text editor. Then, write a simple Express.js application:

```
const express = require('express');
const app = express();


app.get('/', (req, res) => {
  res.send('Hello, Express!');
});


app.listen(3000, () => {
  console.log('Express server is running on http://localhost:3000');
});
```

- **Run the Express App**: Save the changes to your app.js file and return to the command line.

  Run the following command to start your Express.js server: **node app.js**

Your Express.js server should now be running, and **you can access it by visiting http://localhost:3000 in your web browser.** You should see the message "Hello, Express!" displayed in the browser.

## 2. Node.Js Vs Express.Js

**Node.js:**

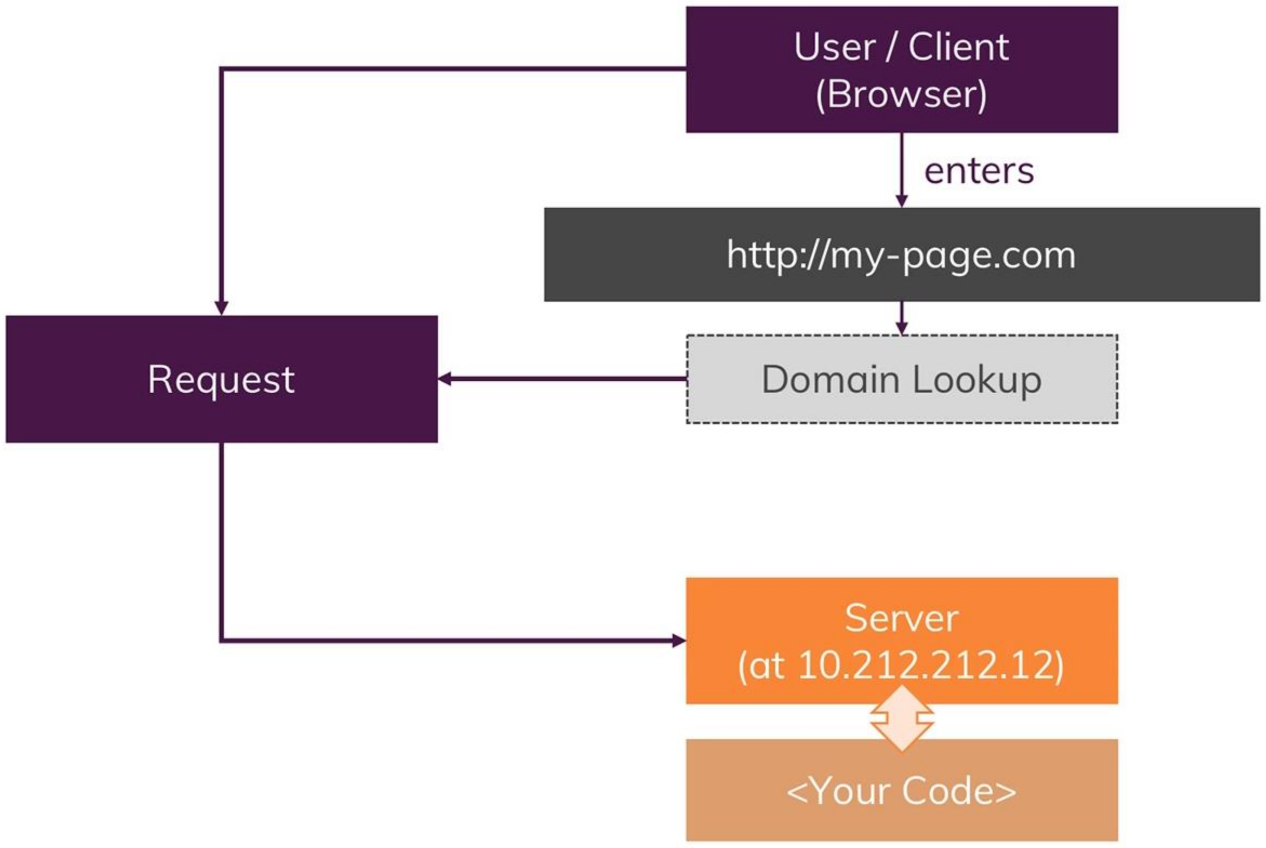
- A runtime environment for executing JavaScript code on the server-side.

- Empowers developers to build scalable and high-performance network applications.
- Provides event-driven architecture, non-blocking I/O operations, and asynchronous programming capabilities.
- Offers a vast ecosystem of modules through npm (Node Package Manager).

**Express.js:**

- A minimalist and flexible web application framework for Node.js.
- Simplifies the process of building robust web applications and APIs.
- Provides essential features like routing, middleware support, and template rendering.
- Offers a lightweight and unopinionated structure, allowing developers to design applications as per their requirements.
- In essence, Node.js lays the foundation for server-side JavaScript execution, while Express.js enhances development by providing a structured framework tailored for web applications and APIs.

## 2.1 Basic Information About How Web-page Works



## 3. Introduction to Routing :

**Definition:**

Routing in web development refers to the process of defining how an application responds to client requests to specific URLs (Uniform Resource Locators). It involves mapping HTTP requests to corresponding application logic or resources. In Node.js web applications, routing is commonly handled by frameworks like Express.js.

In simple Terms Routing is like giving directions to different parts of your web application.

Example :-

Imagine you're building a basic website with different pages:

When a user visits the homepage, they should see a welcome message.

If they go to '/about', they should see information about your website.

When they navigate to '/contact', they should find your contact

## 3.1 Routing in Node.js VS  Express.js

**Node.js:**

In Node.js, routing typically involves manually parsing incoming HTTP requests and defining routes based on the requested URL paths and HTTP methods. Developers often use the built-in http module to create a basic HTTP server and handle routing logic using conditional statements or switch-case blocks.

### Example :-

```
const http = require('http');


const server = http.createServer((req, res) => {
   if (req.url === '/')
   {
      // Set status code to 200 for the root URL
      res.statusCode = 200;
      res.write('Hello, World!');
   }
   else if (req.url === '/about')
   {
      // Set status code to 200 for the '/about' URL
      res.statusCode = 200;
      res.write('About Page');
   }
   else
   {
      // Set status code to 404 for any other URL
```

```
        res.statusCode = 404;
        res.write('Not Found');
    }
    res.end();
});

server.listen(3000, () => {
    console.log('Server is running on http://localhost:3000');
});
```

## Routing in Express.js:

Express.js simplifies routing by providing a dedicated router object that allows developers to define routes for different URL paths and HTTP methods. Routing is more declarative and organized compared to native Node.js routing, enabling cleaner and more maintainable code.

