**Miscellaneous/ Scenario Based QA**

1. **How will you test API CRUD operations without using Postman or SOAP UI?**

**Ans** - If I need to test Rest APIs without using tools like Postman or SoapUI, I’d use Spring Boot’s built-in HTTP clients. For blocking calls, I’d use **RestTemplate (Refer Page 5)**, and for modern non-blocking testing, I’d use **WebClient (Refer Page 7)**

Note- RestTemplate is deprecated in favor of WebClient for new Spring applications, but it's still supported and commonly used.

**SOURCE CODE PRE-REQ. (Q1 to Q3)**

Product Model:

|  |
| --- |
| package com.example.productmanagement.model;  import jakarta.persistence.Entity;  import jakarta.persistence.GeneratedValue;  import jakarta.persistence.GenerationType;  import jakarta.persistence.Id;  import java.math.BigDecimal;  @Entity  public class Product {      @Id      @GeneratedValue(strategy = GenerationType.IDENTITY)      private Long id;      private String name;      private BigDecimal price;      public Product() {}      public Product(String name, BigDecimal price) {          this.name = name;          this.price = price;      }      // Getters and Setters      public Long getId() { return id; }      public void setId(Long id) { this.id = id; }      public String getName() { return name; }      public void setName(String name) { this.name = name; }      public BigDecimal getPrice() { return price; }      public void setPrice(BigDecimal price) { this.price = price; }      @Override      public String toString() {          return "Product{" +                 "id=" + id +                 ", name='" + name + '\'' +                 ", price=" + price +                 '}';      }  } |

Product Repository:

|  |
| --- |
| package com.example.productmanagement.repository;  import com.example.productmanagement.model.Product;  import org.springframework.data.jpa.repository.JpaRepository;  import org.springframework.stereotype.Repository;  @Repository  public interface ProductRepository extends JpaRepository<Product, Long> {  } |

Product Controller:

|  |
| --- |
| package com.example.productmanagement.controller;  import com.example.productmanagement.model.Product;  import com.example.productmanagement.repository.ProductRepository;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.http.HttpStatus;  import org.springframework.http.ResponseEntity;  import org.springframework.web.bind.annotation.\*;  import java.util.List;  import java.util.Optional;  @RestController  @RequestMapping("/api/products")  public class ProductController {      @Autowired      private ProductRepository productRepository;  @Autowired      private ProductService productService;      @PostMapping      public ResponseEntity<Product> createProduct(@RequestBody Product product) {          Product newProduct = productRepository.save(product);          return new ResponseEntity<> (n ewProduct, HttpStatus.CREATED);      }  @GetMapping      public ResponseEntity<List<Product>> getAllProducts() {          List<Product> products = productRepository.findAll();          return new ResponseEntity<> (products, HttpStatus.OK);      }      @GetMapping("/{id}")      public ResponseEntity<Product> getProductById(@PathVariable Long id) {          Optional<Product> product = productRepository.findById(id);          return product.map(value -> new ResponseEntity<> (value, HttpStatus.OK))                        .orElseGet(() -> new ResponseEntity<> (HttpStatus.NOT\_FOUND));      }      @PutMapping("/{id}")      public ResponseEntity<Product> updateProduct(@PathVariable Long id, @RequestBody Product productDetails) {          Optional<Product> productOptional = productRepository.findById(id);          if (productOptional.isPresent()) {              Product existingProduct = productOptional.get();              existingProduct.setName(productDetails.getName());              existingProduct.setPrice(productDetails.getPrice());              Product updatedProduct = productRepository.save(existingProduct);              return new ResponseEntity<> (updatedProduct, HttpStatus.OK);          } else {              return new ResponseEntity<> (HttpStatus.NOT\_FOUND);          }      }        @DeleteMapping("/{id}")      public ResponseEntity<Void> deleteProduct (@PathVariable Long id) {          if (productRepository.existsById(id)) {              productRepository.deleteById(id);              return new ResponseEntity<> (HttpStatus.NO\_CONTENT);          } else {              return new ResponseEntity<> (HttpStatus.NOT\_FOUND);          }      }  @PostMapping("/testCRUD")  public ResponseEntity<Void> testCRUD()  {  try {  productService. runCrudOperations();  return new ResponseEntity<> (HttpStatus.OK);  } catch (Exception ex)  return new ResponseEntity<> (HttpStatus.INTERNAL\_SERVER\_ERROR);  }  }  } |
|  |

