Case Study: Product-Order Management System (With Mockito Testing)

Objective

Develop a simple Product-Order system using Spring Boot with MySQL. Test the business logic of services using **Mockito**. No integration testing or H2 database involved.

Functional Requirements

1. Admin can add, view, and update products.
2. Users can place orders for available products.
3. The system reduces stock when an order is placed.
4. Each order stores order details and is linked to the product.

Entity Design

1. `Product`

- `productId` (PK)
- `name`
- `price`
- `availableQuantity`

2. `Order`

- `orderId` (PK)
- `product` (ManyToOne)
- `orderDate`
- `quantityOrdered`

Repository Layer

- `ProductRepository extends JpaRepository<Product, Long>`
- `OrderRepository extends JpaRepository<Order, Long>`

Service Layer

`ProductService`

- `addProduct(Product p)`
- `getAllProducts()`
- `updateStock(Long productId, int qty)`

`OrderService`

- `placeOrder(Long productId, int quantity)`
  - Check if stock is available
  - Create order
  - Reduce product quantity

Controller Layer

`/api/products`

- `POST /` → Add product
- `GET /` → List all products
- `PUT /{id}/stock` → Update stock

`/api/orders`

- `POST /` → Place order
- `GET /` → List all orders

Unit Testing Strategy (Mockito only)

We test only the **service layer** using **Mockito**, without real DB access.

`ProductServiceTest`

- Mock `ProductRepository`
- Test:

- Adding product
- Fetching all products
- Stock update logic

`OrderServiceTest`

- Mock `OrderRepository` and `ProductRepository`
- Test:
  - Order placed successfully when stock is available
  - Order fails if stock is insufficient

Database Setup (MySQL)

In your `application.properties`:

```
spring.datasource.url=jdbc:mysql://localhost:3306/
product_order_db spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=update
```

No need for test profiles or alternate configurations.

Tools & Tech Stack

- Spring Boot 3+
- Spring Data JPA
- MySQL
- JUnit 5
- Mockito

Summary of Benefits

- Clean separation of concerns (MVC + layered architecture)
- Business logic isolated for testing
- Mockito ensures fast, DB-independent testing

- MySQL used consistently in development and testing

CODE :

Entity Layer

Product.java

```
@Entity
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long productId;
    private String name;
    private double price;
    private int availableQuantity;

    // Getters and Setters
}
```

Order.java

```
@Entity
@Table(name = "orders")
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long orderId;

    @ManyToOne
```

```
    @JoinColumn(name = "product_id")
    private Product product;

    private LocalDate orderDate;
    private int quantityOrdered;

    // Getters and Setters
}
```

2. Repository Layer

ProductRepository.java

```
public interface ProductRepository extends JpaRepository<Product, Long> {}
```

OrderRepository.java

```
public interface OrderRepository extends JpaRepository<Order, Long> {}
```

3. Service Layer

ProductService.java

```
@Service
public class ProductService {

    @Autowired
    private ProductRepository productRepo;

    public Product addProduct(Product product) {
        return productRepo.save(product);
    }
```

```
    public List<Product> getAllProducts() {
        return productRepo.findAll();
    }

    public Product updateStock(Long productId, int qty) {
        Product p = productRepo.findById(productId).orElseThrow(() -> new
RuntimeException("Product not found"));
        p.setAvailableQuantity(qty);
        return productRepo.save(p);
    }
}
```

OrderService.java

```
@Service
public class OrderService {

    @Autowired
    private OrderRepository orderRepo;

    @Autowired
    private ProductRepository productRepo;

    public Order placeOrder(Long productId, int quantity) {
        Product product = productRepo.findById(productId).orElseThrow(() -> new
RuntimeException("Product not found"));

        if (product.getAvailableQuantity() < quantity) {
            throw new RuntimeException("Insufficient stock");
```

```
        }

        Order order = new Order();
        order.setProduct(product);
        order.setOrderDate(LocalDate.now());
        order.setQuantityOrdered(quantity);

        // reduce product stock
        product.setAvailableQuantity(product.getAvailableQuantity() - quantity);
        productRepo.save(product);

        return orderRepo.save(order);
    }

    public List<Order> getAllOrders() {
        return orderRepo.findAll();
    }
}
```

Controller Layer

ProductController.java

```
@RestController
@RequestMapping("/api/products")
public class ProductController {

    @Autowired
```

```
    private ProductService productService;

    @PostMapping
    public Product addProduct(@RequestBody Product p) {
        return productService.addProduct(p);
    }

    @GetMapping
    public List<Product> getAll() {
        return productService.getAllProducts();
    }

    @PutMapping("/{id}/stock")
    public Product updateStock(@PathVariable Long id, @RequestParam int qty) {
        return productService.updateStock(id, qty);
    }
}
```

OrderController.java

```
@RestController
@RequestMapping("/api/orders")
public class OrderController {

    @Autowired
    private OrderService orderService;

    @PostMapping
```

```
    public Order placeOrder(@RequestParam Long productId, @RequestParam int quantity)
{
        return orderService.placeOrder(productId, quantity);
    }

    @GetMapping
    public List<Order> getAll() {
        return orderService.getAllOrders();
    }
}
```

5.application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/product_order_db
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

6. Mockito Unit Tests

ProductServiceTest.java

```
@ExtendWith(MockitoExtension.class)
public class ProductServiceTest {

    @Mock
    private ProductRepository repo;

    @InjectMocks
    private ProductService service;
```

```
    @Test
    public void testAddProduct() {
        Product p = new Product();
        p.setName("Laptop");

        Mockito.when(repo.save(p)).thenReturn(p);
        Product saved = service.addProduct(p);

        Assertions.assertEquals("Laptop", saved.getName());
    }

    @Test
    public void testGetAllProducts() {
        Mockito.when(repo.findAll()).thenReturn(List.of(new Product(), new Product()));
        Assertions.assertEquals(2, service.getAllProducts().size());
    }
}
```

OrderServiceTest.java

```
@ExtendWith(MockitoExtension.class)
public class OrderServiceTest {

    @Mock
    private OrderRepository orderRepo;

    @Mock
    private ProductRepository productRepo;
```

```
@InjectMocks
private OrderService orderService;

@Test
public void testPlaceOrder_Success() {
    Product product = new Product();
    product.setProductId(1L);
    product.setAvailableQuantity(10);

    Mockito.when(productRepo.findById(1L)).thenReturn(Optional.of(product));
    Mockito.when(orderRepo.save(Mockito.any())).thenAnswer(i -> i.getArguments()[0]);

    Order order = orderService.placeOrder(1L, 5);
    Assertions.assertEquals(5, order.getQuantityOrdered());
    Assertions.assertEquals(5, product.getAvailableQuantity());
}

@Test
public void testPlaceOrder_Failure() {
    Product product = new Product();
    product.setProductId(1L);
    product.setAvailableQuantity(2);

    Mockito.when(productRepo.findById(1L)).thenReturn(Optional.of(product));
```

```
        Assertions.assertThrows(RuntimeException.class, () -> {
            orderService.placeOrder(1L, 5);
        });
    }
}
```