\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Project: OBJECT/JSON/XML conversion in all formats**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |
| --- |
| 1. **XML to Object 🡪**   a. JAXB(Unmarshaller),  b. Jackson(XMlMapper— readvalue(xml,person.class))  c. Spring OXM(Object/XMlMapping)🡪Jaxb2Marshaller->setClasstoBeBound(Person.class)  d. Using XStreamXStream->fromXML(xmlString)  e. Using DOM/SAX/Stax parser(low level, Mannual)-DocumentBuilderFactory,Document  f. Using RestTemplate/WebClient with XML Message convertor🡪getForObject   1. **Object to xml mapping🡪**   a. JAXB(Marshaller),  b. Jackson(XMlMapper— writeValuesAsString())  c. Spring OXM(Object/XMlMapping)🡪Jaxb2Marshaller.unmarshal(newe StreamSpurce(new StroinReader(xml))  d. Using XStreamXStream->toXml(xmlString)   1. **Springboot auto conversion🡪** |

**------------------------------------------------------------------------------------------------------------------------**

1. **Using JAXB (Java Architecture for XML Binding)—Marshalling/Unmarshalling**

Spring boot automatically supports JAXB if you add the right dependency.

**Step 1: Add dependency (POM.XML)**

<!-- https://mvnrepository.com/artifact/javax.xml.bind/jaxb-api -->

<dependency>

<groupId>javax.xml.bind</groupId>

<artifactId>jaxb-api</artifactId>

<version>2.3.1</version>

</dependency>

**Step 2: create a model class:**

**@XmlRootElement(name=”employee”)**

**Public class Employee{**

**@xmlElement**

**Private int id;**

**@XmlElement(name="name")**

Private String name;

**}**

**Step3: convert XML to Object**

//file as xml

File file = new File("museums.xml");

JAXBContext jaxbContext = JAXBContext.newInstance(Employee.class);

Unmarshaller jaxbUnmarshaller = jaxbContext.createUnmarshaller();

Employee employee = (Employee) jaxbUnmarshaller.unmarshal(file);

System.out.println(employee);

**//string as xml**

JAXBContext jaxbContext = JAXBContext.newInstance(Employee.class);

Unmarshaller jaxbUnmarshaller = jaxbContext.createUnmarshaller();

Employee employee = (Employee) jaxbUnmarshaller.unmarshal(new StringReader(empxml);

System.out.println(employee);

**Step3: convert Object to XML**

**//string as xml**

JAXBContext jaxbContext = JAXBContext.newInstance(Employee.class);

Marshaller marshaller = jaxbContext.createMarshaller();

marshaller.setProperty(javax.xml.bind.Marshaller.JAXB\_ENCODING, "UTF-8"); //NOI18N

marshaller.setProperty(javax.xml.bind.Marshaller.JAXB\_FORMATTED\_OUTPUT, Boolean.TRUE);

marshaller.marshal(quickXML, System.out);

OutputStream os = new FileOutputStream( "nosferatu.xml" );

marshaller.marshal( quickXML, os );

**Explanation:**

1. **Annotations:**
   * **@XmlRootElement: Marks the class as the root element of the XML.**
   * **@XmlElement: Maps the XML elements to the corresponding fields in the Java class.**
2. **JAXBContext: Used to initialize the JAXB framework for the Employee class.**
3. **Unmarshaller: Converts the XML file into a Java object.**
4. **Error Handling: The try-catch block ensures that any JAXBException is caught and handled.**

**This code is complete and ready to run. Let me know if you need further assistance!**

1. **Using Jackson XML(Jackson-dataformat-xml)**

**Spring boot integrates nicely with Jackson**

**Step 1: add dependency in the POM.xml**

<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.dataformat/jackson-dataformat-xml -->

<dependency>

<groupId>com.fasterxml.jackson.dataformat</groupId>

<artifactId>jackson-dataformat-xml</artifactId>

<version>2.19.2</version>

</dependency>

**Step 2: add a Model**

**import com.fasterxml.jackson.dataformat.xml.annotation.JacksonXmlProperty;**

public class JacksonDataForm {

@JacksonXmlProperty(localName = "id")

private int id;

@JacksonXmlProperty(localName = "name")

private String name;

@JacksonXmlProperty(localName = "email")

private String email;

// Getters and Setters

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getEmail() {

return email;

}

public void setEmail(String email) {

this.email = email;

}

}

**Step 3: conversion object to xml**

import com.fasterxml.jackson.dataformat.xml.XmlMapper;

public class XmlConversionExample {

public static void main(String[] args) {

try {

// Create a sample object

JacksonDataForm dataForm = new JacksonDataForm();

dataForm.setId(1);

dataForm.setName("John Doe");

dataForm.setEmail("john.doe@example.com");

// Create an XmlMapper instance

**XmlMapper xmlMapper = new XmlMapper();**

**// Convert the object to XML**

**String xmlOutput = xmlMapper.writeValueAsString(dataForm);**

**// Print the XML**

**System.out.println("Serialized XML:");**

System.out.println(xmlOutput);

} catch (Exception e) {

e.printStackTrace();

}

}

}

**XML to Object**

import com.fasterxml.jackson.dataformat.xml.XmlMapper;

public class XmlToObjectExample {

public static void main(String[] args) {

String xmlData = "<person>" +

" <id>1</id>" +

" <name>John Doe</name>" +

" <email>john.doe@example.com</email>" +

"</person>";

try {

// Create XmlMapper instance

XmlMapper xmlMapper = new XmlMapper();

// Deserialize XML to Java object

Person person = xmlMapper.readValue(xmlData, Person.class);

// Print the object

System.out.println(person);

} catch (Exception e) {

e.printStackTrace();

}

}

}

**Key Notes:**

1. **Dependencies**: Ensure jackson-dataformat-xml is included in your project.
2. **Annotations**: Use @JacksonXmlProperty for fine-grained control over XML element names.
3. **Error Handling**: Always handle exceptions gracefully in production code.

