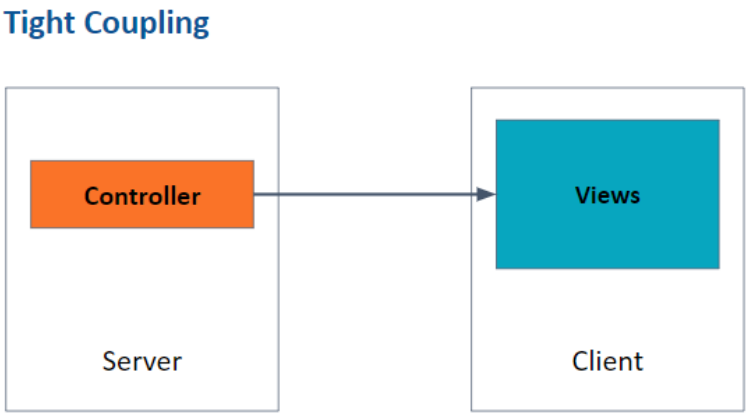
REST API using Express

Understanding API

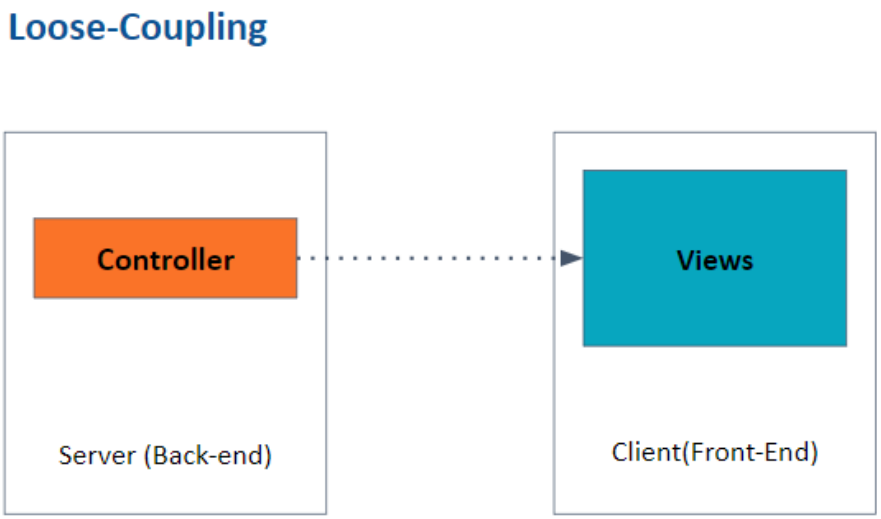
Problems with MVC



- **Complexity**: MVC can become complex as the application grows, making it difficult to maintain and understand the code.
- **Tight coupling**: Components in MVC can be tightly coupled, making it challenging to modify one component without affecting others.
- **Difficulty in making changes**: Due to the complex and tightly coupled nature of MVC, making changes to the application can be difficult.
- **Difficulty in scaling**: As the application grows, scaling an MVC architecture can pose challenges.

API (Application Programming Interface)

APIs are different from MVC in terms of how they handle data. In MVC, the server typically renders views and sends them to the client, which includes both the structure and data to be displayed. On the other hand, APIs primarily focus on sending data instead of views.



- APIs provide a solution to the problems of tightly-coupled systems in MVC applications.
- APIs separate the frontend and backend components, allowing them to communicate through a well-defined interface.
- APIs enable easier modification of individual components without impacting the entire system.
- APIs promote code reusability and facilitate the use of existing services and data from other applications.

Types of APIs

- **SOAP** (Simple Object Access Protocol): A protocol for exchanging structured information in web services. It uses XML and relies on transport protocols like HTTP or SMTP.
- **REST** (Representational State Transfer): An architectural style for designing networked applications. It uses HTTP methods (GET, POST, PUT, DELETE) to perform operations on resources identified by URIs. RESTful APIs are easy to build, maintain, and scale.
- **GraphQL**: A query language and runtime for APIs. It allows clients to request only the data they need, making it more efficient than traditional REST APIs. GraphQL can also aggregate data from multiple sources.