Pom.xml

|  |
| --- |
| <!-- Required for building a REST API and for RestTemplate -->          <dependency>              <groupId>org.springframework.boot</groupId>              <artifactId>spring-boot-starter-web</artifactId>          </dependency>          <!-- Required for WebClient -->          <dependency>              <groupId>org.springframework.boot</groupId>              <artifactId>spring-boot-starter-webflux</artifactId>          </dependency> |

**Approach 1: RestTemplate Client**

RestTemplate is a synchronous, blocking HTTP client. It's a classic Spring tool that is still widely used but is now in maintenance mode.

* Characteristics of RestTemplate:
* Synchronous / Blocking – waits for each request to complete before moving on.
* Simple and easy for quick testing and small applications.

RestTemplate Product Service:

|  |
| --- |
| package com.example.productmanagement.service;  import com.example.productmanagement.model.Product;  import org.springframework.http.\*;  import org.springframework.stereotype.Service;  import org.springframework.web.client.RestTemplate;  import java.math.BigDecimal;  import java.util.Arrays;  import java.util.List;  @Service  public class ProductService {      private final String BASE\_URL = "http://localhost:8080/api/products";      private final RestTemplate restTemplate = new RestTemplate();      public void runCrudOperations() {          System.out.println("--- Starting RestTemplate CRUD Operations ---");  // Create Product  Product newProduct = createProduct(new Product("Smartphone", new BigDecimal("799.99")));          System.out.println("Created Product: " + newProduct);          //Find All Products          List<Product> allProducts = getAllProducts();          System.out.println("All Products: " + allProducts);          //Find Product By ID          Product foundProduct = getProductById(newProduct.getId());          System.out.println("Product by ID: " + foundProduct);            foundProduct.setPrice(new BigDecimal("749.99"));  //Update Product          updateProduct(foundProduct.getId(), foundProduct);          Product updatedProduct = getProductById(foundProduct.getId());          System.out.println("Updated Product: " + updatedProduct);          // Delete Product          deleteProduct(updatedProduct.getId());          System.out.println("Deleted Product with ID: " + updatedProduct.getId());          System.out.println("--- Finished RestTemplate CRUD Operations ---");      }        public Product createProduct(Product product) {          ResponseEntity<Product> response = restTemplate.postForEntity(BASE\_URL, product, Product.class);          return response.getBody();      }        public List<Product> getAllProducts() {          ResponseEntity<Product[]> response = restTemplate.getForEntity(BASE\_URL, Product[].class);          return Arrays.asList(response.getBody());      }        public Product getProductById(Long id) {          ResponseEntity<Product> response = restTemplate.getForEntity(BASE\_URL + "/" + id, Product.class);          return response.getBody();      }      public void updateProduct(Long id, Product product) {          restTemplate.put(BASE\_URL + "/" + id, product);      }        public void deleteProduct(Long id) {          restTemplate.delete(BASE\_URL + "/" + id);      }  } |

**Approach 2: WebClient Client**

WebClient is a asynchronous, non-blocking HTTP client that is the modern replacement for RestTemplate. It provides a fluent, functional API.

Characteristics of WebClient:

* Asynchronous & Non-Blocking – ideal for high-concurrency apps.
* Modern and recommended by Spring
* Requires block() for sync behavior, which we use here just for testing