This approach works seamlessly with Spring Boot and Jackson for XML serialization.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |
| --- |
| **JSON to Object 🡪**   1. ObjectMapper.readValue(json, Employee.class)—>mannual parsing 2. @RequestBody in controller🡪Auto conversion 3. RestTemplate,WebClient🡪API calls between services   **Object to JSON🡪**  a ObjectMapper.writeValuesAsString(obj)—>mannual parsing   1. @ResponseBody return in controller🡪Auto conversion |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |
| --- |
| **SOAP CALL API building step by step**  **Step 1:**   1. **SOAP-Simple Object Access Protocol** 2. **XML based messaging protocol(Using WSDL for contract)** 3. **Communication via HTTP Post with content-type: text/XML** 4. **Works with Request /response style**   **Step 2: create springboot project**  **Use spring Initializer**   1. **Add dependencies**   Spring-boot-starter-web-services,  spring-boot-starter,  jaxb2   1. **Java 17+ recommended**   **POM.xml**  **Jakarta.xml.bind-api**  **Jaxb-runtime**   1. **Define XSD(Contract first approach)**   **Src/main/resources/person.xsd**   1. **Configure JAXB to generate classes—add into pom.xml—pojo generator**   maven-jaxb2-plugin—**run-mvn clean compile**  jaxb2-maven-plugin  Using jaxb2-maven-plugin is the easiest way. Define the plugins as below :  <build>  <plugins>  <plugin>  <groupId>org.codehaus.mojo</groupId>  <artifactId>jaxb2-maven-plugin</artifactId>  <version>1.6</version>  <executions>  <execution>  <goals>  <goal>xjc</goal>  </goals>  </execution>  </executions>  <configuration>  <schemaDirectory>${project.basedir}/src/main/xsd/</schemaDirectory>  <schemaFiles>MARC21slim.xsd</schemaFiles>  </configuration>  </plugin>  </plugins>  </build>  and execute :  mvn jaxb2:xjc  the generated files will be located in target\generated-sources\jaxb  One simple way to convert .xsd files to Java file is [xjc](https://docs.oracle.com/javase/8/docs/technotes/tools/unix/xjc.html) tool. Just execute the following command in the same working directory:  xjc test.xsd   * + 1. **Xjc test.xsd**        1. It is Standalone command line tool        2. Takes your XSD filer and generate POJOs directly        3. Example   Xjc test.xsd -d src/main/java -p com.example.generate   * + - 1. Works fine up to java 8(built in)       2. For java 11+, you need to add jaxb-xjc dependency(because JAXB was removed from JDK)       3. You must run it manually wherever XSD changes       4. Best for quick test, learning,small     1. **mvn jaxb2:xjc**        1. **A maven pluggin (maven-jaxb2-plugin)**        2. **Rund during maven lifecycle**        3. **Example**   <plugin>  <groupId>org.jcnnet.jaxb2.maven2</groupId>  <artifactId>jaxb2-maven-plugin</artifactId>  <version>0.14.0</version>  <executions>  <execution>  <goals>  <goal>generate</goal>  </goals>  </execution>  </executions>  <configuration>  <sources>  <source>src/main/resources</source>  </sources>  <packageName>comexample.generated</packageName>  </configuration>  </plugin>  RUN: mcn clean compile   * + - 1. POJOs will be auto generated into target/generated-sources/jaxb       2. Fully automated: every build generated updated class       3. Best for Spring Boot/Microservices/Enterpri projects with CI/CD   Xjc->manual,old-school,good for small  Mvn jaxb2:xjc->standard build  Mvn clean compile-standard build, autogenerated pojo if plugins is configured   1. **Web services configuration**   **@EnableWsd.**  **@Configuration**  **Class extends WsconfigureAdapter**   1. **Implement endpoint**   **@Endpoint**  **@PayloadRoot()**   1. **Run & test services**   **Start Springboot app->mvn spring-boot:run**  **WSDL available at** [**http://localhost:ws/person.wsdl**](http://localhost:ws/person.wsdl)  **Test with SOAPUI or CURL**  **Curl request : curl -X POST url\**  **-H “Content-Type:test/xml”\**  **-d ‘s<soapenv:Envelop ----:envelop>’**  **Response: in xml**   1. **Create a soap client**   **@Configuration** |