Understanding REST

RESTful API

- **Representational State Transfer** is an architectural style for designing networked applications that use standard HTTP protocols to communicate between the client and server. REST APIs are built on top of the HTTP protocol and work with resources identified by URLs.

- **Stateless architecture** is a key principle of REST, meaning the server doesn't store client-specific states between requests. Each request must contain all the necessary information for the server to process it. The advantages of a stateless architecture include improved scalability, easier maintenance, and better fault tolerance.

- It is an Architectural Guideline
- It is popularly used across different types of systems.

Benefits REST API

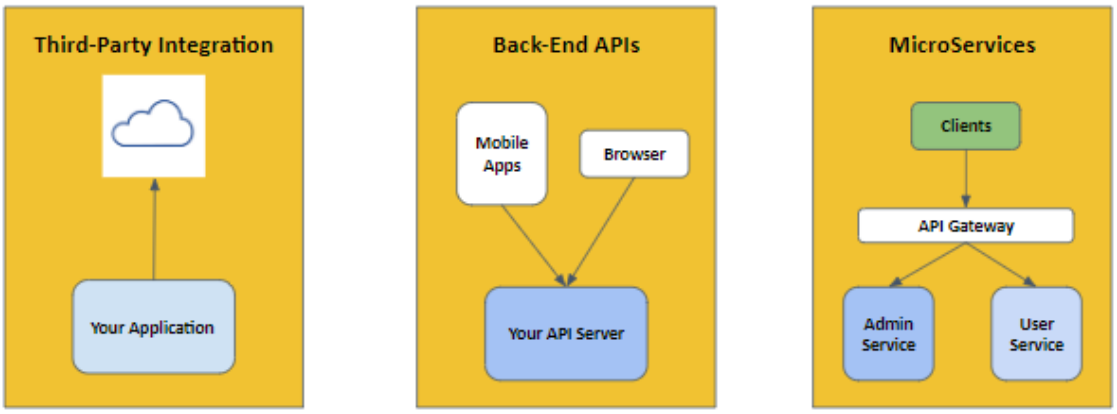
The benefits of using REST APIs include simplicity, scalability, cacheability, interoperability, flexibility, and security.

REST Methods

REST APIs use standard HTTP methods like GET, POST, PUT, and DELETE to perform operations on resources.

- **GET** requests to retrieve information about a resource or a collection of resources.
- **POST** requests create a new resource.
- **PUT** requests to update an existing resource.
- **DELETE** requests remove a resource.

Applications of REST

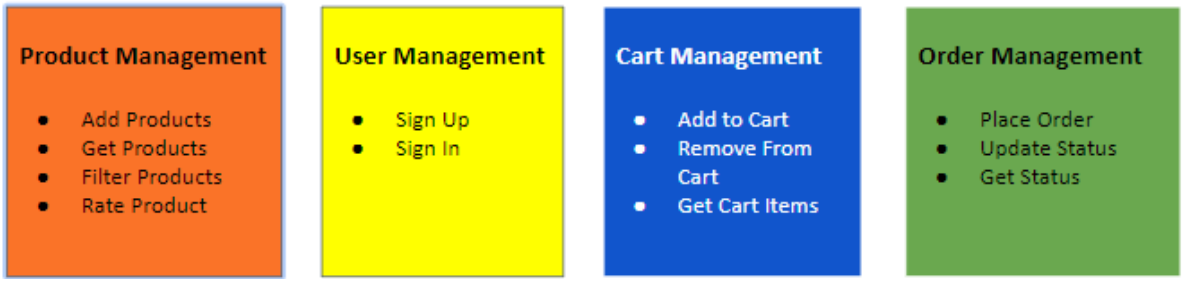


REST APIs play a crucial role in modern web applications, allowing them to efficiently communicate and exchange data with various systems. Some common applications of REST APIs include:

- **Integrating with third-party services**: REST APIs can be used to interact with external services like social media platforms, payment gateways, or analytics tools, adding more functionality to your web application.
- **Creating a consistent backend for multiple platforms**: REST APIs can serve as a common backend for web, mobile, and desktop applications, ensuring that all platforms access the same data and logic.
- **Developing microservices**: REST APIs can be used to break down a monolithic application into smaller, more manageable microservices, improving scalability and maintainability.

