### **1. Testing the Web Layer: OrderControllerTest**

Use **slice testing** to test the OrderController, which is part of the web layer. The goal is to verify that the REST endpoints handle HTTP requests correctly. Use @WebMvcTest and @MockBean to isolate the controller from its dependencies.

#### **Step-by-step Execution**

1. **Set Up the Test Class**: Create a test class and annotate it with @WebMvcTest(OrderController.class). This tells Spring to load only the necessary web components and the specified controller.
2. **Mock Dependencies**: Use @MockBean to create a mock of the OrderService. This mock will replace the real OrderService in the application context during the test, ensuring that the test does not depend on the business logic layer.
3. **Inject MockMvc**: Use @Autowired to inject MockMvc. This is a powerful Spring utility that allows you to perform mock HTTP requests (e.g., POST, GET) without a real server running.
4. **Define Mock Behavior**: Use Mockito.when() to define the behavior of the mocked OrderService. We tell it to return a pre-defined Order object whenever the createOrder method is called.
5. **Perform the Test**: Use MockMvc.perform() to simulate an HTTP POST request to the /orders endpoint. We specify the request body and content type.
6. **Verify the Result**: Use andExpect() to verify the HTTP response. We check for a 200 OK status and use jsonPath() to assert that the JSON response body contains the expected values, such as the "CONFIRMED" status.
7. **Verify Mock Interaction**: Use Mockito.verify() to confirm that the createOrder method on our mock OrderService was called exactly once. This ensures that the controller correctly delegates the request to the service layer.

### **2. Testing the Persistence Layer: InventoryRepositoryTest**

Here, test the InventoryRepository, which handles database operations. Use @DataJpaTest to set up a fast, in-memory database for testing.

#### **Step-by-step Execution**

1. **Set Up the Test Class**: Annotate the class with @DataJpaTest. This will auto-configure an in-memory database and load only the JPA components and our repository bean.
2. **Inject TestEntityManager**: Inject TestEntityManager. This Spring utility allows us to insert and retrieve data directly from the in-memory database for setting up test data, bypassing our repository.
3. **Inject the Repository**: Use @Autowired to inject the InventoryRepository that we want to test.
4. **Arrange**: Use TestEntityManager.persist() to insert a test InventoryItem record into the in-memory database.
5. **Act**: Call the repository method under test, inventoryRepository.findByName("Laptop").
6. **Assert**: Use Assertions.assertThat() to verify the result. Check that an item was found (isPresent()) and that its properties match the expected values.

### **3. Testing the Service Layer: OrderServiceIntegrationTest**

This is an **integration test** for the OrderService. The service depends on an external PaymentService API, which we'll mock using **WireMock**. This tests the service's logic without relying on a real, potentially slow, or unreliable external service.

#### **Step-by-step Execution**

1. **Set Up the Test Class**: Annotate the class with @SpringBootTest to load the full application context. Also, use @AutoConfigureWireMock to set up a mock server.
2. **Define Mock Server Behavior**: Use stubFor() from WireMock to define how the mock server should respond to a specific request. Tell it that when a POST request is sent to /payments/process, it should return a 200 OK status with a success JSON body.
3. **Inject the Service**: Use @Autowired to inject the OrderService to test. This will be the real service bean.
4. **Act**: Call the method under test, orderService.processPayment(order). This method will internally make an HTTP call, which WireMock will intercept.
5. **Assert**: Use Assertions.assertThat() to verify that the service's logic worked correctly based on the mock server's response.
6. **Verify Mock Interaction**: Use verify() from WireMock to ensure that the service actually made the expected HTTP call to the mock server.