Example :-

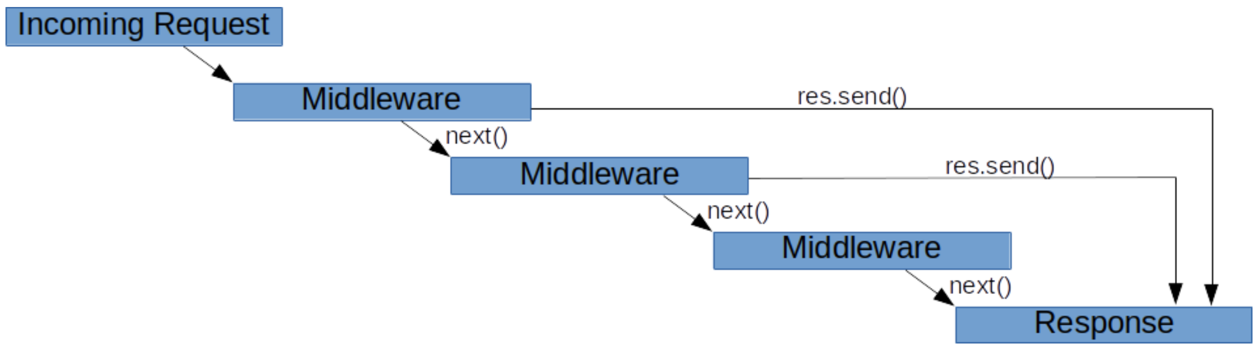
```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
    res.send('Hello, World!');
});

app.get('/about', (req, res) => {
  res.send('About Page');
});

app.use((req, res) => {
  res.status(404).send('Not Found');
});

app.listen(3000, () => {
  console.log('Express server is running on http://localhost:3000');
});
```

## 3.2 Benefits of Express.Js Routing

- Easy Configuration: Define routes with simple syntax, reducing development time.
- Modular Design: Routes can be organized into separate files for better code organization.
- Middleware Integration: it will become very easy to use middleware functions to enhance routing functionality.

## 4. Middleware

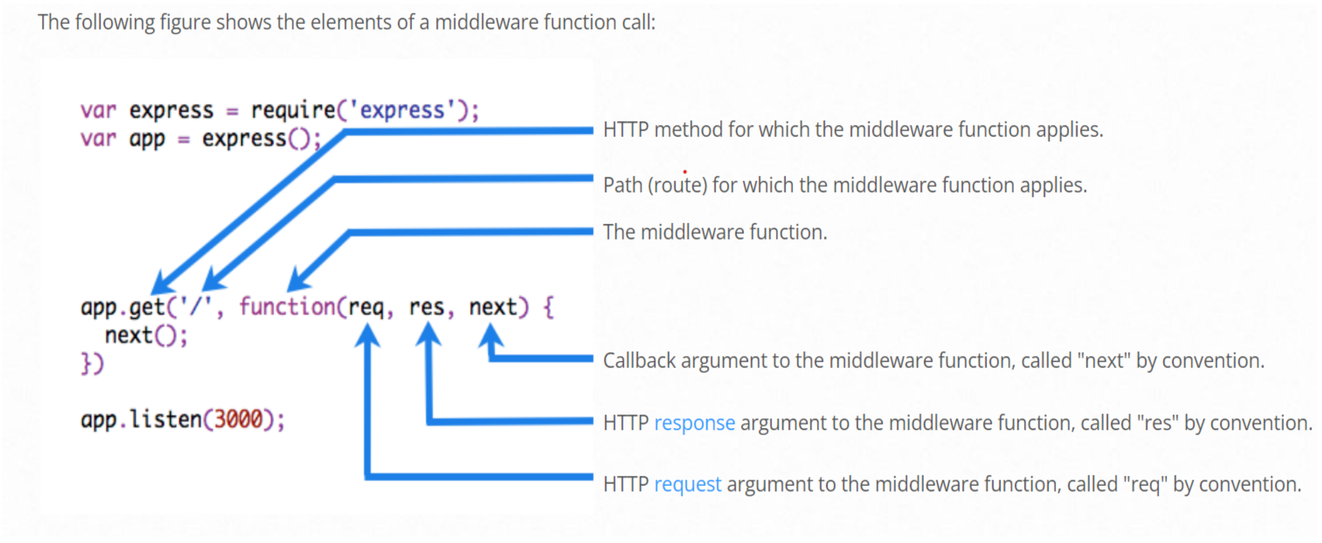
Middleware is software that acts as an intermediary between different applications, systems, or components within a computing environment. It enables communication, integration, and interaction between these disparate elements by providing services such as message routing, data transformation, security enforcement, and transaction management. Middleware abstracts the complexities of underlying infrastructure and facilitates interoperability, scalability, and flexibility in distributed computing environments.



### 4.1 Benifits of Middleware :-

- **Interoperability**: Enables seamless communication between different systems.
- **Abstraction**: Simplifies development by hiding underlying complexities.
- **Scalability**: Supports distributed computing, adapting to changing workloads.
- **Performance**: Optimizes system performance with features like caching and load balancing.
- **Security**: Provides robust security mechanisms to protect data integrity and confidentiality.