Getting Started with API Project

E-Commerce APIs



Creating Folder Structure

Steps to create the folder structure

1. Create a new folder named "**E-COM-API**" for the project.
2. Inside the "E-COM-API" folder, add a file named "**server.js**" to serve as the main server file for the project.

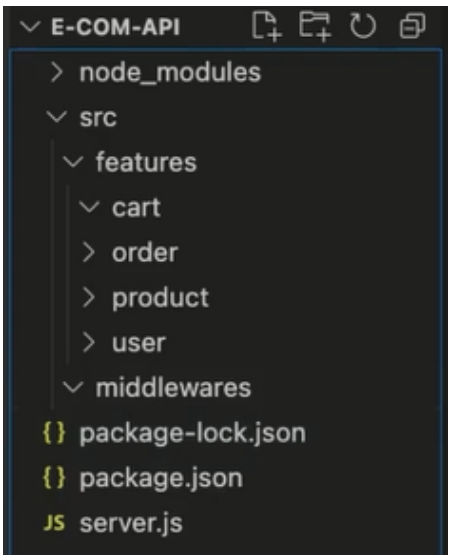
```
// 1. Import Express
import express from 'express'

// 2. Create Server
const server = express();

// 3. Default request handler
server.get("/",(req, res)=>{
    res.send('Welcome to E-commerce APIs');
})

// 4. Specify port
server.listen(3200);
```

3. Create a new folder named "**src**" inside the "E-COM-API" folder to store the source code of the application.
4. Inside the "src" folder, create a folder named “**features**”
5. Inside the "**features**" folder, create separate folders for different modules of the application such as "**cart**", "**order**", "**product**", and "**user**". These folders will contain the respective module-related files.
6. Additionally, create a folder named "**middlewares**" inside the "src" folder to store middleware files that will be used in the application.



Setting Up Routes For Product

1. The goal is to create APIs related to products in the E-COM-API project.

2. The APIs to be created for the product module are:
    - Get all products
    - Add a product
    - Get one product
    - Rate a product
    - Filter Product

3. These APIs will be handled in the product controller.

4. A separate folder for controllers can be created within the product module to manage multiple controllers.

5. The product controller file is created as "product.controller.js" in the product folder.

6. The product roots file is created as "product.roots.js" in the product folder.

7. Express router module is used to handle paths from the server to controller methods.

8. The product roots file manages the paths to the product controller.

9. The path "/api/products" is used as a good practice for API paths.

10. The server file redirects requests related to products to the product roots file.

11. The product roots file specifies the paths and calls the respective controller methods.

12. Other routes for different modules like user and order can be implemented similarly.

13. Separate route files are recommended for each feature to maintain a modular structure.

Code for product.controller.js file:

```
export default class ProductController {
  getAllProducts(request, response) {
    // Code for getting all products
  }

  addProduct(request, response) {
    // Code for adding a product
  }

  rateProduct(request, response) {
    // Code for rating a product
  }

  getOneProduct(request, response) {
    // Code for getting one product
  }
}
```

Code for product.routes.js:

```
import express from 'express';
import ProductController from './product.controller.js';

const router = express.Router();
const productController = new ProductController();
```

```
router.get('/', productController.getAllProducts);
router.post('/', productController.addProduct);
router.get('/:id', productController.getOneProduct);
router.post('/rate', productController.rateProduct);

export default router;
```