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

|  |
| --- |
| **Downstream API/Services call**   1. **restTemplate** 2. **Web client** 3. **Feign client** 4. **HttpClient/okHttp** 5. **Asynch RestTemplate** 6. **Messaging(Kafka,RabbitMQ, IBM MQ(WMQ))** 7. **gRPC**   **The modern microservices, the most common choices are**  **webClient(reactive)**  **Feign client(declarative, springcloud)**   1. **RestTemplate(blocking,old)** 2. Matured, well tested 3. Can use SSL/TLS interceptors and headers 4. Deprecated in spring 5-no new security features 5. Blocking model-hard to handle large scale secure call 6. Safe if used with HTTPS+security configs, but it is Outdated 7. **WebClient(Reactive, modern)** 8. Supports TLS,OAuth2,JWT, API keys easily 9. Built in support for Spring security integration 10. Non-blocking-avoid thread exhaustion attacks(safer under load) 11. Better for secure token propagation(e.g between icroservices) 12. Slightly more complex that restTemplate 13. It is more securte for modern apps because it integrates cleanly with Oauth2/OpenID 14. It is introduced in Spring 5 and springboot 2.x 15. Spring 5.3 and Spring boot 2.4+->Webclient is recommended replacement for rsttemplate 16. Spring6/spring boot 3(java 17 baseline)->Webclient is the primarly HTTPClient(with first security,HTTP/2, mtls etc) 17. **Open-Feign client** 18. Very developer friendly -less boilerplate, fewer mistakes 19. Works well withSpring security OAuth2, JWT,Basic Auth(token relay, interceptors) 20. Supports slower than webclient(adds abstraction) 21. Slightly slower than webclient(adds abstraction) 22. Security depends on how you configure interceptors 23. Safe, but must add token management properly 24. **okHttp, apacjhe HttpClient,gPRC, SOAP, Kafka)** 25. okhttp/apache httpclien-fine grained TLS & proxy config.   Coding example:  **webClient**  **step 1: add dependency in pom.xml**  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-webflux</artifactId>  </dependency>  WebClient client = WebClient.create();  WebClient client = WebClient.create("http://localhost:8080");  WebClient client = WebClient.builder()  .baseUrl("http://localhost:8080")  .defaultCookie("cookieKey", "cookieValue")  .defaultHeader(HttpHeaders.CONTENT\_TYPE, MediaType.APPLICATION\_JSON\_VALUE)  .defaultUriVariables(Collections.singletonMap("url", "http://localhost:8080"))  .build();  What is mono?   1. Mono<t> is a reactive type from project Reactor(which is Spring webflux). 2. It represents a single asynchronous value(0 or 1 element)   What is flux?   1. Flux<T> represents a stream of 0—n asynchronous values. 2. Used for multiple results(like a list of users, or a stream of events)   **What is BodyToMono?**   1. When you call an API with WebClient, you need to convert the Http response body into a java type. 2. .bodyToMono(foo.class)-means read the body and convert it into a Mono<Foo>(exactly one Foo object)   **What is bodyToFlux? We use this**   1. **If the response body is JSON array then**   **Note: Mono<T>-a box that may eventually may deliver one thing**  **Flux<T>->a conveyor belt that may deliver many things over time**  **bodyToMono(Foo.class)->unwrap the HTTP response into a single Foo inside a Moni.**  **FeignClient:**  **Step 1: add dependency in pom.xml**  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-openfeign</artifactId> </dependency>  **Step 2: Enable feign client in project**  @SpringBootApplication @EnableFeignClients public class Application {   public static void main(String[] args) {  SpringApplication.run(Application.class, args);  }  }  **Step 3: create a Feign client interface**  @FeignClient(name = "giveYourServiceName", url = "provideYourUrlHere", path = "provideYourContextPathHere") public interface AddressClient {   @GetMapping("/address/{id}")  public ResponseEntity<AddressResponse> getAddressByEmployeeId(@PathVariable("id") int id);  }  **HttpClient (Java 11+ Native HTTP Client)**  Java 11 introduced a new HttpClient API that can be used for HTTP communication. It is not specific to Spring but can be integrated into Spring Boot applications.  **Example: Using HttpClient**  import java.net.URI;  import java.net.http.HttpClient;  import java.net.http.HttpRequest;  import java.net.http.HttpResponse;  public class HttpClientExample {  public String getDataFromAnotherService(String url) throws Exception {  HttpClient client = HttpClient.newHttpClient();  HttpRequest request = HttpRequest.newBuilder()  .uri(new URI(url))  .GET()  .build();  HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());  if (response.statusCode() == 200) {  return response.body();  } else {  throw new RuntimeException("Failed to fetch data. HTTP Status: " + response.statusCode());  }  }  }  **Key Points:**   * **Native**: Part of the Java standard library (Java 11+). * **Synchronous and Asynchronous**: Supports both modes. * **No Spring Integration**: Requires manual integration with Spring.   **Choosing the Right Tool**   * Use **RestTemplate** for simple, synchronous use cases in legacy applications. * Use **WebClient** for modern, reactive, and asynchronous applications. * Use **HttpClient** if you prefer Java's native HTTP client and are not tied to Spring's ecosystem. |
|  |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |
| --- |
| **URI and Headers builder in Springboot microservices**  In Spring Boot microservices, the UriComponentsBuilder is a powerful utility class used to construct and manipulate URIs in a clean and flexible way. It is particularly useful when building URLs dynamically for REST API calls, redirects, or linking between services.  **Key Features of UriComponentsBuilder:**   1. **Dynamic URI Construction**: Build URIs dynamically by appending paths, query parameters, and other components. 2. **Immutable Design**: The builder creates immutable UriComponents objects, ensuring thread safety. 3. **Fluent API**: Provides a chainable API for constructing URIs.   **Example Use Cases:**   1. **Building a URI for Inter-Service Communication**: When one microservice needs to call another, you can dynamically construct the URI. 2. **Adding Query Parameters**: Useful for APIs that require query parameters for filtering or pagination. 3. **Handling Base URLs**: You can set a base URL and append paths or parameters dynamically.   **Example Code: Using UriComponentsBuilder**  package com.example.demo;  import org.springframework.web.util.UriComponentsBuilder;  public class UriBuilderExample {  public static void main(String[] args) {  // Base URL of the microservice  String baseUrl = "http://localhost:8080/api";  // Build a URI dynamically  String uri = UriComponentsBuilder.fromHttpUrl(baseUrl)  .path("/users/{id}") // Append path with a placeholder  .queryParam("includeDetails", true) // Add query parameter  .buildAndExpand(123) // Replace placeholder with actual value  .toUriString(); // Convert to String  System.out.println("Constructed URI: " + uri);  }  }  Output:  Constructed URI: <http://localhost:8080/api/users/123?includeDetails=true>  **Explanation:**   1. **fromHttpUrl(baseUrl)**: Sets the base URL. 2. **.path("/users/{id}")**: Appends a path with a placeholder ({id}). 