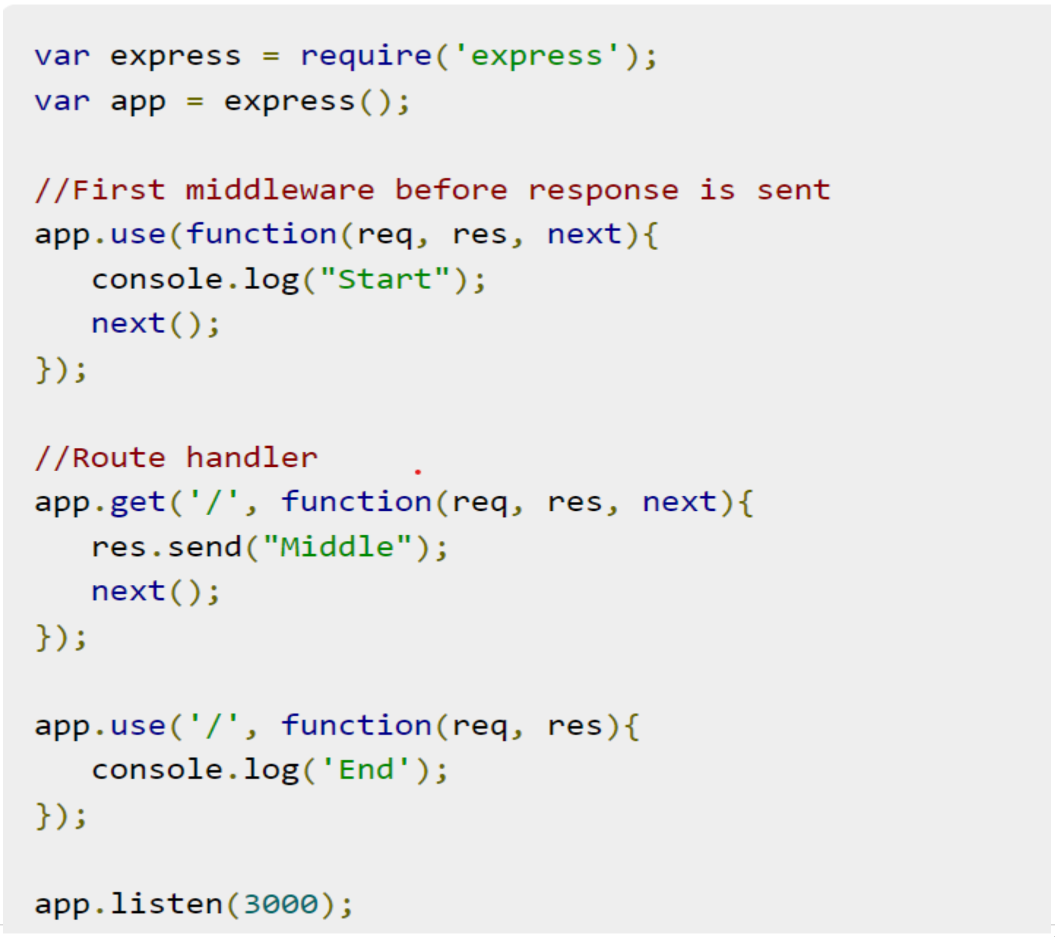
## 4.2 Elements of a Middleware


The following figure shows the elements of a middleware function call:

```
var express = require('express');
var app = express();

app.get('/', function(req, res, next) {
  next();
})

app.listen(3000);
```

HTTP method for which the middleware function applies.
Path (route) for which the middleware function applies.
The middleware function.
Callback argument to the middleware function, called "next" by convention.
HTTP response argument to the middleware function, called "res" by convention.
HTTP request argument to the middleware function, called "req" by convention.

## 4.3 Example Of Middleware ;-



```
var express = require('express');
var app = express();

//First middleware before response is sent
app.use(function(req, res, next){
   console.log("Start");
   next();
});

//Route handler
app.get('/', function(req, res, next){
   res.send("Middle");
   next();
});

app.use('/', function(req, res){
   console.log('End');
});

app.listen(3000);
```

## 5. Error Handling

- **Error Handling** refers to how Express catches and processes errors that occur both synchronously and asynchronously. Express comes with a ***default error handler*** so you don't need to write your own to get started.
- **Catching Errors**
- It's important to ensure that Express catches all errors that occur while running route handlers and middleware.
- Errors that occur in synchronous code inside route handlers and middleware require no extra work. If synchronous code throws an error, then Express will catch and process it. For example:

### 5.1 Example of Error Handling :

```
app.get('/', (req, res) => {
  throw new Error('BROKEN') // Express will catch this on its own.
})
```

For errors returned from asynchronous functions invoked by route handlers and middleware, you must pass them to the next() function, where Express will catch and process them. For example**:**

```
app.get('/', (req, res, next) => {
  fs.readFile('/file-does-not-exist', (err, data) => {
    if (err) {
      next(err) // Pass errors to Express.
    } else {
      res.send(data)
    }
  })
})
```

Starting with Express 5, route handlers and middleware that return a Promise will call next(value) automatically when they reject or throw an error. For example:

```
app.get('/user/:id', async (req, res, next) => {
  const user = await getUserById(req.params.id)
  res.send(user)
})
```

If getUserById throws an error or rejects, next will be called with either the thrown error or the rejected value. If no rejected value is provided, next will be called with a default Error object provided by the Express router.

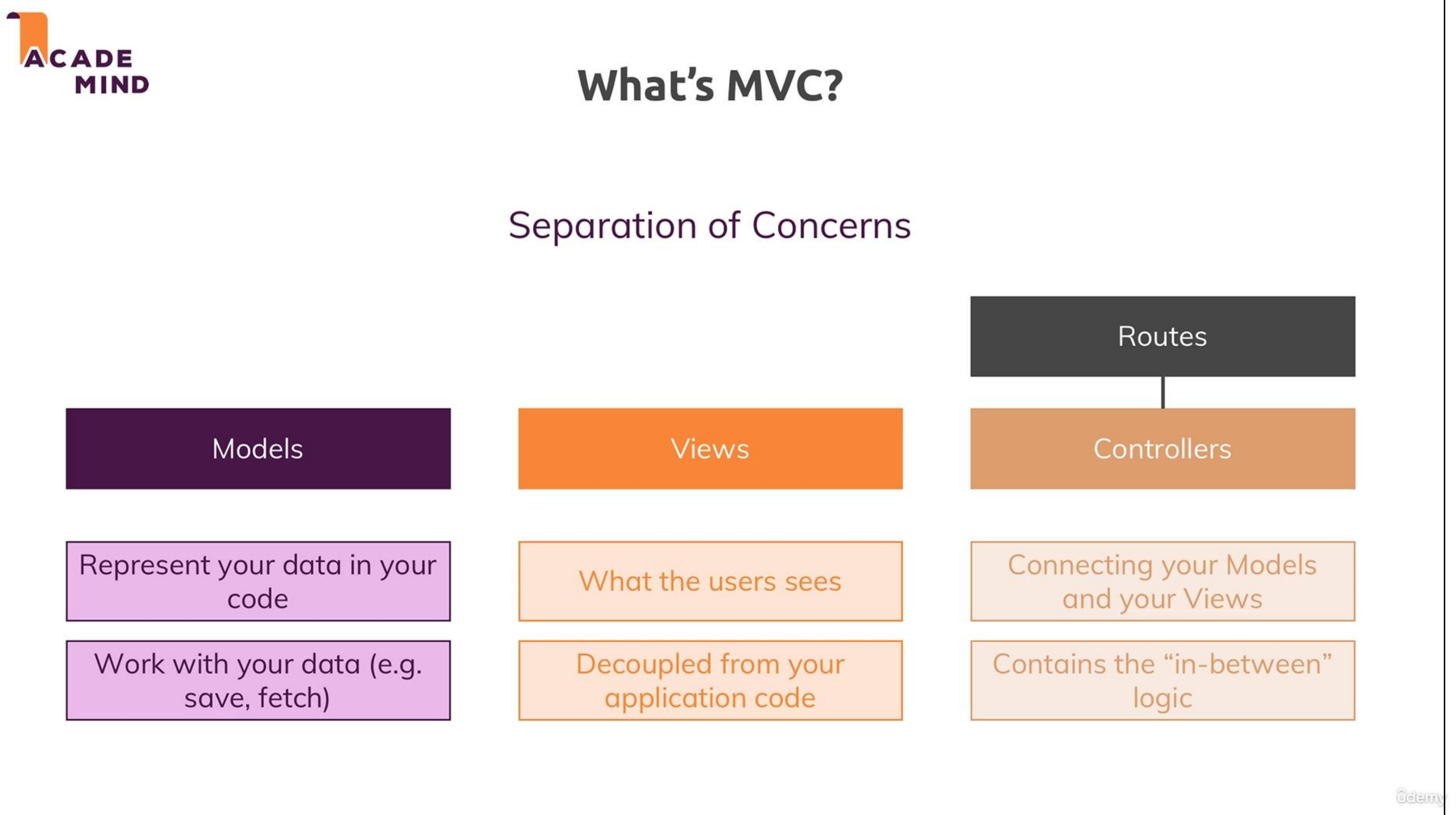
If you pass anything to the next() function (except the string 'route'), Express regards the current request as being an error and will skip any remaining non-error handling routing and middleware functions.

