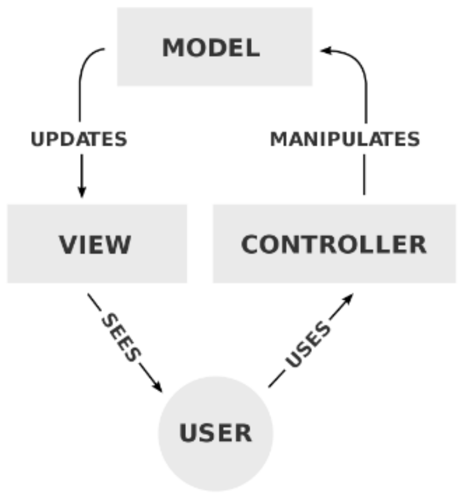
Model View Controller - I

Getting Started with Web Development

Large-scale web applications use architectures and design patterns to structure code and maintain a clear separation of concerns.

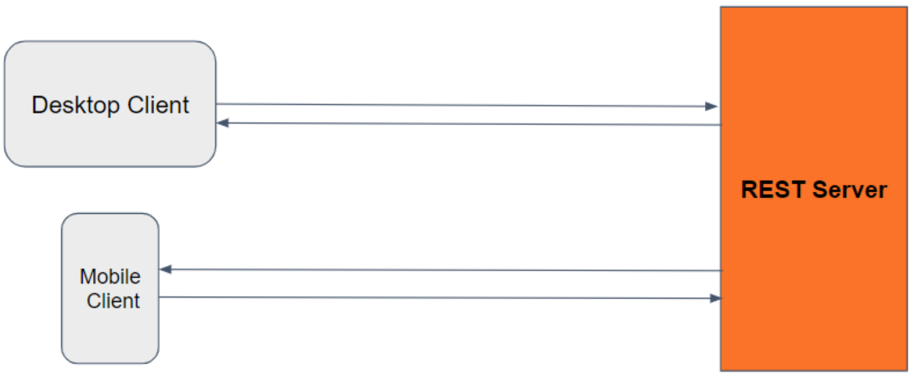
Different Architectures and Patterns

- **Model-View-Controller (MVC)**: A widely-used design pattern that separates data (Model), user interface (View), and control logic (Controller). Promotes a clear separation of concerns, making the code more modular, maintainable, and scalable.



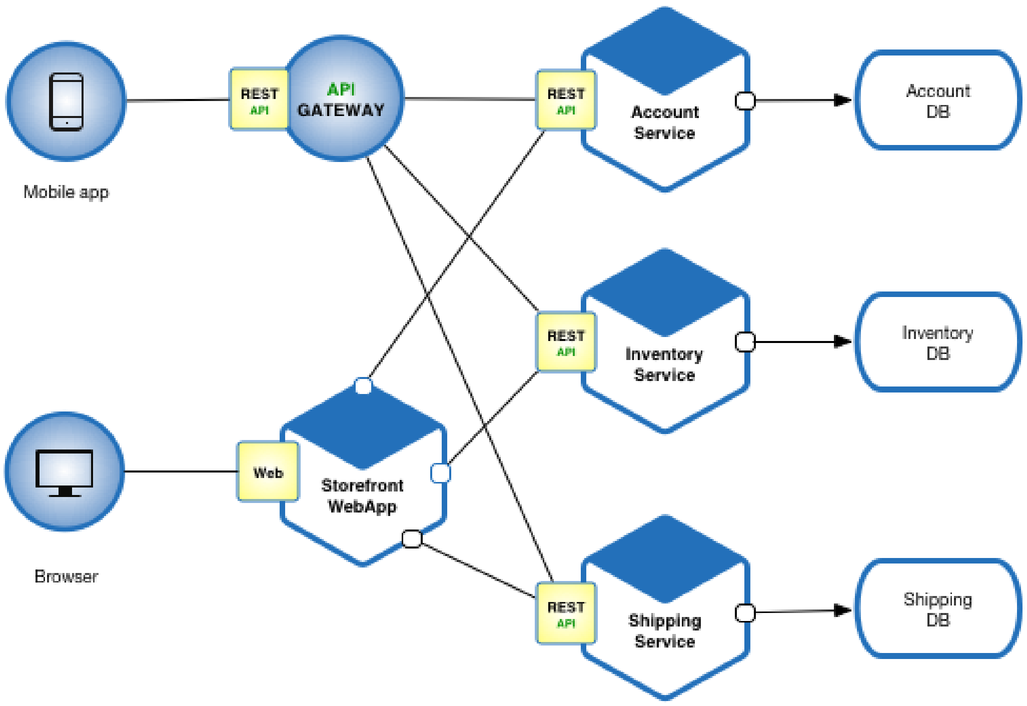
  - **Model**: Represents the data and business logic of the application. It interacts with the database and performs data operations.
  - **View**: Displays the user interface and presents the data to the user. It receives input from the user and sends it to the Controller.
  - **Controller**: Handles user requests, interacts with the Model and View, and processes the application's logic. It updates the Model based on user input and updates the View accordingly.