Code for server.js

```
import express from 'express';
import productRoutes from './product/product.routes.js;

const app = express();
const PORT = 3000;

app.use('/api/products', productRoutes);

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

Product Model

1. Create the product model class in "product.model.js":

```
export default class ProductModel {
  constructor(ID, name, description, imageURL, category,
price, sizes) {
    this.ID = ID;
    this.name = name;
    this.description = description;
    this.imageURL = imageURL;
    this.category = category;
    this.price = price;
    this.sizes = sizes;
```

```
  }

  static getAll() {
    // Returns all the products
    return [
      new ProductModel(1, "Product 1", "Description 1",
"image1.jpg", 1, 9.99, []),
      new ProductModel(2, "Product 2", "Description 2",
"image2.jpg", 2, 19.99, ["M", "XL"]),
      new ProductModel(3, "Product 3", "Description 3",
"image3.jpg", 3, 29.99, ["S"]),
    ];
  }
}
```

Explanation:

- The product model is created as a class with properties such as ID, name, description, imageURL, category, price, and sizes.
- The constructor initializes these properties when a new product object is created.
- The static method getAll() returns an array of product objects. These objects represent the default products in the system.

2. Implementing the API in the product controller:

```
const Product = require('../models/product.model');

exports.getAllProducts = (req, res) => {
  const products = Product.getSampleProducts();
  res.json(products);
};
```

Explanation:

- The product controller imports the ProductModel class from the product.model.js file.
- The getAllProducts function is a request handler for the route that retrieves all products.
- Inside the function, it calls the static getAll() method of the ProductModel to retrieve the products.
- The retrieved products are sent as the response using res.send().

- The status code 200 (OK) is set to indicate a successful response.

3. Setting up the route in the product router:

```
const express = require('express');
const router = express.Router();
const productController =
require("../controllers/product.controller");

router.get('/', productController.getAllProducts);

router.get('/:id', (req, res) => {
  // Logic to fetch a single product by its ID
});

router.post('/', (req, res) => {
  // Logic to create a new product
});

module.exports = router;
```

Explanation:

- The product router is created using express.Router().
- The router is configured to handle a GET request at the root path ("/") and call the getAllProducts function from the product controller.
- The router is exported to be used in the server file.

4. Importing and using the product router in the server:

```
const express = require("express");
const productRoutes =
require("./src/routes/product.routes");

const server = express();

server.use("/api/products", productRoutes);
server.get("/",(req, res)=>{
    res.send("Server is Ready at 4100");
})

server.listen(4100);
```

```
console.log("Server is listening on 4100");
```

Explanation:

- The product router is imported from the "product.router.js" file.
- The router is used as middleware with the base path '/api/products'.
- When a request is made to the server with the path '/api/products', it will be handled by the product router.
- The server listens on the specified port (3000) and logs a message when it is running.

5. Testing the API:
   Make a GET request to `http://localhost:3000/api/products` in a browser or using a tool like Postman. The response will be an array of product objects as specified in the getAll() function of the product model.

Add Product API

Create an API to return all the products from the server.

Create a new file named product.model.js.
Define the ProductModel class with properties: ID, name, description, imageURL, category, price, and sizes.
Initialize the properties in the constructor of the ProductModel class.
Define some default products with IDs, names, descriptions, prices, image URLs, categories, and sizes.
Create a static function in the ProductModel class called getAll to return all the products.
Import the ProductModel into the product controller.
Call the getAll function of the ProductModel in the controller and assign the returned products to a constant named products.
Send the products data as a response using response.send() in the controller.
Set the status code of the response to 200 for a successful request.
Test the API by running the server and accessing the endpoint /api/products in the browser.

Code for product.model.js file:

```
export default class ProductModel {
```

```
  constructor(ID, name, description, imageURL, category,
price, sizes) {
    this.ID = ID;
    this.name = name;
    this.description = description;
    this.imageURL = imageURL;
    this.category = category;
    this.price = price;
    this.sizes = sizes;
  }

  static getAll() {
    return [
      // Default products
      // ...
    ];
  }
}
```

Code for product.controller.js:

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

// ...

router.get('/', (request, response) => {
  const products = ProductModel.getAll();
  response.status(200).send(products);
});
```

Code for server.js file:

```
import productRouter from './routes/product.router.js';

// ...

app.use('/api/products', productRouter);
```

For Testing:
Access `http://localhost:<port>/api/products` in the browser.
Verify that an array of products is returned in the response body.