If the callback in a sequence provides no data, only errors, you can simplify this code as follows:

```
app.get('/', [
  function (req, res, next) {
    fs.writeFile('/inaccessible-path', 'data', next)
  },
  function (req, res) {
    res.send('OK')
  }
])
```

in the above example, next is provided as the callback for fs.writeFile, which is called with or without errors. If there is no error, the second handler is executed, otherwise Express catches and processes the error.

## 6.0 Introduction to MVC (MODLES VIEW CONTROLLER)



ACADEMIND

What's MVC?

Separation of Concerns

| Models | Views | Controllers |
|---|---|---|
| Represent your data in your code | What the users sees | Connecting your Models and your Views |
| Work with your data (e.g. save, fetch) | Decoupled from your application code | Contains the "in-between" logic |

Routes

Udemy

**MVC**, which stands for ***Model-View-Controller***, is a software architectural pattern used primarily in web development to organize code and separate concerns. It divides an application into three interconnected components:

- **Model**: Represents the data and business logic of the application. It encapsulates data manipulation and validation, as well as interactions with the database or any other data source.

- **View**: Represents the presentation layer of the application. It is responsible for rendering the user interface and displaying data to the user. Views are typically HTML templates that may contain dynamic content generated by the controller.

- **Controller**: Acts as an intermediary between the model and the view. It receives user input from the view, processes it (such as validating input and updating the model), and determines the appropriate response to send back to the view for rendering.

- The MVC pattern promotes separation of concerns, modularization, and maintainability by organizing code into distinct components with clearly defined responsibilities. It also facilitates code reuse, as changes to one component (e.g., the model) do not necessitate modifications to other components (e.g., the view or controller).

## 6.1 Example of MVC :-

**Model**: Represents the data and business logic. In this case, it's the to-do items.

```
// model/todoModel.js
const todos = [];

function addTodo(todo) {
  todos.push(todo);
}

function getTodos() {
  return todos;
}

module.exports = {
  addTodo,
  getTodos
};
```

**View**: Represents the presentation layer. In our case, it's a simple HTML page with EJS templating for dynamic content.

```
<!-- views/todoView.ejs -->

<html>

<head>

  <title>To-Do List</title>

</head>

<body>

  <h1>To-Do List</h1>

  <ul>

    <% todos.forEach(todo => { %>

      <li><%= todo %></li>

    <% }); %>

  </ul>

  <form action="/add" method="post">

    <input type="text" name="todo" placeholder="Enter new task">

    <button type="submit">Add</button>

  </form>

</body>

</html>
```

**Controller**: Handles requests, interacts with the model, and renders views.

```
// controller/todoController.js
const express = require('express');
const router = express.Router();
const todoModel = require('../model/todoModel');

// Route for rendering the to-do list
router.get('/', (req, res) => {
   const todos = todoModel.getTodos();
   res.render('todoView', { todos });
});

// Route for adding a new to-do item
router.post('/add', (req, res) => {
   const todo = req.body.todo;
   todoModel.addTodo(todo);
   res.redirect('/');
});

module.exports = router;
```

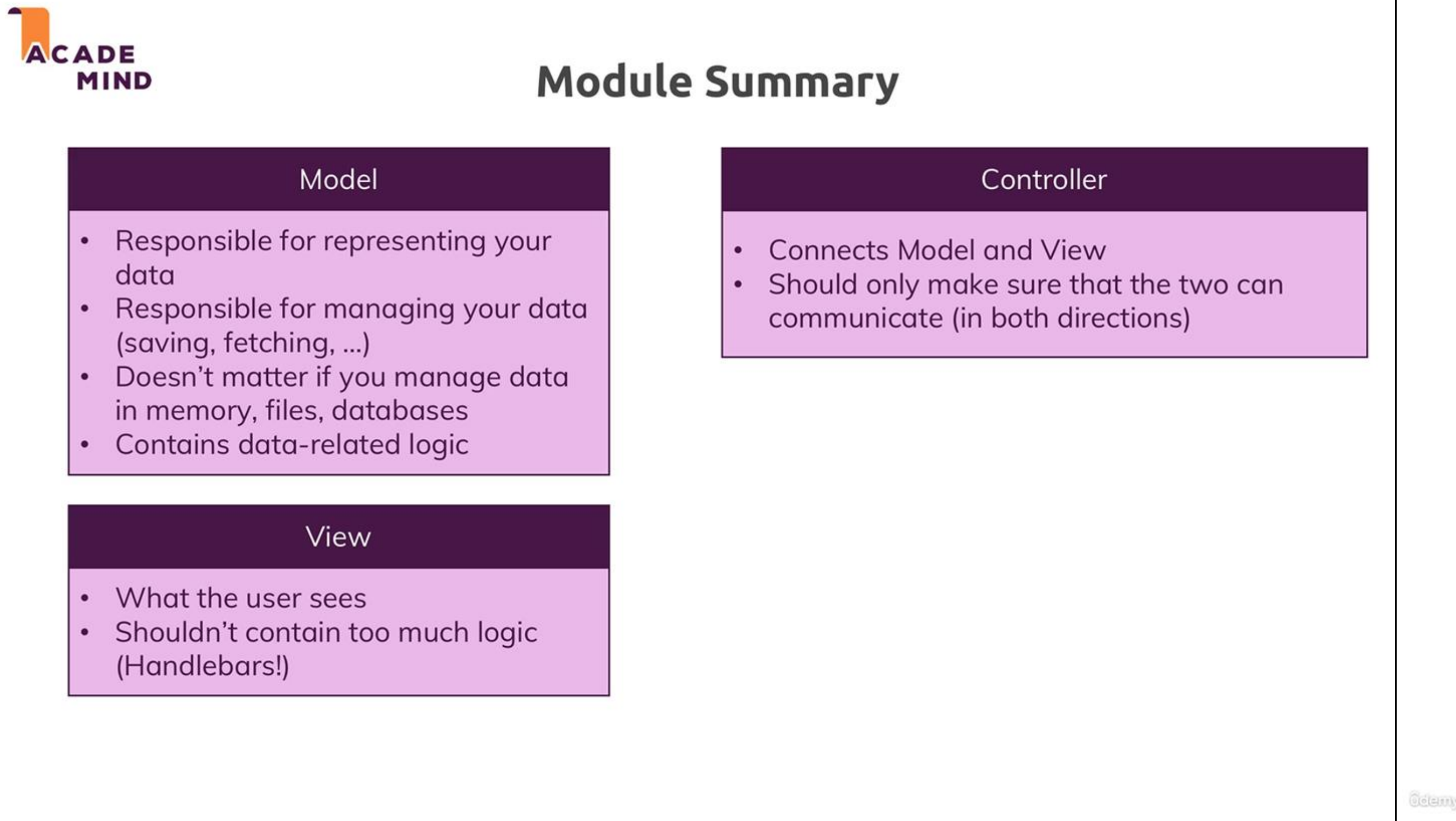
## In this simplified example:

- ***The model*** (todoModel.js) contains an array to store to-do items and functions to add a new to-do item (addTodo) and retrieve all to-do items (getTodos).