- **REST API (Representational State Transfer)**



  - It provides cross-platform support.
  - Popular architectural style for designing networked applications.
  - Enables scalability, maintainability, and stateless web services.

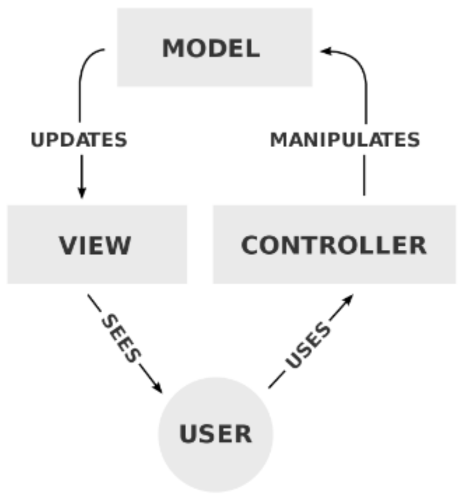
- **Microservices Architecture**



  - Breaks down the application into smaller, independent services that communicate with each other.
  - Each microservice focuses on a specific business capability or functionality.
  - Services are loosely coupled, allowing independent development, deployment, and scalability.
  - Microservices enable scalability and maintainability, as changes in one service do not impact the entire application.

Understanding the Model-View-Controller (MVC) Pattern

The MVC pattern is a popular design pattern used in web development to organize and manage code effectively. It helps separate the concerns of an application, making it easier to maintain and extend.



Components of the MVC Pattern

1. **Model**:
    - Represents the data and business logic of the application.
    - Responsible for retrieving, storing, and processing data.
    - Examples include handling CRUD operations, database interactions, and data validation.

2. **View**:
    - The user interface of the application.
    - Displays data provided by the model to the user.
    - Receives user inputs and forwards them to the controller.
    - Examples include HTML templates, UI components, and user forms.

3. **Controller**:
    - Acts as a bridge between the model and the view.
    - Processes user inputs received from the view.
    - Updates the model based on user actions.
    - Examples include handling user actions, updating data, and coordinating between the model and view.

Benefits of the MVC Pattern

- **Organization and Maintainability**: MVC separates concerns, making code more organized and easier to maintain.
- **Code Reusability**: With clear separation of concerns, components can be reused in different contexts or scenarios.
- **Scalability**: The separation of responsibilities allows for independent development and scaling of each component.
- **Testability**: Each component can be tested separately, facilitating unit testing and improving overall code quality.
- **Collaboration**: MVC promotes a modular architecture, enabling multiple developers to work on different components concurrently.