WebClient Product Service:

|  |
| --- |
| package com.example.productmanagement.service;  import com.example.productmanagement.model.Product;  import org.springframework.http.HttpStatus;  import org.springframework.http.MediaType;  import org.springframework.stereotype.Service;  import org.springframework.web.reactive.function.client.WebClient;  import reactor.core.publisher.Mono;  import java.math.BigDecimal;  import java.util.List;  @Service  public class ProductServiceWebClient {      private final String BASE\_URL = "http://localhost:8080/api/products";      private final WebClient webClient = WebClient.create();      public void runCrudOperations() {          System.out.println("--- Starting WebClient CRUD Operations ---");          // CREATE          Product newProduct = createProduct(new Product("Laptop", new BigDecimal("1200.00")));          System.out.println("CREATED Product: " + newProduct);          // READ (all)          List<Product> allProducts = getAllProducts();          System.out.println("READ all Products: " + allProducts);          // READ (by ID)          Product foundProduct = getProductById(newProduct.getId());          System.out.println("READ Product by ID: " + foundProduct);          // UPDATE          foundProduct.setPrice(new BigDecimal("1150.00"));          updateProduct(foundProduct.getId(), foundProduct);          Product updatedProduct = getProductById(foundProduct.getId());          System.out.println("UPDATED Product: " + updatedProduct);          // DELETE          deleteProduct(updatedProduct.getId());          System.out.println("DELETED Product with ID: " + updatedProduct.getId());          System.out.println("--- Finished WebClient CRUD Operations ---");      }      // Create Product      public Product createProduct(Product product) {          return webClient.post()                          .uri(BASE\_URL)                          .contentType(MediaType.APPLICATION\_JSON)                          .bodyValue(product)                          .retrieve()                          .bodyToMono(Product.class)                          .block(); // block() for synchronous execution for this example      }      // Fetch All Products      public List<Product> getAllProducts() {          return webClient.get()                          .uri(BASE\_URL)                          .retrieve()                          .bodyToFlux(Product.class)                          .collectList()                          .block();      }      // Fetch Product By ID      public Product getProductById(Long id) {          return webClient.get()                          .uri(BASE\_URL + "/" + id)                          .retrieve()                          .bodyToMono(Product.class)                          .block();      }      // Update Product      public void updateProduct(Long id, Product product) {          webClient.put()                   .uri(BASE\_URL + "/" + id)                   .contentType(MediaType.APPLICATION\_JSON)                   .body(Mono.just(product), Product.class)                   .retrieve()                   .toBodilessEntity()                   .block();      }      // Delete Product      public void deleteProduct(Long id) {          webClient.delete()                   .uri(BASE\_URL + "/" + id)                   .retrieve()                   .toBodilessEntity()                   .block();      }  } |

1. **How would you test all the endpoints on application start-up?**

**Ans**- I’d use CommandLineRunner. It’s a functional interface provided by Spring Boot. It is used to execute code after the Spring Boot application has started — typically for startup logic, testing, or data initialization.

**Approach 1: RestTemplate**

|  |
| --- |
| @Component  public class ProductApiTester implements CommandLineRunner {  private final String BASE\_URL = "http://localhost:8080/api/products";  private final RestTemplate restTemplate = new RestTemplate();  @Override  public void run(String... args) throws Exception {  // 1. Create Product  Product newProduct = new Product(null, "Laptop", 1200.0);  ResponseEntity<Product> createResponse = restTemplate.postForEntity(BASE\_URL, newProduct, Product.class);  Product createdProduct = createResponse.getBody();  // 2. Get All Products  ResponseEntity<Product[]> getAll = restTemplate.getForEntity(BASE\_URL, Product[].class);  // 3. Get Product by ID  Product productById = restTemplate.getForObject(BASE\_URL + "/" + createdProduct.getId(), Product.class);  // 4. Update Product  productById.setPrice(1300.0);  restTemplate.put(BASE\_URL + "/" + createdProduct.getId(), productById);  // 5. Delete Product  restTemplate.delete(BASE\_URL + "/" + createdProduct.getId());  }  } |