- ***The view*** (todoView.ejs) is a simple HTML page with a list of to-do items displayed dynamically using EJS templating. It also contains a form to add a new to-do item.

- ***The controller*** (todoController.js) defines routes for rendering the to-do list and adding new to-do items. When a new to-do item is added, it interacts with the model to update the data and then redirects back to the home page to display the updated to-do list.

- This example demonstrates a basic implementation of MVC in an Express.js application, keeping concerns separated and code organized.



## Module Summary

**Model**

- Responsible for representing your data
- Responsible for managing your data (saving, fetching, ...)
- Doesn't matter if you manage data in memory, files, databases
- Contains data-related logic

**View**

- What the user sees
- Shouldn't contain too much logic (Handlebars!)

**Controller**

- Connects Model and View
- Should only make sure that the two can communicate (in both directions)

## 7. Introduction to Templating Engines :-

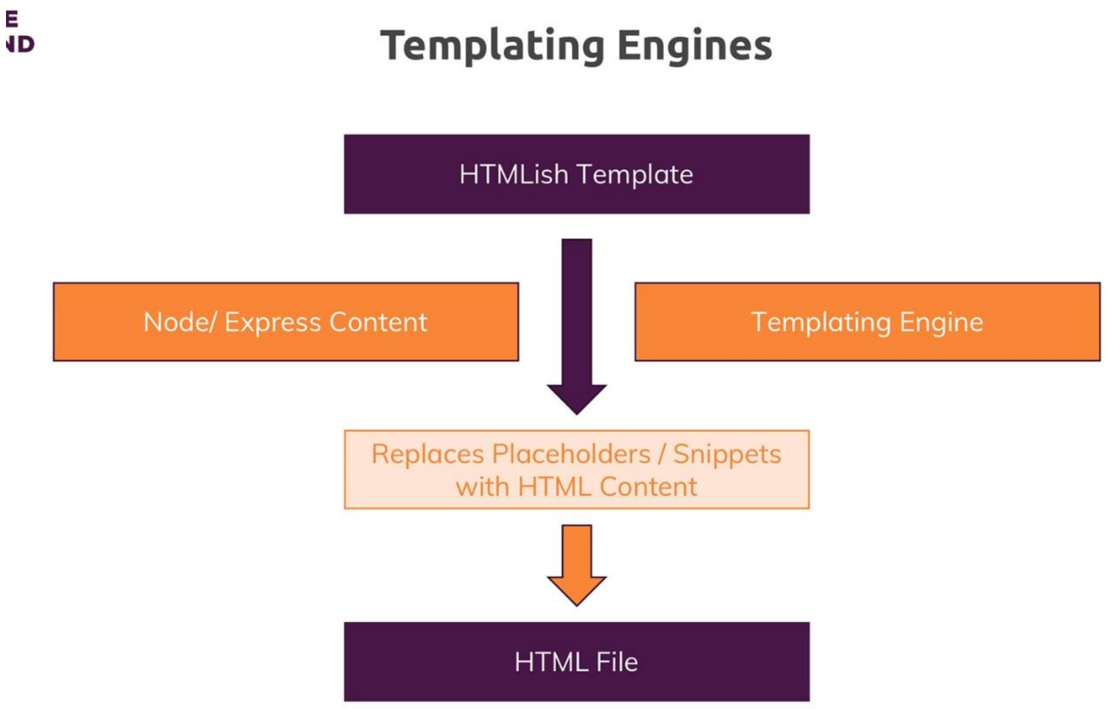
Express.js is a popular web framework for Node.js that simplifies the process of building web applications and APIs. One of the key features of Express.js is its support for templating engines, which allow developers to dynamically generate HTML content based on data.

### 7.1 What is a Templating Engine?

A templating engine is a tool that facilitates the generation of dynamic content by combining static HTML markup with dynamic data. Instead of writing HTML code with embedded JavaScript for data interpolation, templating engines provide a more organized and maintainable approach by separating the presentation layer from the application logic.

### 7.2 Why Use Templating Engines in Express.js?

*Using a templating engine in Express.js offers several advantages:*



- **Dynamic Content**: Templating engines enable developers to generate HTML content dynamically based on data from the server, such as database records or user input.

- **Code Reusability**: Templating engines allow developers to create reusable templates for common page elements (e.g., headers, footers) and layouts, reducing code duplication and improving maintainability.

- **Separation of Concerns**: Templating engines promote separation of concerns by separating the presentation layer (HTML markup) from the application logic (JavaScript code), making code easier to understand and maintain.

- **Improved Developer Experience**: Templating engines provide syntax highlighting, auto-indentation, and other features that enhance the developer experience, making it easier to work with HTML templates.

