**Overview :**

This Application is called **Customer & Product Management System (CPMS)**. It is a Spring Boot application developed to manage both customers and products using RESTful JSON standards. The purpose of CPMS is to provide a simple, yet comprehensive API for CRUD operations—Create, Read, Update, and Delete—related to customer and product information. It ensures efficient data management for an organization, which could be beneficial for retail businesses or service providers looking for an easy way to handle customer data and inventory details. The system uses various technologies to provide robust features like Swagger for API documentation, Redis caching for performance, and GitHub Actions for continuous integration, making it a scalable and well-documented solution.

**Layered Architecture:**

The application is structured with multiple layers, each handling a specific responsibility:

* **Presentation Layer (API Layer)**:
  + This layer is responsible for exposing RESTful endpoints. It uses controllers annotated with @RestController to handle incoming HTTP requests.
  + **Technologies Used**: Spring MVC for request handling, Swagger for API documentation.
* **Service Layer (Business Logic Layer)**:
  + This layer contains the business logic of the application. It processes data received from the controllers before passing it to the data access layer.
  + **Technologies Used**: Plain Java classes with Spring annotations like @Service.
* **Data Access Layer (Persistence Layer)**:
  + This layer interacts with the database to perform CRUD operations. It uses Spring Data JPA repositories to abstract database access logic.
  + **Technologies Used**: Spring Data JPA with @Repository for interacting with SQL databases like H2 database.
* **Data Storage Layer (Database)**:
  + This layer is the database itself, which stores customer and product information.
  + **Technologies Used**: H2 inmemory database.

**Architecture Flow:**

**Presentation Layer <---- RESTful API endpoints using Spring MVC**

**↓**

**Service Layer <---- Business logic, processing data**

**↓**

**Data Access Layer <---- CRUD operations using Spring Data JPA**

**↓**

**Data Storage Layer <---- H2 database**

**Integration Components:**

 **Caching** :

* Redis caching can be added to the architecture to improve performance by caching frequently accessed data like customer and product details.
* **Technologies Used**: Spring Boot Redis integration.

 **Message Broker** :

* Kafka is used for publishing events such as changes in products or customers, which could be consumed by other systems for further processing.
* **Technologies Used**: Spring Kafka for event handling.

Application Flow

 A **client** sends an HTTP request to the **presentation layer** (controller), e.g., to create a new customer.

 The **controller** receives the request and forwards it to the **service layer** after performing input validation.

 The **service layer** processes the business logic, such as verifying if the customer already exists.

 The request is then passed to the **data access layer**, which interacts with the **database** to save or update the customer record.

 Once the operation is successful, the **controller** sends back an appropriate HTTP response.

 For frequently accessed information, the **service layer** may interact with **Redis caching** to reduce database load.

 Any significant event (e.g., new product addition) can be published to **Kafka** to notify other microservice.

Other Components

 **Swagger for API Documentation**: Swagger is integrated to provide easy documentation and testing for all available REST endpoints.

 **Logging with Logback**: Logback is used for maintaining logs for both application activities and API requests/responses.

 **CI/CD with GitHub Actions**: GitHub Actions are used to automate the building, testing, and deployment of the application.

API Demonstration :

### ****1. Overview of Swagger UI****

* **Swagger URL**: Begin by opening the Swagger UI, which provides interactive documentation for the RESTful APIs. Mention that Swagger is being used for documenting and testing each API endpoint.
* Show the endpoints for **customers** and **products**, explaining the available CRUD operations.

### ****2. Demonstration of CRUD Operations****

#### 2.1 **Customers API**

* **POST /customers**: Create a new customer
  + **Demonstrate**: Use the Swagger form to add a customer.
  + **Fields**: First\_Name, Last\_Name, Email (office, personal), Family Members.
  + **Request Body**: Show a JSON request body:

Json Copy code

|  |
| --- |
| {    "firstName": "Kiran 333",    "lastName": "bakka",    "officeEmail": "Kiran@company.com",    "personalEmail": "bakka@gmail.com",    "familyMembers": [      "kiran ",      "mohan kumar",      "Raj kumar"    ]  } |

* + **Response**: Point out the 201 Created response code and display the response, which includes the generated ID for the customer.
* **GET /customers**: Retrieve a list of all customers
  + **Demonstrate**: Use Swagger to make a GET request.
  + **Response**: Display the list of customers and highlight key details.
* **GET /customers/{id}**: Retrieve a specific customer by ID
  + **Demonstrate**: Enter a specific customer ID and show the response with the customer's information.
  + **Point Out**: Mention validation errors if an incorrect ID is entered.
* **PUT /customers/{id}**: Update an existing customer's details
  + **Demonstrate**: Update details like lastName or add a new family member.
  + **Request Body**: Show how updates can be applied selectively:

Json Copy code

|  |
| --- |
| {    "firstName": "Kiran 555",    "lastName": "bakka",    "officeEmail": "Kiran@company.com",    "personalEmail": "bakka@gmail.com",    "familyMembers": [      "kiran ",      "mohan kumar",      "Raj kumar"    ]  } |

* + **Response**: Display the updated customer object.
* **DELETE /customers/{id}**: Delete a customer by ID
  + **Demonstrate**: Perform a DELETE operation.
  + **Response**: Show the 204 No Content status and explain the removal.