3. **.queryParam("includeDetails", true)**: Adds a query parameter. 4. **.buildAndExpand(123)**: Replaces the {id} placeholder with the value 123. 5. **.toUriString()**: Converts the URI to a string.   Example: Using RestTemplate with UriComponentsBuilder  package com.example.demo;  import org.springframework.web.client.RestTemplate;  import org.springframework.web.util.UriComponentsBuilder;  public class RestTemplateExample {  public static void main(String[] args) {  RestTemplate restTemplate = new RestTemplate();  // Base URL of the microservice  String baseUrl = "http://localhost:8080/api";  // Build the URI  String uri = UriComponentsBuilder.fromHttpUrl(baseUrl)  .path("/products")  .queryParam("category", "electronics")  .queryParam("page", 1)  .toUriString();  // Make a GET request  String response = restTemplate.getForObject(uri, String.class);  System.out.println("Response: " + response);  }  }  **Example: Using UriComponentsBuilder in a Spring Controller**  package com.example.demo;  import org.springframework.web.bind.annotation.GetMapping;  import org.springframework.web.bind.annotation.RestController;  import org.springframework.web.util.UriComponentsBuilder;  import javax.servlet.http.HttpServletRequest;  @RestController  public class UriBuilderController {  @GetMapping("/build-uri")  public String buildUri(HttpServletRequest request) {  // Get the base URL dynamically from the request  String baseUrl = UriComponentsBuilder.fromHttpUrl(request.getRequestURL().toString())  .replacePath(null)  .toUriString();  // Build a new URI  String uri = UriComponentsBuilder.fromHttpUrl(baseUrl)  .path("/api/orders")  .queryParam("status", "pending")  .toUriString();  return "Constructed URI: " + uri;  }  }  **Key Methods in UriComponentsBuilder:**   * **fromHttpUrl(String url)**: Creates a builder with the given base URL. * **path(String path)**: Appends a path to the URI. * **queryParam(String name, Object... values)**: Adds query parameters. * **buildAndExpand(Object... uriVariables)**: Replaces placeholders in the URI. * **toUriString()**: Converts the URI to a string.   **Advantages in Microservices:**   1. **Dynamic and Flexible**: Easily construct URIs for inter-service communication. 2. **Readability**: The fluent API makes the code more readable and maintainable. 3. **Error-Free**: Automatically handles encoding of query parameters and special characters.   This approach is widely used in Spring Boot microservices for clean and efficient URI construction. |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |
| --- |
| **Message broker**  In a Spring Boot microservices architecture, a message broker plays a crucial role in enabling asynchronous communication between services. It helps decouple services, improve scalability, and ensure reliable message delivery. Below is an overview of how message brokers are used in Spring Boot microservices, along with examples of popular brokers like RabbitMQ and Apache Kafka.  **What is a Message Broker?**  A **message broker** is middleware that facilitates communication between microservices by:   * Receiving messages from a producer (sender). * Routing and delivering messages to one or more consumers (receivers). * Supporting patterns like **publish/subscribe** and **point-to-point messaging**.   Popular message brokers include:   * **RabbitMQ** (AMQP protocol) * **Apache Kafka** (distributed event streaming) * **ActiveMQ/artemis** * **Redis Streams** * Amazon SQS/SNS(cloud managed queues/topics) * Google Pub/Sub,Azurte service Bus etc * IBM MQ(WMQ)- IBM MQ is a message broker , but it’s not part of the “default” message brokers commonly used with Spring boot(like RabitMq or Kafka), Instead it is an enterprize option that you can integrate via Spring JMS * [Read and Write to IBM MQ Queue Using Java JMS | Baeldung](https://www.baeldung.com/java-message-service-ibm-mq-read-write)   **Why Use a Message Broker in Microservices?**   1. **Decoupling**: Services do not need to know about each other directly. 2. **Asynchronous Communication**: Services can process messages at their own pace. 3. **Scalability**: Enables horizontal scaling by distributing workloads. 4. **Reliability**: Ensures message delivery even if a service is temporarily unavailable. 5. **Event-Driven Architecture**: Supports real-time event processing.   **Spring Boot Integration with Message Brokers**  Spring Boot provides libraries and abstractions to work with message brokers:   1. **Spring AMQP**: For RabbitMQ. 2. **Spring Kafka**: For Apache Kafka. 3. **Spring Cloud Stream**: A unified abstraction for multiple messaging systems.   **1. Using RabbitMQ with Spring Boot**  RabbitMQ is a lightweight message broker that implements the **AMQP protocol**.  **Steps to Integrate RabbitMQ:**  **Add Dependencies**: Add the spring-boot-starter-amqp dependency to your pom.xml:  Copy code   1. <dependency> 2. <groupId>org.springframework.boot</groupId> 3. <artifactId>spring-boot-starter-amqp</artifactId> 4. </dependency>   **Configure RabbitMQ**: Add RabbitMQ connection details in application.properties:   1. Copy codespring.rabbitmq.host=localhost 2. spring.rabbitmq.port=5672 3. spring.rabbitmq.username=guest 4. spring.rabbitmq.password=guest   **Producer Example**:   1. Copy codeimport org.springframework.amqp.rabbit.core.RabbitTemplate; 2. import org.springframework.beans.factory.annotation.Autowired; 3. import org.springframework.stereotype.Service;   @Service   1. public class MessageProducer { 2. private static final String EXCHANGE\_NAME = "my\_exchange"; 3. private static final String ROUTING\_KEY = "my\_routing\_key"; 4. @Autowired 5. private RabbitTemplate rabbitTemplate; 6. public void sendMessage(String message) { 7. rabbitTemplate.convertAndSend(EXCHANGE\_NAME, ROUTING\_KEY, message); 8. System.out.println("Message sent: " + message); 9. } 10. }   **Consumer Example**:   1. Copy codeimport org.springframework.amqp.rabbit.annotation.RabbitListener; 2. import org.springframework.stereotype.Service; 3. @Service 4. public class MessageConsumer { 5. @RabbitListener(queues = "my\_queue") 6. public void receiveMessage(String message) { 7. System.out.println("Message received: " + message); 8. } 9. }   **Queue, Exchange, and Binding Configuration**:   1. Copy codeimport org.springframework.amqp.core.