**Approach 2: WebClient**

|  |
| --- |
| @Component  public class ProductWebClientTester implements CommandLineRunner {  private final WebClient webClient = WebClient.create("http://localhost:8080/api");  @Override  public void run(String... args) throws Exception {  // Create Product  Product newProduct = new Product(null, "Smartphone", 800.0);  Product createdProduct = webClient.post()  .uri("/products")  .body(Mono.just(newProduct), Product.class)  .retrieve()  .bodyToMono(Product.class)  .block(); // block() to make it sync for testing  // Get Product By ID  Product fetched = webClient.get()  .uri("/products/{id}", createdProduct.getId())  .retrieve()  .bodyToMono(Product.class)  .block();  // Get All Products      public List<Product> getAllProducts() {          return webClient.get()                          .uri("/products")                          .retrieve()                          .bodyToFlux(Product.class)                          .collectList()                          .block();      }  // Update Product  fetched.setPrice(850.0);  webClient.put()  .uri("/products/{id}", createdProduct.getId())  .body(Mono.just(fetched), Product.class)  .retrieve()  .bodyToMono(Void.class)  .block();  // Delete Product  webClient.delete()  .uri("/products/{id}", createdProduct.getId())  .retrieve()  .bodyToMono(Void.class)  .block();  }  } |

FYI –

**Flow of Execution**

|  |
| --- |
| 1. Application Context Initialization: The Spring Boot application starts, and the Spring application context is loaded. All beans, including your WebClient bean and the class implementing CommandLineRunner, are created and initialized. 2. CommandLineRunner Execution: Once the application context is fully loaded, Spring Boot automatically detects and invokes the run() method of all beans that implement the CommandLineRunner interface. 3. Dependency Injection: Inside the run() method, the WebClient instance (which was created as a @Bean) is injected. This allows you to use the WebClient to make HTTP calls. 4. REST API Call: You use the injected WebClient to perform your REST API test. This typically involves a chain of reactive operations:    * webClient.method(): Specify the HTTP method (e.g., get(), post()).    * .uri(): Define the endpoint URI.    * .retrieve() or .exchange(): Trigger the request.    * .bodyToMono() or .bodyToFlux(): Convert the response body into a reactive Mono (for a single object) or Flux (for multiple objects).    * .block(): Since CommandLineRunner is a synchronous operation, you'll need to call .block() to wait for the reactive stream to complete and get the result. This is acceptable for a one-off startup task like testing. 5. Test and Log Results: After the block() call returns the response object, you can perform assertions or simply log the results to the console. This is where you would check the response status, payload, or any other details to verify the API's behavior. 6. Application Shutdown: Once the run() method completes, the application may shut down, as the CommandLineRunner has finished its purpose. |

1. **How will you test API CRUD operations with Java (without using Spring or any external tool)?**

I’d use HttpClient or HttpURLConnection which are a core part of the standard Java library for handling HTTP requests.

**Approach 1: HttpClient (introduced in Java 11)**

|  |
| --- |
| import java.io.IOException;  import java.net.URI;  import java.net.http.HttpClient;  import java.net.http.HttpRequest;  import java.net.http.HttpResponse;  import java.net.http.HttpRequest.BodyPublishers;  import java.util.concurrent.ExecutionException;  public class ApiTestClient {  private static final String BASE\_URL = "http://localhost:8080/api/products";  private static final String JSON\_TYPE = "application/json";  private static final HttpClient HTTP\_CLIENT = HttpClient.newHttpClient();  public static void main(String[] args) throws IOException, InterruptedException, ExecutionException {  System.out.println("--- Starting API CRUD Test with Java HttpClient ---");  // Create a new product  String createJson = "{\"name\":\"Monitor\",\"price\":300.50}";  HttpResponse<String> createResponse = sendPostRequest(createJson);  System.out.println("POST request responded with status code: " + createResponse.statusCode());  System.out.println("Response body: " + createResponse.body());  long productId = 1; // Assuming the first product gets ID 1  // Get the created product (GET)  HttpResponse<String> getResponse = sendGetRequest(productId);  System.out.println("GET request responded with status code: " + getResponse.statusCode());  System.out.println("GET request response body: " + getResponse.body());  // Update the product  String updateJson = "{\"id\":" + productId + ",\"name\":\"4K Gaming Monitor\",\"price\":450.75}";  HttpResponse<Void> updateResponse = sendPutRequest(productId, updateJson);  System.out.println("PUT request responded with status code: " + updateResponse.statusCode());  //Delete the product  HttpResponse<Void> deleteResponse = sendDeleteRequest(productId);  System.out.println("DELETE request responded with status code: " + deleteResponse.statusCode());  }  private static HttpResponse<String> sendPostRequest(String jsonPayload) throws IOException, InterruptedException {  HttpRequest request = HttpRequest.newBuilder()  .uri(URI.create(BASE\_URL))  .header("Content-Type", JSON\_TYPE)  .POST(BodyPublishers.ofString(jsonPayload))  .build();  return HTTP\_CLIENT.send(request, HttpResponse.BodyHandlers.ofString());  }  private static HttpResponse<String> sendGetRequest(long id) throws IOException, InterruptedException {  HttpRequest request = HttpRequest.newBuilder()  .uri(URI.create(BASE\_URL + "/" + id))  .GET()  .build();  return HTTP\_CLIENT.send(request, HttpResponse.BodyHandlers.ofString());  }  private static HttpResponse<Void> sendPutRequest(long id, String jsonPayload) throws IOException, InterruptedException {  HttpRequest request = HttpRequest.newBuilder()  .uri(URI.create(BASE\_URL + "/" + id))  .header("Content-Type", JSON\_TYPE)  .PUT(BodyPublishers.ofString(jsonPayload))  .build();  return HTTP\_CLIENT.send(request, HttpResponse.BodyHandlers.discarding());  }  private static HttpResponse<Void> sendDeleteRequest(long id) throws IOException, InterruptedException {  HttpRequest request = HttpRequest.newBuilder()  .uri(URI.create(BASE\_URL + "/" + id))  .DELETE()  .build();  return HTTP\_CLIENT.send(request, HttpResponse.BodyHandlers.discarding());  }  } |