Real-World Examples of MVC Usage

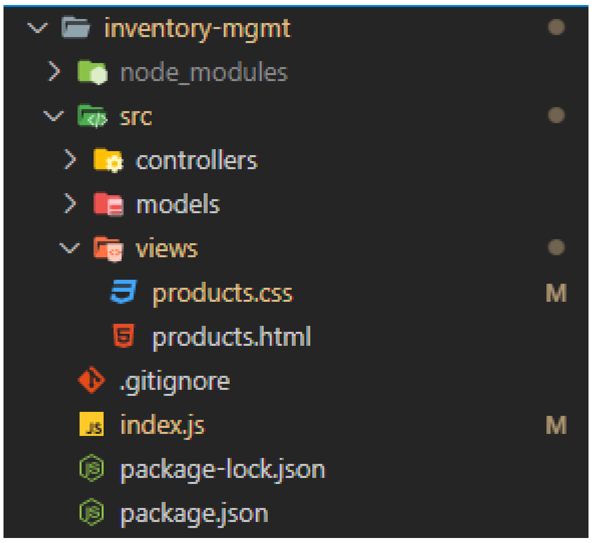
- **E-commerce websites**: MVC helps manage product data, shopping cart functionality, and user interfaces.
- **Social media platforms**: MVC handles user profiles, posts, comments, and interactions.

Creating View

The first step is to create a folder structure for our MVC project. We'll have separate folders for our models, views, and controllers. Here are the steps to create a basic MVC application folder structure:

1. Create a new folder for your project and name **inventory-mgmt**. This will be the root directory of your project.
2. Inside the root directory, create a new file called **index.js**. This will be the entry point for your application and will contain the server code.
3. Inside the root directory, create a new folder called **src**. This will be the main source directory for your project.
4. Inside the src directory, create three new folders called **models**, **views**, and **controllers**. These will be the main components of your MVC architecture.
5. Inside the views directory, create a new file called **products.html**. This will be the products page for your web application.

6. Inside the views directory, create a new file called products.css. This will be the stylesheet for your web application.


```
inventory-mgmt
  node_modules
  src
    controllers
    models
    views
      products.css      M
      products.html
  .gitignore
  index.js              M
  package-lock.json
  package.json
```

7. In index.js, you will need to import and use the Express framework to handle HTTP requests and serve your HTML and CSS files. Here's a basic example of how to do this:

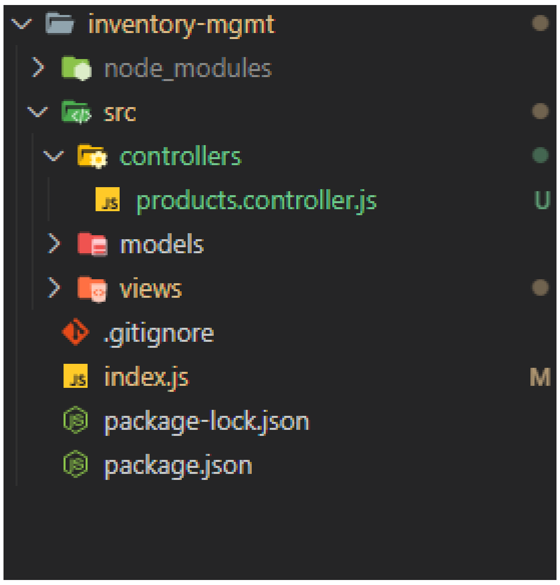
```
import express from 'express'
import path from 'path'

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

server.get('/', (req, res) => {
  res.sendFile(path.join(path.resolve(),"inventory-mgmt" , "src",
"views", "products.html"))
})
server.listen(3400)
```

Here we were sending html file directly, next we will use controller as an intermediate.

Creating Controller

Steps to setup a controller:

1. Create a new file called 'products.controller.js' inside the 'controllers' folder..



```
inventory-mgmt
  node_modules
  src
    controllers
      products.controller.js    U
    models
    views
  .gitignore
  index.js                      M
  package-lock.json
  package.json
```

2. Define a class in the 'products.controller.js' file that sends our 'products.html' file as a response.

   Example code for products.controller.js:

```
import path from 'path'

export default class ProductsController{
    getProducts(req, res){
        res.sendFile(path.join(path.resolve(),"inventory-mgmt",
                                          "src", "views", "products.html"))
      }
}
```