\*; 2. import org.springframework.context.annotation.Bean; 3. import org.springframework.context.annotation.Configuration; 4. @Configuration 5. public class RabbitMQConfig { 6. @Bean 7. public Queue queue() { 8. return new Queue("my\_queue", true); 9. } 10. @Bean 11. public TopicExchange exchange() { 12. return new TopicExchange("my\_exchange"); 13. } 14. @Bean 15. public Binding binding(Queue queue, TopicExchange exchange) { 16. return BindingBuilder.bind(queue).to(exchange).with("my\_routing\_key"); 17. } 18. }   **2. Using Apache Kafka with Spring Boot**  Kafka is a distributed event-streaming platform designed for high-throughput and fault-tolerant messaging.  **Steps to Integrate Kafka:**  **Add Dependencies**: Add the spring-kafka dependency to your pom.xml:   1. Copy code<dependency> 2. <groupId>org.springframework.boot</groupId> 3. <artifactId>spring-boot-starter-kafka</artifactId> 4. </dependency>   **Configure Kafka**: Add Kafka connection details in application.properties:   1. Copy codespring.kafka.bootstrap-servers=localhost:9092 2. spring.kafka.consumer.group-id=my-group 3. spring.kafka.consumer.auto-offset-reset=earliest   **Producer Example**:   1. Copy codeimport org.springframework.beans.factory.annotation.Autowired; 2. import org.springframework.kafka.core.KafkaTemplate; 3. import org.springframework.stereotype.Service;   @Service   1. public class KafkaProducer { 2. private static final String TOPIC = "my\_topic"; 3. @Autowired 4. private KafkaTemplate<String, String> kafkaTemplate; 5. public void sendMessage(String message) { 6. kafkaTemplate.send(TOPIC, message); 7. System.out.println("Message sent: " + message); 8. } 9. }   **Consumer Example**:   1. Copy codeimport org.springframework.kafka.annotation.KafkaListener; 2. import org.springframework.stereotype.Service; 3. @Service 4. public class KafkaConsumer { 5. @KafkaListener(topics = "my\_topic", groupId = "my-group") 6. public void consumeMessage(String message) { 7. System.out.println("Message received: " + message); 8. } 9. }   **3. Using Spring Cloud Stream**  Spring Cloud Stream provides a unified programming model for multiple messaging systems.  **Steps to Use Spring Cloud Stream:**  **Add Dependencies**: Add the following dependency to your pom.xml:   1. Copy code<dependency> 2. <groupId>org.springframework.cloud</groupId> 3. <artifactId>spring-cloud-starter-stream-rabbit</artifactId> 4. </dependency>   **Define Input/Output Channels**:   1. Copy codeimport org.springframework.cloud.stream.annotation.EnableBinding; 2. import org.springframework.cloud.stream.messaging.Source; 3. import org.springframework.messaging.support.MessageBuilder; 4. import org.springframework.stereotype.Service; 5. @EnableBinding(Source.class) 6. @Service 7. public class StreamProducer { 8. private final Source source; 9. public StreamProducer(Source source) { 10. this.source = source; 11. } 12. public void sendMessage(String message) { 13. source.output().send(MessageBuilder.withPayload(message).build()); 14. System.out.println("Message sent: " + message); 15. } 16. }   **When to Use Which Broker?**   * **RabbitMQ**: Best for traditional message queuing and lightweight use cases. * **Kafka**: Ideal for high-throughput, distributed event streaming. * **Spring Cloud Stream**: Use when you want to abstract the messaging system and support multiple brokers.   By integrating a message broker, you can build a robust, scalable, and decoupled microservices architecture in Spring Boot.  **IBM MQ:** [**Read and Write to IBM MQ Queue Using Java JMS | Baeldung**](https://www.baeldung.com/java-message-service-ibm-mq-read-write)  **public void receiveMessage() {**  **try {**  **Message message = receiver.receive(1000);**  **if (message instanceof TextMessage) {**  **TextMessage textMessage = (TextMessage) message;**  **} else {**  **// ...**  **}**  **} catch (JMSException e) {**  **// handle exception**  **} finally {**  **// close resources**  **}**  **}**  **TextMessage message = session.createTextMessage();**  **message.setText(messageText);**  **message.setStringProperty("OrderID", "12345");**  **Message message = receiver.receive(1000);**  **if (message instanceof TextMessage) {**  **TextMessage textMessage = (TextMessage) message;**  **String orderID = message.getStringProperty("OrderID");**  **}**  **Message message = receiver.receive(1000);**  **if (message instanceof TextMessage) {**  **TextMessage textMessage = (TextMessage) message;**  **String messageId = message.getJMSMessageID();**  **long timestamp = message.getJMSTimestamp();**  **long expiration = message.getJMSExpiration();**  **int priority = message.getJMSPriority();**  **}**  **String messageText = "Hello Baeldung! Nice to meet you!";**  **doNothing().when(sender).send(any(TextMessage.class));**  **messageSender.sendMessage(messageText);**  **verify(sender).send(any(TextMessage.class));**  **verify(textMessage).setText(messageText);**  **when(receiver.receive(anyLong())).thenReturn(textMessage);**  **when(textMessage.getText()).thenReturn("Hello Baeldung! Nice to meet you!");**  **messageReceiver.receiveMessage();**  **verify(textMessage).getText();** |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Log implementation**  In Spring Boot microservices, logging can be implemented in several ways using different frameworks and configurations. Below are the **main approaches** to implement logging:  **1. Default Logging with Spring Boot (Logback)**   * Spring Boot uses **Logback** as the default logging framework. * No additional configuration is required; it works out of the box. * You can customize logging levels and patterns in the application.properties or application.yml file: * logging.level.root=INFO * logging.level.com.example=DEBUG * logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} - %msg%n   **2. Using Log4j2**   * Replace Logback with **Log4j2** by excluding the default Logback dependency and adding Log4j2 dependencies in pom.xml: * <dependency> * <groupId>org.springframework.boot</groupId> * <artifactId>spring-boot-starter-log4j2</artifactId> * </dependency> * Configure log4j2.xml for advanced logging features like asynchronous logging, custom appenders, and filters.   **3. Using SLF4J (Simple Logging Facade for Java)**   * SLF4J is a logging facade that allows you to plug in different logging frameworks (e.g., Logback, Log4j2). * Use SLF4J APIs in your code: * import org.slf4j.Logger; * import org.slf4j.LoggerFactory; * public class MyService { * private static final Logger logger = LoggerFactory.getLogger(MyService.class); * public void doSomething() { * logger.info("This is an info message"); * logger.debug("This is a debug message"); * } * }   **4. Centralized Logging with ELK Stack (Elasticsearch, Logstash, Kibana)**   * Use the **ELK stack** to aggregate and visualize logs from multiple microservices. * Steps:   1. Configure your microservices to send logs to **Logstash**.   2. Logstash processes the logs and sends them to **Elasticsearch**.   