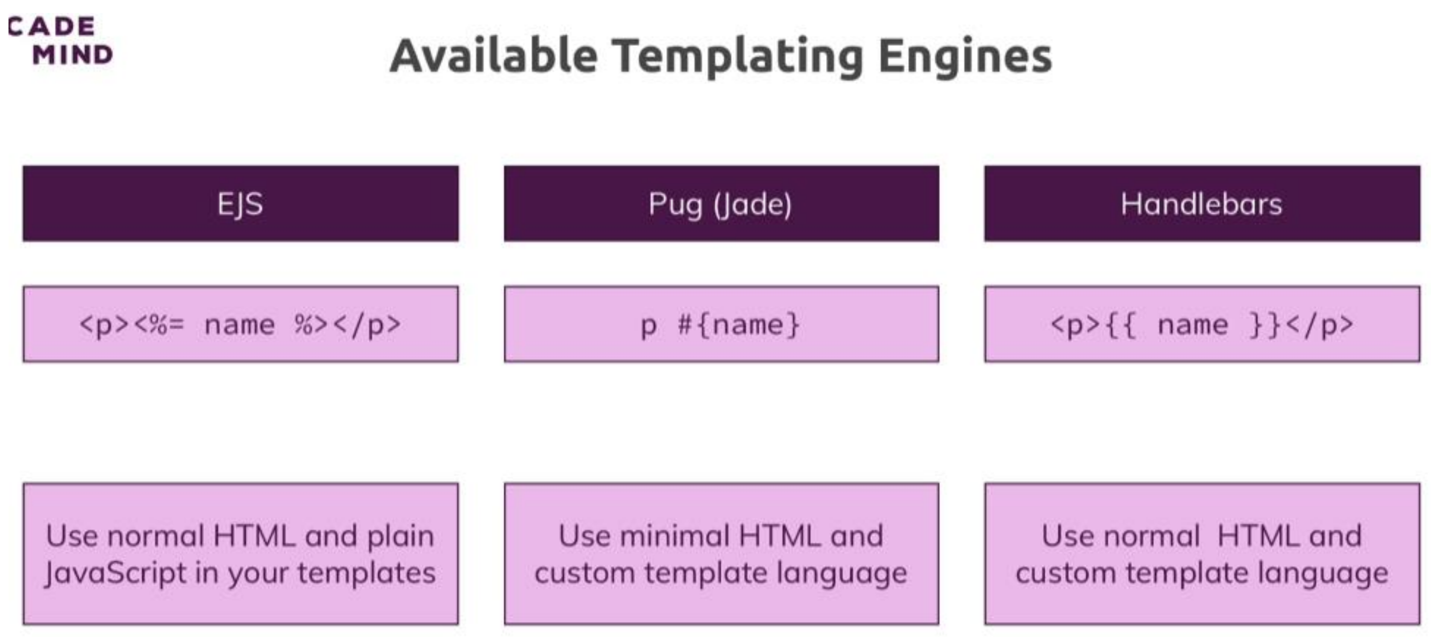
## 7.3 Popular Templating Engines for Express.js

- Express.js supports various templating engines, each with its own syntax and features. Some popular templating engines used in Express.js applications include:

- ***EJS (Embedded JavaScript):*** EJS allows developers to embed JavaScript code directly within HTML templates, making it easy to generate dynamic content.

- ***Pug (formerly known as Jade):*** Pug uses indentation-based syntax to define HTML markup, providing a concise and readable way to create templates.

- ***Handlebars:*** Handlebars provides a simple and intuitive syntax for defining templates with placeholders (placeholders), which are replaced with actual data during rendering.


CADE
MIND

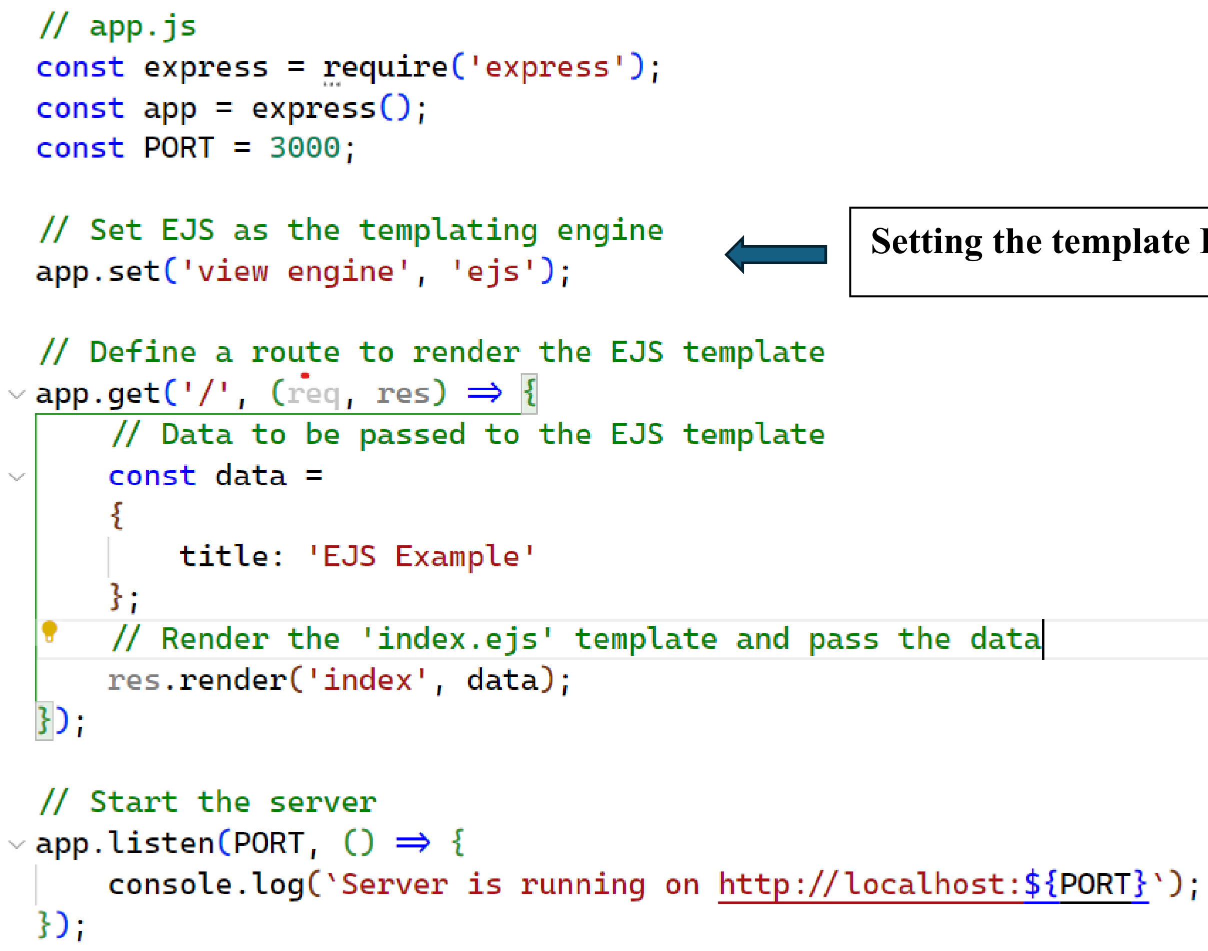
## Available Templating Engines

| EJS | Pug (Jade) | Handlebars |
|---|---|---|
| `<p><%= name %></p>` | `p #{name}` | `<p>{{ name }}</p>` |
| Use normal HTML and plain JavaScript in your templates | Use minimal HTML and custom template language | Use normal HTML and custom template language |

## 7.4 Installation Commands:

```
# Install EJS
npm install ejs

# Install Handlebars
npm install express-handlebars

# Install Pug
npm install pug
```