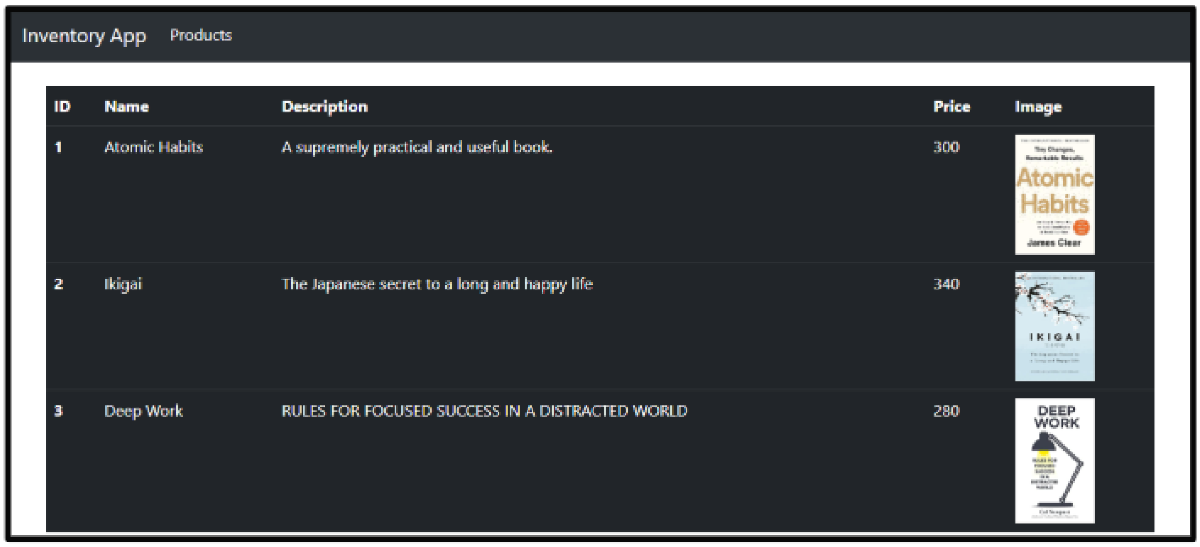
3. In the 'index.js' file, import the 'ProductsController' class.
4. Create an instance of the 'ProductsController' class.
5. Use 'express.static' method to serve the static files present in the 'src/views' folder.
6. Define a route to handle GET request for '/' and pass the callback method 'getProducts' of the 'ProductsController' instance.

7. Listen to a port using 'listen' method of the express instance.

   Example code for index.js:

```
import express from 'express'
import ProductsController from
               './src/controllers/products.controller.js'

const server = express()

// instance of ProductsController
const productController = new ProductsController()

server.use(express.static('src/views'))
server.get('/', productController.getProducts)

server.listen(3400)
```

8. To see the response, open a web browser and navigate to `http://localhost:3400/` in the address bar. This will send a GET request to the server, and the server will respond with the products.html file, which will be displayed in the browser.



ID	Name	Description	Price	Image
1	Atomic Habits	A supremely practical and useful book.	300	
2	Ikigai	The Japanese secret to a long and happy life	340	
3	Deep Work	RULES FOR FOCUSED SUCCESS IN A DISTRACTED WORLD	280	

Creating Model

Models are the components responsible for managing the data in an application. They usually represent real-world objects, like products in our case, and handle interactions with databases, APIs, or any other data source. In short, Models are the heart of your application's data logic.

Creating a Product Model

To create a Product Model, we'll need to define a class that represents the product and its properties.

Here are the steps to create the Product Model:

1. Create a new file called product.model.js in the models folder.
2. Define a ProductModel class with a constructor that takes in 5 parameters: _id, _name, _desc, _price, and _imageUrl. Inside the constructor, set the properties of the class based on the passed-in values.
3. Add a static method called get() to the ProductModel class. Inside this method, return an array of ProductModel objects with some sample data.
4. Export the ProductModel class from the product.model.js file.

Example Code for product.model.js :

```
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 get() {
    return products
  }
}

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',
  ),
  new ProductModel(
    2,
    'Product 2',
    'Description for Product 2',
    29.99,
'https://m.media-amazon.com/images/I/51xwGSNX-EL._SX356_BO1,204,203,200
_.jpg'
  ),
  new ProductModel(
    3,
    'Product 3',
    'Description for Product 3',
    39.99,

'https://m.media-amazon.com/images/I/31PBdo581fL._SX317_BO1,204,203,200
_.jpg'
  )
]
```

Note: To obtain the product details from the ProductModel class, you can import it into your index.js file and utilize the static get() method. Once you have fetched the product data, you may choose to log it to the console or use it in any other way that suits your application's requirements.