#### 2.2 **Products API**

* **POST /products**: Create a new product
  + **Demonstrate**: Add a product with details like Book\_Title, Book\_Price, and Book\_Quantity.
  + **Request Body**:

Json Copy code

|  |
| --- |
| {    "bookTitle": "The Java Handbook\_ttt",    "bookPrice": 29.99,    "bookQuantity": 10  } |

* + **Response**: Display the created product details.
* **GET /products**: Retrieve all products
  + **Demonstrate**: Get a list of products.
  + **Response**: Show the products with their title, price, and quantity.
* **GET /products/{id}**: Retrieve a specific product by ID
  + **Demonstrate**: Show how to get a product by ID.
  + **Response**: Display product details and mention the response format.
* **PUT /products/{id}**: Update product quantity
  + **Demonstrate**: Update the Book\_Quantity of a product.
  + **Request Body**:

Json Copy code

|  |
| --- |
| {    "bookTitle": "The Java updated",    "bookPrice": 29.99,    "bookQuantity": 10  } |

* + **Response**: Show the updated product quantity.
* **DELETE /products/{id}**: Delete a product by ID
  + **Demonstrate**: Remove a product using its ID.
  + **Response**: Show the confirmation of product deletion.

### ****3. Error Handling and Logging Demonstration****

* **Error Scenarios**: Demonstrate a few error scenarios like:
  + **GET /customers/{id}** with a non-existent ID.
  + **POST /customers** with missing mandatory fields.
* **Showcase Logging**: Explain how logging is implemented with Logback and show the console/log file output when these errors occur.

### ****4. Redis Caching (Optional)****

* **Demonstrate Caching**: If Redis caching is implemented, show the following:
  + **GET /customers/{id}** twice for the same customer.
  + **Point Out**: Highlight that the second request fetched data from the cache, improving response time.

### ****5. Kafka Event Publishing (Optional)****

* **Demonstrate Kafka Integration**: If Kafka is integrated, show an event being published when a new product is created.
* **Show Console Output**: Display the event message published to the Kafka topic, using logs or Kafka monitoring tools.

### ****6. Testing Using Postman (Optional)****

* If not using Swagger, you can demonstrate the same requests using **Postman**.
* **Collection Setup**: Show that you have a collection of requests saved for easy access.

**Project structure:**

customer-product-management/

├── src/

│ ├── main/

│ │ ├── java/

│ │ │ └── com.example.cpms/

│ │ │ ├── controller/

│ │ │ │ ├── CustomerController.java

│ │ │ │ └── ProductController.java

│ │ │ ├── service/

│ │ │ │ ├── CustomerService.java

│ │ │ │ └── ProductService.java

│ │ │ ├── repository/

│ │ │ │ ├── CustomerRepository.java

│ │ │ │ └── ProductRepository.java

│ │ │ ├── model/

│ │ │ │ ├── Customer.java

│ │ │ │ └── Product.java

│ │ │ ├── config/

│ │ │ │ ├── KafkaProducerConfig.java

│ │ │ │ └── RedisConfig.java

│ │ │ └── exception/

│ │ │ └── CustomExceptionHandler.java

│ └── resources/

│ ├── application.properties

└── pom.xml

Code Highlights by Layer

Model Layer

Repository Layer

Service Layer

Controller Layer

Exception Handling

Kafka Integration

Redis Caching

Testing and GitHub Actions

@SpringBootTest

public class CustomerServiceTests {

@MockBean

private CustomerRepository customerRepository;

@Autowired

private CustomerService customerService;

@Test

public void testCreateCustomer() {

Customer customer = new Customer();

customer.setFirstName("John");

when(customerRepository.save(any(Customer.class))).thenReturn(customer);

Customer createdCustomer = customerService.createCustomer(customer);

assertEquals("John", createdCustomer.getFirstName());

}

}

**Highlight**:

* + Mocking **CustomerRepository**.
  + Usage of @MockBean and assertions for unit testing.

**GitHub Actions**:

* Show the **CI/CD Workflow** file

**Logging**

For the **Customer & Product Management System (CPMS)**, logging is a crucial part of monitoring, debugging, and auditing application activities. Here's how you can incorporate logging effectively into the project, highlighting best practices for logging at different layers.

Use Logback as the logging framework. Spring Boot comes with Logback by default, so you only need to configure it.

* **logback-spring.xml** file (located in the src/main/resources directory):

**Explanation**:

* **Log Path**: Logs will be written to ./logs/cpms.log.
* **Console Appender**: Logs output to the console.
* **File Appender**: Logs output to a file with a rolling policy (daily).
* **Logger Level**:
  + Set the logger level for com.example.cpms to DEBUG to capture detailed information for your application.
  + The root logger level is set to INFO.

### ****Logging Usage in Application Code****

#### 2.1 Controller Layer

Use logging to track incoming requests, responses, and errors.

* **CustomerController.java**

**Explanation**:

* **Logging Request Details**: Log incoming requests to help with tracking activities.
* **Logging Response Information**: Log successful responses to keep a history of actions taken.

#### 2.2 Service Layer

Log important events such as business logic processing and exceptions.

* **CustomerService.java**

**Explanation**:

* **Debug Logging**: Use logger.debug() to log detailed processing information, helpful during development or when investigating issues.
* **Error Logging**: Log errors at the point where exceptions are raised to capture what went wrong.

#### 2.3 Exception Handling

Log all unhandled exceptions globally for consistent error tracking.

* **CustomExceptionHandler.java**

**Explanation**:

* **Error Handling Logging**: Log specific exceptions like ResourceNotFoundException at the error level.
* **Global Exception Logging**: Use @ExceptionHandler for catching and logging unexpected errors.