3. Use **Kibana** to visualize and analyze the logs. * Example: Use a Logback appender to send logs to Logstash: * <appender name="LOGSTASH" class="net.logstash.logback.appender.LogstashTcpSocketAppender"> * <destination>localhost:5000</destination> * </appender>   **5. Distributed Tracing with Sleuth and Zipkin**   * Use **Spring Cloud Sleuth** for distributed tracing and logging across microservices. * Sleuth adds trace IDs and span IDs to logs, making it easier to trace requests across services. * Example configuration in application.properties: * spring.sleuth.sampler.probability=1.0 * spring.zipkin.base-url=http://localhost:9411   **6. Structured Logging with JSON**   * Log messages in **JSON format** for better integration with log aggregation tools. * Configure Logback or Log4j2 to output logs in JSON: * <encoder class="net.logstash.logback.encoder.LoggingEventCompositeJsonEncoder" />   **7. External Logging Services (e.g., Splunk, Datadog, AWS CloudWatch)**   * Integrate with external logging services for advanced log management and monitoring. * Use specific appenders or SDKs provided by these services to send logs.   **8. Custom Logging**   * Implement custom logging logic using interceptors, filters, or aspects. * Example: Use a Spring AOP aspect to log method execution details: * @Aspect * @Component * public class LoggingAspect { * private static final Logger logger = LoggerFactory.getLogger(LoggingAspect.class); * @Around("@annotation(org.springframework.web.bind.annotation.RequestMapping)") * public Object logExecution(ProceedingJoinPoint joinPoint) throws Throwable { * logger.info("Method called: " + joinPoint.getSignature()); * Object result = joinPoint.proceed(); * logger.info("Method finished: " + joinPoint.getSignature()); * return result; * } * }   **Summary Table**   | **Method** | **Use Case** | | --- | --- | | Default Logback | Simple, default logging. | | Log4j2 | Advanced logging features like async logging. | | SLF4J | Logging abstraction for flexibility. | | ELK Stack | Centralized logging and visualization. | | Sleuth + Zipkin | Distributed tracing for microservices. | | JSON Logging | Structured logs for better parsing. | | External Services | Advanced log management (e.g., Splunk, Datadog). | | Custom Logging | Application-specific logging requirements. |   Each method has its own advantages, and the choice depends on your application's requirements.  In Spring Boot microservices, **appenders** and **asynchronous appenders** are concepts primarily related to logging frameworks like **Logback** or **Log4j2**, which are commonly used for logging in Java applications. Here's a detailed explanation of the difference between the two:  **1. Appender**  An **appender** is a component in logging frameworks that is responsible for writing log messages to a specific destination, such as:   * Console * File * Database * Remote server, etc.   **Characteristics:**   * **Synchronous by default**: Standard appenders process log events synchronously, meaning the logging operation happens in the same thread as the application logic. * **Blocking behavior**: If the appender is slow (e.g., writing to a file or sending logs to a remote server), it can block the application thread, potentially degrading performance. * **Use case**: Suitable for applications where logging performance is not critical or where logs are written to fast destinations like the console.   **Example (Logback configuration for a synchronous file appender):**  <appender name="FILE" class="ch.qos.logback.core.FileAppender">  <file>application.log</file>  <encoder>  <pattern>%d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n</pattern>  </encoder>  </appender>  **2. Asynchronous Appender**  An **asynchronous appender** is a special type of appender that processes log events in a separate thread, decoupling the logging operation from the main application thread.  **Characteristics:**   * **Non-blocking**: Log events are placed in a queue and processed by a separate thread, ensuring that the application thread is not blocked by slow logging operations. * **Improved performance**: Reduces the impact of logging on application performance, especially in high-throughput systems or microservices. * **Queue-based**: Uses an internal queue to buffer log events before processing them. If the queue is full, log events may be dropped or the application may block (depending on configuration). * **Use case**: Ideal for high-performance applications or microservices where logging should not interfere with the main application logic.   **Example (Logback configuration for an asynchronous appender):**  <appender name="ASYNC\_FILE" class="ch.qos.logback.classic.AsyncAppender">  <queueSize>500</queueSize> <!-- Maximum number of log events in the queue -->  <discardingThreshold>0</discardingThreshold> <!-- Drop logs if queue is full -->  <appender-ref ref="FILE" />  </appender>  **Key Differences**   | **Feature** | **Appender (Synchronous)** | **Asynchronous Appender** | | --- | --- | --- | | **Threading** | Runs in the same thread as the application | Runs in a separate thread | | **Performance Impact** | Can block the application thread | Non-blocking, minimal impact on performance | | **Use Case** | Low-throughput systems or simple logging | High-throughput systems or microservices | | **Queue** | No queue; logs are processed immediately | Uses a queue to buffer log events | | **Risk** | Slower logging can degrade application performance | Risk of log loss if the queue overflows |   **When to Use Asynchronous Appenders in Microservices**   * **High Throughput**: If your microservice generates a large volume of logs, asynchronous appenders ensure that logging does not slow down the application. * **Remote Logging**: When logs are sent to a remote server (e.g., via HTTP or Kafka), asynchronous appenders prevent network latency from affecting the application. * **Scalability**: Asynchronous logging helps microservices scale better by reducing contention on application threads.   **Example in a Spring Boot Microservice**  To configure asynchronous logging in a Spring Boot application using Logback:   1. Add the logback.xml file in the src/main/resources directory. 2. Use an AsyncAppender to wrap your existing appenders.   <configuration>  <!-- Synchronous File Appender -->  <appender name="FILE" class="ch.qos.logback.core.FileAppender">  <file>application.log</file>  <encoder>  <pattern>%d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n</pattern>  </encoder>  </appender>  <!-- Asynchronous Appender -->  <appender name="ASYNC\_FILE" class="ch.qos.logback.classic.AsyncAppender">  <queueSize>1000</queueSize>  <appender-ref ref="FILE" />  </appender>  <!-- Root Logger -->  <root level="info">  <appender-ref ref="ASYNC\_FILE" />  </root>  </configuration>  **Summary**   * Use **synchronous appenders** for simple applications where performance is not a concern. * Use **asynchronous appenders** in microservices or high-performance applications to ensure logging does not block the main application logic. |