Understanding View Engines

View Engines are tools that help us generate HTML content dynamically based on the data provided. They allow us to use templates with placeholders for data, which are then replaced with actual data when the page is rendered. This makes it easier to create and maintain web pages, especially when the content changes frequently.

A real use case

Imagine your online store's product list needs to be updated regularly based on new products or changes in product information. Manually updating the HTML every time there's a change is time-consuming and error-prone. View Engines can help us generate dynamic HTML content, making it easier to keep our web pages up to date.

EJS View Engine for Node.js

EJS, or Embedded JavaScript, is a popular View Engine for Node.js applications. It allows us to embed JavaScript code within HTML templates to generate dynamic content.

Installing EJS and Migrating HTML to EJS

To install EJS, navigate to your project folder in the terminal and run:

```
npm install ejs
```

Changes in index.js file of inventory management project

This code sets up a server using Express and EJS as the view engine to render dynamic content. It imports the ProductController from a file and creates an instance of it to handle requests for products. Here's what each line does:

1. Imports the necessary modules and creates a new Express app.

```
import express from 'express'
import ProductController from
'./src/controllers/product.controller.js'
import path from 'path'
const server = express()
```

2. Sets up the EJS view engine to be used by the app and specifies the directory where the views are located.

```
server.set('view engine', 'ejs')
server.set('views', path.join(path.resolve(), 'src', 'views'))
```

3. Configures the app to use EJS Layouts, which is a middleware that allows the use of templates with multiple views.

```
server.use(ejsLayouts)
```

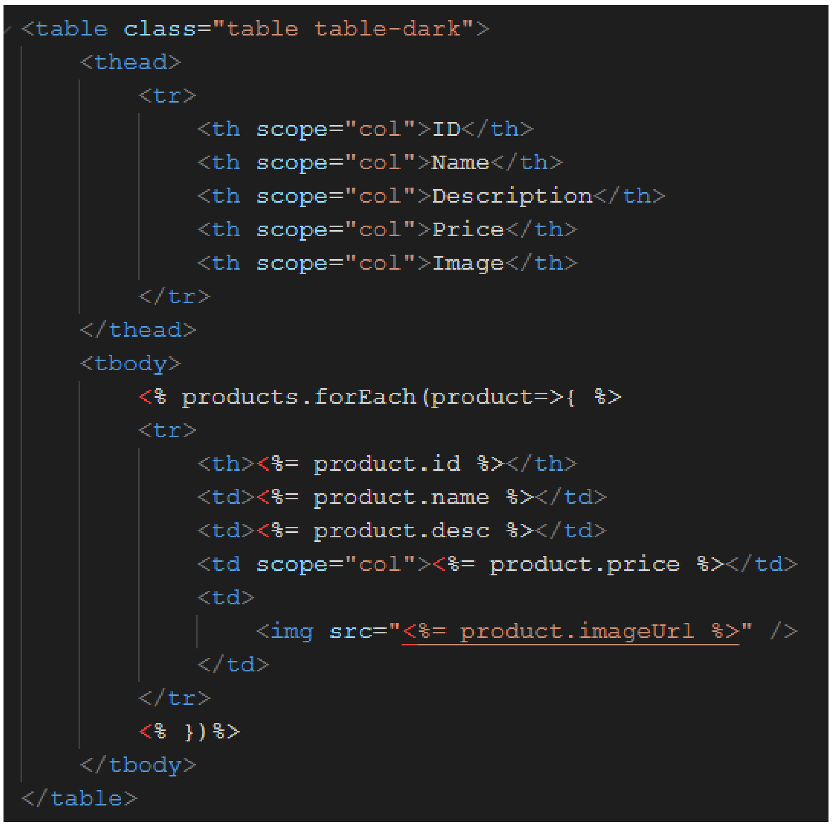
4. Serves the static files from the views directory to the browser.