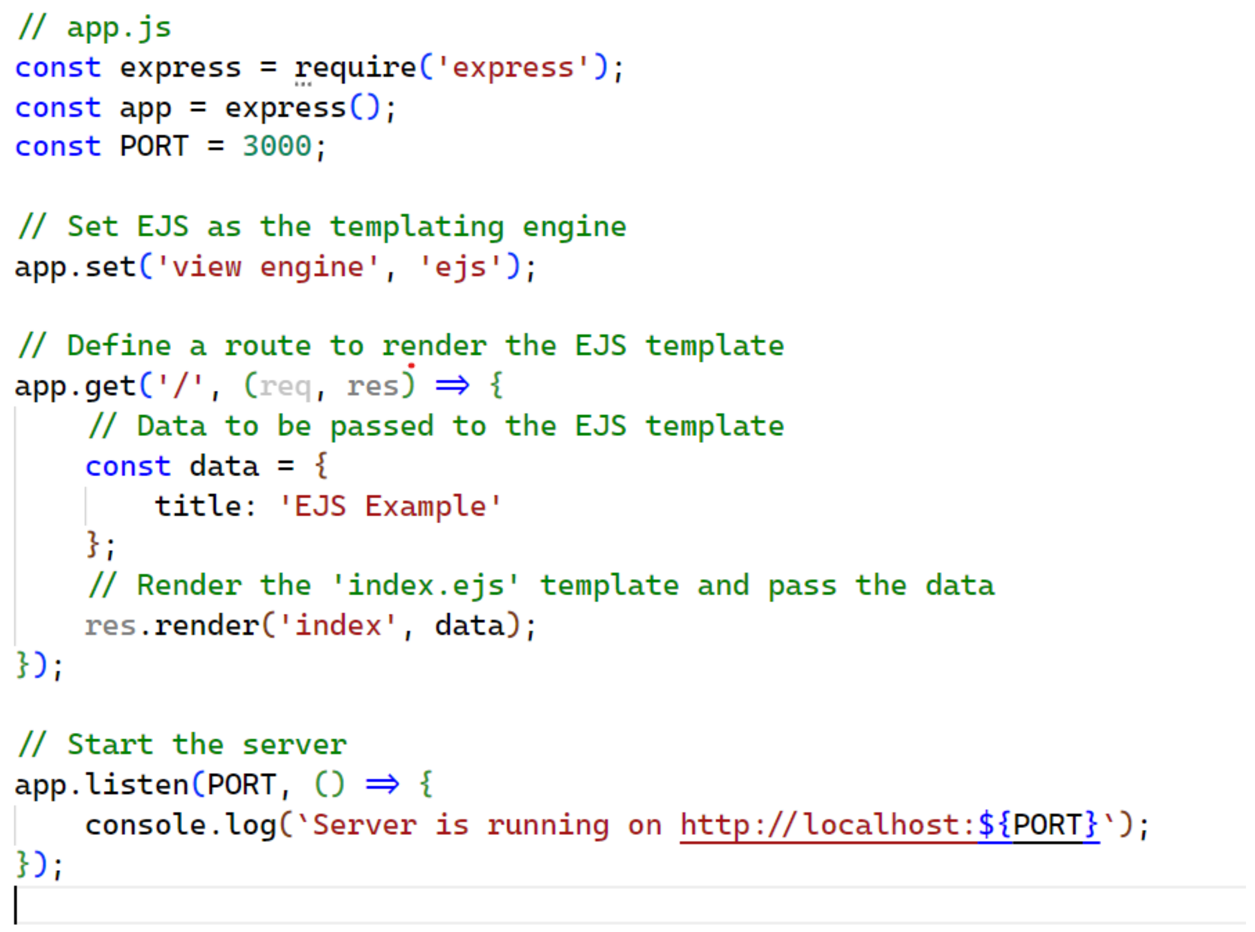
## 7.5 How to set the Template Engines :-



```
// app.js
const express = require('express');
const app = express();
const PORT = 3000;

// Set EJS as the templating engine
app.set('view engine', 'ejs');

// Define a route to render the EJS template
app.get('/', (req, res) => {
    // Data to be passed to the EJS template
    const data =
    {
        title: 'EJS Example'
    };
    // Render the 'index.ejs' template and pass the data
    res.render('index', data);
});

// Start the server
app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
});
```

**Setting the template Engine**

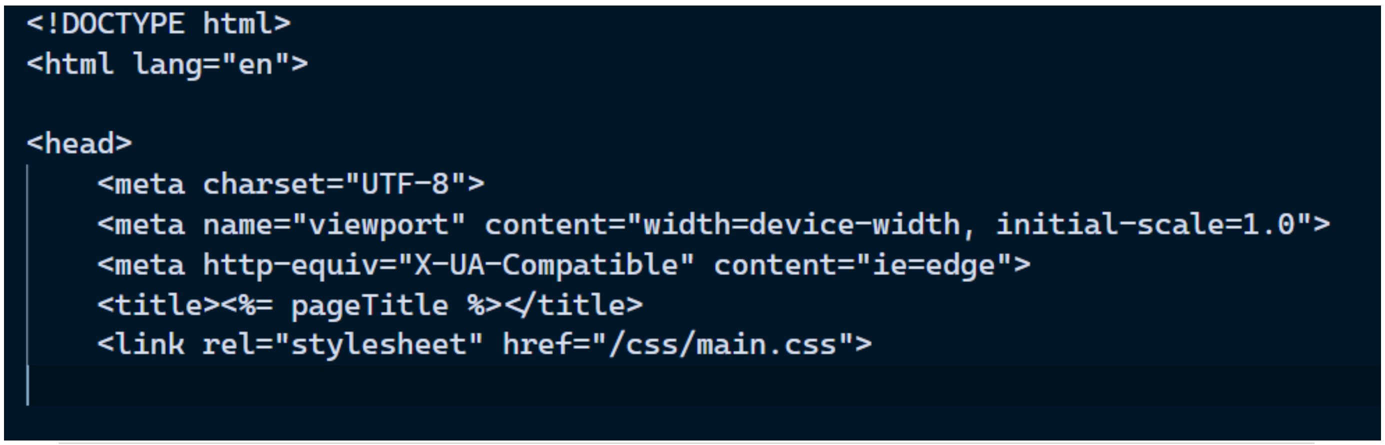
## 7.6 Example of Template Engines using EJS :-



```
// app.js
const express = require('express');
const app = express();
const PORT = 3000;

// Set EJS as the templating engine
app.set('view engine', 'ejs');

// Define a route to render the EJS template
app.get('/', (req, res) => {
    // Data to be passed to the EJS template
    const data = {
        title: 'EJS Example'
    };
    // Render the 'index.ejs' template and pass the data
    res.render('index', data);
});

// Start the server
app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
});
```