Get one Product

- Here we will be using MULTER to upload files in a REST API and retrieving a single product from the server using an API.
- The root parameters are used, which were previously learned in MVC.
- The process involves creating a function in the model to return a single product based on the provided ID.
- In the controller, a function is created to receive the ID from the parameters and call the model's function to get the product.
- If the product is not found, a "product not found" error message with a status code of 404 is returned. Otherwise, the product is sent with a status code of 200.
- The function in the controller is called from the root using the product router's GET method.
- Postman is recommended for testing the API, but it can also be tested from the browser's address bar.
- Guidelines for REST API development are mentioned, including using the correct request methods and response codes.

Model function to return single Product:

```
static getProductById(id) {
  const product = products.find((p) => p.ID === id);
  return product;
}
```

Controller function to handle the GET request for a single product:

```
function getOneProduct(req, res) {
  const { id } = req.params;
  const product = ProductModel.getProductById(id);

  if (!product) {
    return res.status(404).send('Product not found');
  }
```

```
  return res.status(200).send(product);
}
```

Routing the GET request for a single product:

```
productRouter.get('/:id', controller.getOneProduct);
```

Note: The actual implementation may require additional code and configuration depending on the framework or libraries used.

Filter Products

- The goal is to implement an API to filter products in an e-commerce application.
- The filter criteria include minimum price, maximum price, and category.
- The filter function is added to the product model as a static function.
- The filter function uses the JavaScript filter method to filter products based on the given criteria.
- The filter function receives minimum price, maximum price, and category as parameters.
- The filtered products are stored in the 'result' constant and returned.
- The filter function is called from the controller.
- The filter products method is added to the controller to handle the filter API request.
- Query parameters are used to pass the filter criteria from the client to the server.
- The filter API route is created using a GET request and the '/filter' path.
- The filter API route calls the filter products method in the controller.
- The filter criteria are retrieved from the query object using request.query.
- The filter criteria are passed to the filter function in the product model.
- The filtered data is sent back to the client as a response with a status of 200 (OK).
- Users can specify the filter criteria as query parameters in the API URL.

1. Adding the filter function to the product model:

```
// Product model
static filter(minPrice, maxPrice, category) {
  const result = products.filter((product) => {
    return (
      product.price >= minPrice &&
```

```
      product.price <= maxPrice &&
      product.category === category
    );
  });
  return result;
}
```

2. Implementing the filter products method in the controller:

```
// Controller
filterProducts(req, res) {
  const { minPrice, maxPrice, category } = req.query;
  const result = Product.filter(+minPrice, +maxPrice,
category);
  res.status(200).send(result);
}
```

3. Creating the filter API route:

```
// API route
app.get('/products/filter',
productController.filterProducts);
```

Note: The code snippets assume the existence of a product model named "Product" and an array of products named "products". The syntax +minPrice is used to convert the query parameter value from string to a number.

Summarising it

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

1. Explored difficulties encountered with MVC architecture.
2. Acquired knowledge about various types of APIs and their functionalities.
3. Gained insights into REST APIs and their practical uses.
4. Commenced the development of an E-Commerce API project.
5. Established a well-organized folder structure for the project.
6. Familiarized oneself with Express Router and established product routes for the project.

7. Designed the product model for the API project.
8. Successfully incorporated APIs for adding products, retrieving individual products, and filtering products.

Some Additional Resources:

- **Introduction to REST APIs**
- **Express routing**
- **Multer: Easily upload files with Node.js and Express**