FYI -

* HttpClient: A class in Java used to send HTTP requests and receive responses. It is the main entry point for making HTTP calls.
* HttpRequest: Represents an HTTP request. You use this class to define the request method (GET, POST, etc.), URI, headers, and body.
* HttpResponse: Represents the response returned from an HTTP request. It contains status code, headers, and the response body.
* BodyPublisher: A class that converts high-level Java objects (like a String, a file, or a byte array) into a stream of byte buffers suitable for sending as an HTTP request body.

**Approach 2: Using HttpURLConnection (Old Approach – Introduced in Java 1.1) (Least Imp. – Just For Knowledge)**

|  |
| --- |
| import java.io.BufferedReader;  import java.io.InputStreamReader;  import java.io.OutputStream;  import java.net.HttpURLConnection;  import java.net.URL;  import java.nio.charset.StandardCharsets;  public class ApiTestClient {  private static final String BASE\_URL = "http://localhost:8080/api/products";  private static final String JSON\_TYPE = "application/json";  public static void main(String[] args) throws Exception {  System.out.println("--- Starting API CRUD Test with Plain Java (HttpURLConnection) ---");  String createJson = "{\"name\":\"Monitor\",\"price\":300.50}";  int createResponseCode = sendPostRequest(createJson);  System.out.println("POST request responded with status code: " + createResponseCode);  long productId = 1;  String getResponse = sendGetRequest(productId);  System.out.println("GET request response body: " + getResponse);  String updateJson = "{\"id\":" + productId + ",\"name\":\"4K Gaming Monitor\",\"price\":450.75}";  int updateResponseCode = sendPutRequest(productId, updateJson);  System.out.println("PUT request responded with status code: " + updateResponseCode);  String getUpdatedResponse = sendGetRequest(productId);  System.out.println("GET request response body after update: " + getUpdatedResponse);  int deleteResponseCode = sendDeleteRequest(productId);  System.out.println("DELETE request responded with status code: " + deleteResponseCode);  String getDeletedResponse = sendGetRequest(productId);  System.out.println("GET request response body after delete (should be empty): " + getDeletedResponse);  System.out.println("--- API CRUD Test Complete ---");  }  private static int sendPostRequest(String jsonPayload) throws Exception {  URL url = new URL(BASE\_URL);  HttpURLConnection conn = (HttpURLConnection) url.openConnection();  conn.setRequestMethod("POST");  conn.setRequestProperty("Content-Type", JSON\_TYPE);  conn.setDoOutput(true);  try (OutputStream os = conn.getOutputStream()) {  byte[] input = jsonPayload.getBytes(StandardCharsets.UTF\_8);  os.write(input, 0, input.length);  }  return conn.getResponseCode();  }  private static String sendGetRequest(long id) throws Exception {  URL url = new URL(BASE\_URL + "/" + id);  HttpURLConnection conn = (HttpURLConnection) url.openConnection();  conn.setRequestMethod("GET");    int responseCode = conn.getResponseCode();  StringBuilder content = new StringBuilder();  if (responseCode == HttpURLConnection.HTTP\_OK) {  try (BufferedReader br = new BufferedReader(new InputStreamReader(conn.getInputStream(), StandardCharsets.UTF\_8))) {  String line;  while ((line = br.readLine()) != null) {  content.append(line);  content.append(System.lineSeparator());  }  }  } else {  // Read error stream for non-200 responses  try (BufferedReader br = new