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Token management**   * 1. **JWT token**   2. **E2E\_trust\_token**   3. **IAM token**   **In Spring Boot microservices, tokens are commonly used for authentication and authorization. The most widely used token type is the JSON Web Token (JWT). Below is an overview of token types and their usage in Spring Boot microservices:**  **1. JSON Web Token (JWT)**   * **What is JWT?**   + **JWT is a compact, URL-safe token format defined by the RFC 7519 standard.**   + **It is used to securely transmit information between parties as a JSON object.**   + **JWTs are self-contained, meaning they include all the necessary information (e.g., user roles, expiration time) to verify the user's identity without querying a database.** * **Structure of JWT: A JWT consists of three parts:**   + **Header: Specifies the type of token (JWT) and the signing algorithm (e.g., HS256, RS256).**   + **Payload: Contains claims (e.g., user ID, roles, expiration time).**   + **Signature: Ensures the token's integrity and authenticity.**   **Example:**  **eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJ1c2VySWQiLCJyb2xlcyI6IlVTRVIiLCJleHAiOjE2ODI2NzYwMDB9.abc123signature**   * **Advantages:**   + **Stateless: No need to store tokens on the server.**   + **Compact: Easy to transmit in HTTP headers or URLs.**   + **Secure: Can be signed and optionally encrypted.** * **Use in Spring Boot Microservices:**   + **Authentication: Users log in and receive a JWT.**   + **Authorization: JWT is passed with each request (usually in the Authorization header as a Bearer token) to verify access to resources.**   **2. OAuth2 Access Tokens**   * **What is OAuth2?**   + **OAuth2 is an authorization framework that issues access tokens to clients for accessing protected resources.**   + **Tokens can be JWT-based or opaque (non-readable).** * **Types of OAuth2 Tokens:**   + **Access Token: Grants access to specific resources for a limited time.**   + **Refresh Token: Used to obtain a new access token without requiring the user to log in again.** * **Use in Spring Boot Microservices:**   + **Spring Security provides built-in support for OAuth2.**   + **OAuth2 tokens are often used in distributed systems where microservices communicate with each other.**   **3. Session Tokens**   * **What are Session Tokens?**   + **Session tokens are server-side tokens stored in a database or in-memory store (e.g., Redis).**   + **The token is issued to the client and validated on the server for each request.** * **Use in Spring Boot Microservices:**   + **Less common in microservices due to the need for centralized session storage.**   + **Suitable for monolithic applications or when JWT is not preferred.**   **Implementation of JWT in Spring Boot Microservices**  **Here’s a basic example of how JWT is implemented in Spring Boot:**  **1. Generate JWT**  **import io.jsonwebtoken.Jwts;**  **import io.jsonwebtoken.SignatureAlgorithm;**  **import java.util.Date;**  **public class JwtUtil {**  **private static final String SECRET\_KEY = "your\_secret\_key";**  **public static String generateToken(String username) {**  **return Jwts.builder()**  **.setSubject(username)**  **.setIssuedAt(new Date())**  **.setExpiration(new Date(System.currentTimeMillis() + 1000 \* 60 \* 60)) // 1 hour**  **.signWith(SignatureAlgorithm.HS256, SECRET\_KEY)**  **.compact();**  **}**  **}**  **2. Validate JWT**  **import io.jsonwebtoken.Claims;**  **import io.jsonwebtoken.Jwts;**  **public class JwtUtil {**  **private static final String SECRET\_KEY = "your\_secret\_key";**  **public static Claims validateToken(String token) {**  **return Jwts.parser()**  **.setSigningKey(SECRET\_KEY)**  **.parseClaimsJws(token)**  **.getBody();**  **}**  **}**  **3. Use JWT in Spring Security**   * **Configure Spring Security to validate JWTs incoming requests.** * **Add a filter to extract and validate the token from the Authorization header.**   **Conclusion**   * **JWT is the most common token type in Spring Boot microservices due to its stateless nature and ease of use.** * **OAuth2 tokens are used when integrating with third-party services or implementing advanced authorization flows.** * **Session tokens are less common in microservices but may be used in specific scenarios.**   **If you need a detailed implementation guide, let me know!**  **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***  **Here’s a basic JWT implementation example in a Spring Boot microservice using Spring Security and jjwt (Java JWT library). This setup includes:**   * **User authentication** * **JWT token generation** * **Token validation** * **Securing endpoints**   **1. Add Dependencies (Maven)**  <dependencies>      <dependency>          <groupId>org.springframework.boot</groupId>  **<artifactId>spring-boot-starter-security</artifactId>**      </dependency>      <dependency>          <groupId>io.jsonwebtoken</groupId>          <artifactId>jjwt</artifactId>          <version>0.9.1</version>      </dependency>      <dependency>          <groupId>org.springframework.boot</groupId>  **<artifactId>spring-boot-starter-web</artifactId>**      </dependency>  </dependencies>  **2. JWT Utility Class**  **import io.jsonwebtoken.\*;**  **import org.springframework.stereotype.Component;**  **import java.util.Date;**  **@Component**  **public class JwtUtil {**  **private final String SECRET\_KEY = "your\_secret\_key";**  **generateToken(String username) {**  **return Jwts.builder()**  **.setSubject(username)**  **.setIssuedAt(new Date())**  **.setExpiration(new Date(System.currentTimeMillis() + 1000 \* 60 \* 60)) // 1 hour**  **.signWith(SignatureAlgorithm.HS256, SECRET\_KEY)**  **.compact();**  **}**  **public String extractUsername(String token) {**  **return Jwts.parser().setSigningKey(SECRET\_KEY)**  **.parseClaimsJws(token).getBody().getSubject();**  **}**  **public boolean validateToken(String token) {**  **try {**  **Jwts.parser().setSigningKey(SECRET\_KEY).parseClaimsJws(token);**  **return true;**  **} catch (JwtException | IllegalArgumentException e) {**  **return false;**  **}**  **}**  **}**  **3. Authentication Controller**  **import org.springframework.beans.factory.annotation.Autowired;**  **import org.springframework.web.bind.annotation.\*;**  **@RestController**  **@RequestMapping("/auth")**  **publicController {**  **@Autowired**  **private JwtUtil jwtUtil;**  **@PostMapping("/login")**  **public String login(@RequestParam String username, @RequestParam String password) {**  **// Dummy check, replace with real authentication**  **if ("user".equals(username) &amp;&amp; "password".equals(password)) {**  **return jwtUtil.generateToken(username);**  **} else {**  **throw new RuntimeException("Invalid credentials");**  **}**  **}**  **}**  **4. JWT Filter**  **import jakarta.servlet.