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

5. Creates an instance of the ProductController and sets up a route to handle GET requests to the root URL. Finally, the server listens on port 3400 for incoming requests.

```
const productController = new ProductController()
server.get('/', productController.getProducts)
server.listen(3400)
```

Changes in products.html file of inventory management project



```html
<table class="table table-dark">
    <thead>
        <tr>
            <th scope="col">ID</th>
            <th scope="col">Name</th>
            <th scope="col">Description</th>
            <th scope="col">Price</th>
            <th scope="col">Image</th>
        </tr>
    </thead>
    <tbody>
        <% products.forEach(product=>{ %>
        <tr>
            <th><%= product.id %></th>
            <td><%= product.name %></td>
            <td><%= product.desc %></td>
            <td scope="col"><%= product.price %></td>
            <td>
                <img src="<%= product.imageUrl %>" />
            </td>
        </tr>
        <% })%>
    </tbody>
</table>
```

This HTML file is an EJS template used to render dynamic product data on the page. The EJS syntax is used to inject data from the server-side JavaScript code into the HTML template.

The syntax uses special tags **<%** and **%>** to enclose JavaScript code within the HTML file. In this particular example, the forEach loop is used to iterate over an array of products and generate table rows with product information. The product object's properties such as id, name, desc, price, and imageUrl are accessed using the **<%= %>** tag within the table cells to render them on the page.

For example, the line **<%= product.id %>** accesses the id property of the current product object within the loop and inserts its value into the table cell. Similarly, other product properties are accessed using <%= %> tags to dynamically generate the HTML content on the page.

Changes in product.controller.js file of inventory management project

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

// Creating ProductController class with getProducts method
export default class ProductController{

  // Defining getProducts method
  getProducts(req, res){

    // Retrieve products using the ProductModel
    let products = ProductModel.get()

    // Use the res.render() method to render the 'products' view, and
pass in the products data
    res.render("products", {products:products})
  }
}
```

Explanation:
This code exports a ProductController class with a getProducts method. Inside the getProducts method, it retrieves the products data from the ProductModel using the ProductModel.get() method. Finally, the res.render() method is used to render the 'products' view using EJS. The res.render() method takes two arguments - the name of the view to render ('products' in this case), and an object containing the data to be passed to the view (the products data in this case). This allows the 'products' view to access the products data and use it to dynamically generate the HTML content.

Creating Layout Page

Using Express-EJS-Layouts Library

In web development, it's common for multiple pages to share a common header, footer, and navigation menu. Instead of duplicating this code in each page, you can create a layout file to keep your code organized and maintain a consistent look and feel throughout your website.

To create a layout file in EJS using Express-EJS-Layouts, you can follow these steps:

1. Install express-ejs-layouts module using npm:
   ```
   npm install express-ejs-layouts
   ```

2. In your index.js file, require the express-ejs-layouts module:
   ```
   const expressLayouts = require('express-ejs-layouts')
   ```

3. Tell your app to use the expressLayouts middleware:
   ```
   app.use(expressLayouts)
   ```

4. Create a new file in your views folder called layout.ejs. This file will contain the common structure for your web pages like navigation menu, header, and footer.

5. Example code:
   ```
   <html>
   <head>
       <title>
           <%= title %>
       </title>
       <!-- Add your stylesheets and scripts here -->
   </head>
   <body>
       <%- body %> <!-- This will be replaced with the
   actual page content -->
   </body>
   </html>
   ```

   Here, the <%- body %> placeholder is used to insert page-specific content into the layout when we render a view. This allows us to maintain a consistent structure across all our pages while still displaying unique content on each page.

Summarising it

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

- Learned about different design patterns like MVC, REST, and Microservices.
- Got an overview of the Inventory Management project.
- Created a View folder and added necessary files for implementing MVC architecture.
- Learned about creating a Controller, which handles incoming requests.
- Learned about creating a Model, which interacts with the database and defines the data structure.
- Learned about View Engine (template engine) and EJS, which is a popular template engine for generating dynamic HTML pages.
- Learned about creating layouts using the express-ejs-layouts library, which helps maintain a consistent structure across all web pages.

Some Additional Resources:

- **What is the MVC, Creating a [Node.js-Express] MVC Application**
- **EJS**
- **How to use EJS to template your Node.js application**