BufferedReader(new InputStreamReader(conn.getErrorStream(), StandardCharsets.UTF\_8))) {  String line;  while ((line = br.readLine()) != null) {  content.append(line);  content.append(System.lineSeparator());  }  }  return "Error (" + responseCode + "): " + content.toString();  }  return content.toString();  }  private static int sendPutRequest(long id, String jsonPayload) throws Exception {  URL url = new URL(BASE\_URL + "/" + id);  HttpURLConnection conn = (HttpURLConnection) url.openConnection();  conn.setRequestMethod("PUT");  conn.setRequestProperty("Content-Type", JSON\_TYPE);  conn.setDoOutput(true);  try (OutputStream os = conn.getOutputStream()) {  byte[] input = jsonPayload.getBytes(StandardCharsets.UTF\_8);  os.write(input, 0, input.length);  }  return conn.getResponseCode();  }  private static int sendDeleteRequest(long id) throws Exception {  URL url = new URL(BASE\_URL + "/" + id);  HttpURLConnection conn = (HttpURLConnection) url.openConnection();  conn.setRequestMethod("DELETE");  return conn.getResponseCode();  } |

1. **How do two micro services communicate with each other? OR**

**How do you handle inter-service communication in a microservices architecture?**

In microservices architecture, two microservices can communicate with each other via synchronous HTTP calls using RestTemplate, HttpClient or WebClient, or via messaging using tools like RabbitMQ or Apache Kafka.

~ RestTemplate, WebClient & HttpClient:

For each of these, refer to Q1 and Q2. Simply mention the base URL of the targeted microservice along with the required endpoints for POST, GET, PUT, and DELETE methods that the current microservice needs to communicate with.

**~ Apache Kafka:**

**A blue square with a yellow tube

AI-generated content may be incorrect.**

**Reference -** [**Microservices Communication with Apache Kafka in Spring Boot - GeeksforGeeks**](https://www.geeksforgeeks.org/advance-java/microservices-communication-with-apache-kafka-in-spring-boot/)<https://www.geeksforgeeks.org/advance-java/microservices-communication-with-apache-kafka-in-spring-boot/>

How it Works

Let's use your example of a product service and an order service to illustrate this.

1. Producers and Consumers: In Kafka, services that send messages are called producers, and services that read messages are called consumers. The producer and consumer services are unaware of each other's existence. They only know about the Kafka topics.
2. Topics: A Kafka topic is a category or feed name where events are published. Think of it as a logical channel. For instance, you could have a new\_orders topic.
3. The Process:
   * The order service acts as a producer. When a new order is placed, the order service publishes an "order created" event to a Kafka topic, let's say order\_events. This event contains details about the order, like the order ID, customer ID, and the products purchased.
   * The product service acts as a consumer. It subscribes to the order\_events topic. When a new message (the "order created" event) arrives, the product service consumes it and can then take action, such as updating the inventory for the products that were just ordered.
4. **How do you consume an external API in Spring Boot?**

Using Spring WebClient or RestTemplate

1. **How do you implement exception handling in a Spring Boot REST API?   
   (**Follow up ~ Qs - How do you return proper HTTP status codes and error messages?

How do you handle global exceptions?)