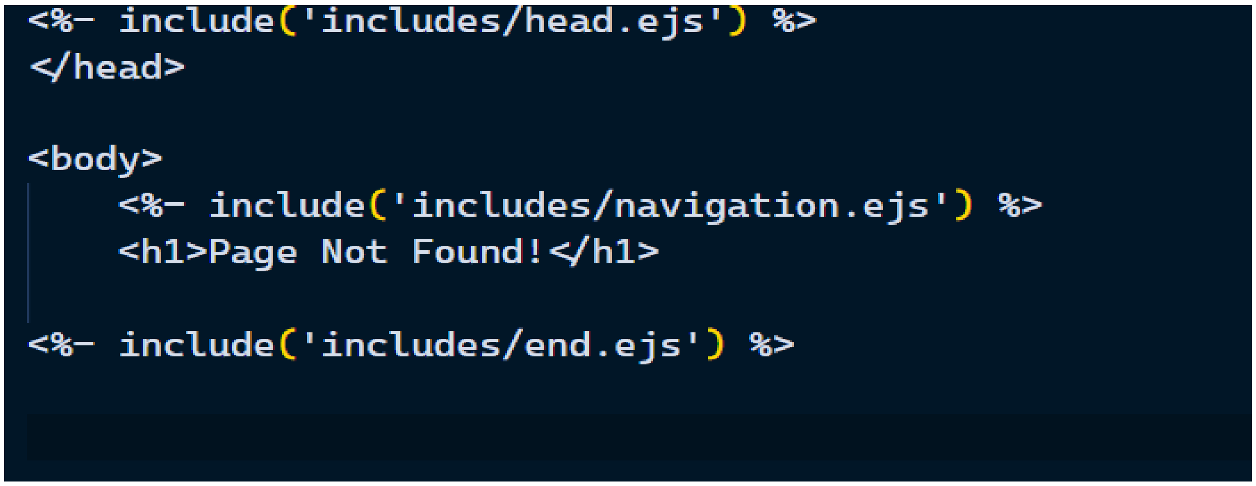
## Making Components of EJS :-

(Head Component)

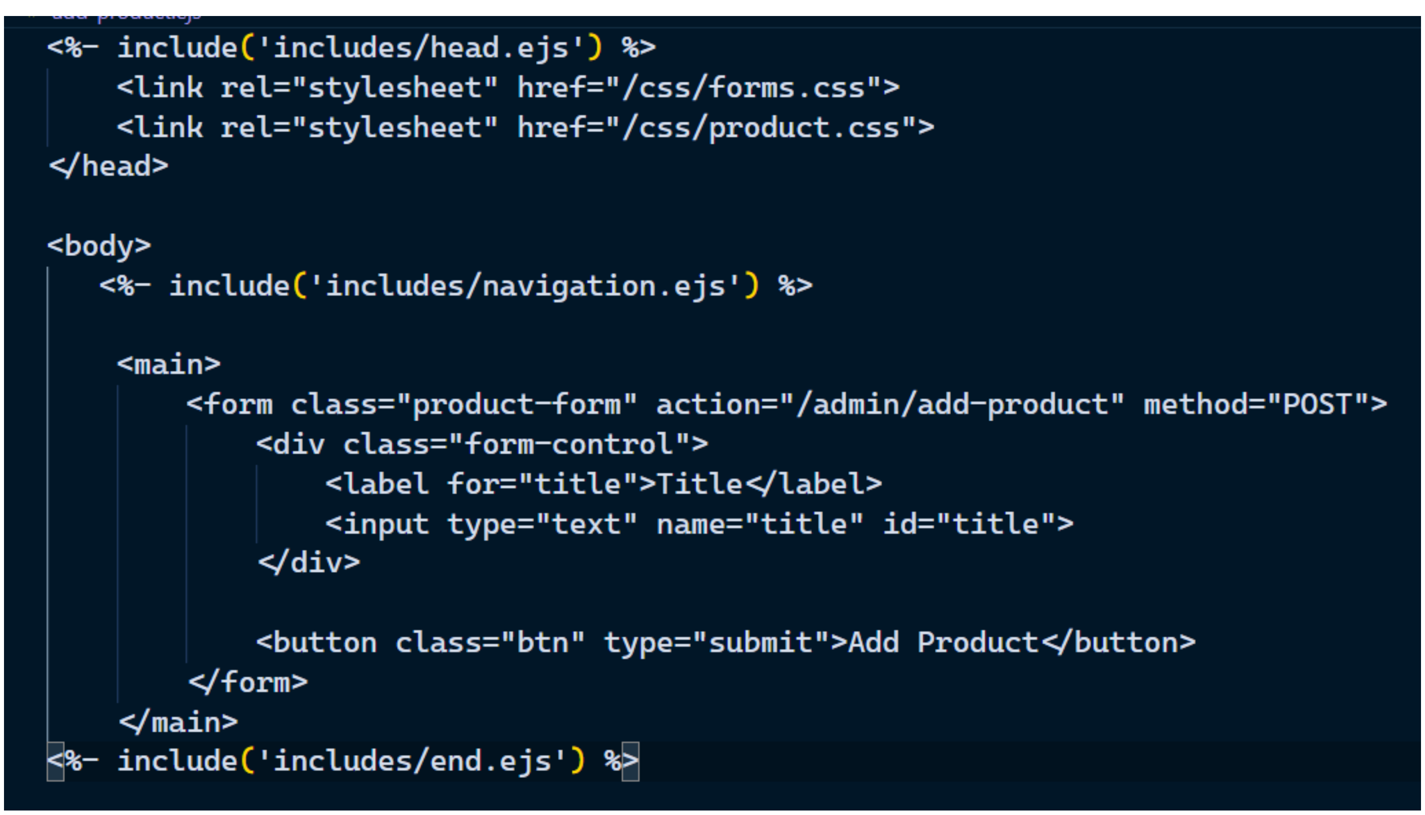


```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title><%= pageTitle %></title>
    <link rel="stylesheet" href="/css/main.css">
```

**Invoking the Head and End component in pageNotFound.ejs PAGE :-**



```
<%- include('includes/head.ejs') %>
</head>

<body>
    <%- include('includes/navigation.ejs') %>
    <h1>Page Not Found!</h1>

<%- include('includes/end.ejs') %>
```



```
<%- include('includes/head.ejs') %>
    <link rel="stylesheet" href="/css/forms.css">
    <link rel="stylesheet" href="/css/product.css">
</head>

<body>
   <%- include('includes/navigation.ejs') %>

    <main>
        <form class="product-form" action="/admin/add-product" method="POST">
            <div class="form-control">
                <label for="title">Title</label>
                <input type="text" name="title" id="title">
            </div>

            <button class="btn" type="submit">Add Product</button>
        </form>
    </main>
<%- include('includes/end.ejs') %>
```

## 8. References

- https://expressjs.com/

  https://nodejs.org/en
- https://en.wikipedia.org/
- https://devdocs.io/express/
- https://www.geeksforgeeks.org/express-js/