\*;**  **import jakarta.servlet.http.HttpServletRequest;**  **import jakarta.servlet.http.HttpServletResponse;**  **import org.springframework.beans.factory.annotation.Autowired;**  **import org.springframework.stereotype.Component;**  **import java.io.IOException;**  **@Component**  **public class JwtFilter implements Filter {**  **@Autowired**  **private JwtUtil jwtUtil;**  **@Override**  **public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)**  **throws IOException, ServletException {**  **HttpServletRequest req = (HttpServletRequest) request;**  **String authHeader = req.getHeader("Authorization");**  **if (authHeader != null &amp;&amp; authHeader.startsWith("Bearer ")) {**  **String token = authHeader.substring(7);**  **if (!jwtUtil.validateToken(token)) {**  **((HttpServletResponse) response).sendError(HttpServletResponse.SC\_UNAUTHORIZED, "Invalid Token");**  **return;**  **}**  **} else {**  **((HttpServletResponse) response).sendError(HttpServletResponse.SC\_UNAUTHORIZED, "Missing Token");**  **return;**  **}**  **chain.doFilter(request, response);**  **}**  **}**  **5. Security Configuration**  **import org.springframework.beans.factory.annotation.Autowired;**  **import org.springframework.context.annotation.Configuration;**  **import org.springframework.security.config.annotation.web.builders.HttpSecurity;**  **import org.springframework.security.web.SecurityFilterChain;**  **import org.springframework.context.annotation.Bean;**  **@Configuration**  **public class SecurityConfig {**  **@Autowired**  **private JwtFilter jwtFilter;**  **@Bean**  **public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {**  **http.csrf().disable()**  **.authorizeHttpRequests(auth -&gt; auth**  **.requestMatchers("/auth/\*\*").permitAll()**  **.anyRequest().authenticated()**  **)**  **.addFilterBefore(jwtFilter, org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter.class);**  **return http.build();**  **}**  **}**  **6. Sample Secured Endpoint**  **@RestController**  **@RequestMapping("/api")**  **public class ApiController {**  **@GetMapping("/secure-data")**  **public String getSecureData() {**  **return "This is protected data";**  **}**  **}**  **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***  **🛡️ HSBC E2E Trust Token Overview**  **HSBC uses a signed JWT (JSON Web Token) called the E2E Trust Token for secure API authentication**  **1**  **. This token must be included in the HTTP request headers when invoking HSBC's Taas services.**  **Token Structure**   1. **Header:**    * **ver: Version (e.g., "1.0")**    * **kid: Public Key ID used by HSBC to verify the signature**    * **typ: Token type ("JWT")**    * **alg: Signature algorithm (recommended: PS256, PS384, PS512)** 2. **Payload:**    * **sub: Subject (API Profile ID)**    * **aud: Audience (intended recipient)**    * **payload\_hash: Hash of the encrypted HTTP request body**    * **payload\_hash\_alg: Hashing algorithm (e.g., SHA-256)**    * **iat: Issued at timestamp**    * **jti: Unique token ID to prevent replay attacks** 3. **Signature:**    * **Signed using the client's private credentials and the specified algorithm.**   **This structure ensures authentication, integrity, and non-repudiation of API requests**  **1**  **.**  **🔄 JWT Propagation Between Microservices (Spring Boot)**  **In a microservices architecture, secure token propagation is essential for inter-service communication. Here's how to implement it using Spring Cloud and Feign clients**  **2**  **:**  **Step 1: Feign Client Setup**  **Step 2: JWT Propagation via RequestInterceptor**  **This interceptor ensures that the JWT token is attached to all outgoing requests from the service.**  **Step 3: Asynchronous Context Propagation**  **For @Async methods or custom thread pools, use:**  **This ensures the security context (and JWT) is preserved across threads.**  **✅ Best Practices for Secure Token Handling**   * **Validate tokens in each microservice or at the API Gateway.** * **Use HTTPS for all communications.** * **Implement token expiration, renewal, and revocation strategies.** * **Monitor and audit token usage for compliance and security.** * **Consider token caching to reduce latency.**   **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***  **🔐 IAM Token Usage in HSBC Channel-Side Microservices**  **1. Purpose of IAM Tokens**  **IAM tokens serve as access tokens that:**   * **Authenticate the identity of the calling channel (e.g., mobile app, web portal).** * **Authorize access to specific microservices or APIs.** * **Ensure secure, traceable, and auditable communication.**   **These tokens are often JWTs (JSON Web Tokens) issued by an OAuth 2.0 Authorization Server.**  **🧩 Architecture Overview**  **Authorization Server**   * **Issues IAM tokens using OAuth 2.0 flows (commonly client\_credentials or authorization\_code).** * **Configured using Spring Security and @EnableAuthorizationServer**   **1**  **.**  **Resource Server (Microservices)**   * **Validates IAM tokens using Spring Security OAuth2.** * **Annotated with @EnableResourceServer and configured to check token validity**   **1**    **2**  **.**  **Token Validation**   * **Tokens are validated via a /check\_token endpoint or introspection URL.** * **Scopes and claims in the token determine access levels.**   **⚙️ Spring Boot Implementation Steps**  **Step 1: Add Dependencies**  **Step 2: Configure Authorization Server**  **Step 3: Configure Resource Server**  **Step 4: Token Validation**  **✅ Best Practices in HSBC Context**   * **Use scopes to define fine-grained access control (e.g., SCOPE\_read, SCOPE\_write).** * **Centralize token issuance via a secure IAM service.** * **Propagate IAM tokens across microservices using Feign interceptors or Spring Cloud Gateway.** * **Audit and monitor token usage for compliance and security.**   **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***  **🔐 What Is a Token Group?**  **A token group is a logical classification or grouping of tokens based on their purpose, issuer, or scope. In HSBC-style microservices, token groups help manage:**   * **Channel tokens (used by external clients like mobile/web apps)** * **IAM tokens (used for identity and access management)** * **E2E Trust tokens (used for secure inter-service communication)** * **Service tokens (used internally between microservices)**   **Each group may have different validation rules, lifespans, and scopes.**  **🧭 Token Validation in Proxy Layer (API Gateway)**  **In Spring Boot microservices, the API Gateway acts as the proxy layer that validates tokens before forwarding requests to downstream services. Here's how it works:**  **1. JWT Token Validation Flow**   * **The client sends a request with a JWT in the Authorization header.** * **The gateway intercepts the request using a filter (e.g., JwtAuthenticationFilter).** * **The filter checks:**   + **If the endpoint is secured**   + **If the token is present and well-formed**   + **If the token belongs to a valid token group**   + **If the token is not expired and correctly signed**   **2. Token Group-Based Routing**   * **The gateway can route requests differently based on token group:**   + **Channel tokens → Public APIs**   + **IAM tokens → Internal services**   + **Trust tokens → Secure backend services**   **This is configured using route filters and token metadata (e.g., claims like iss, aud, scope).**  **3. Example: JWT Filter in Gateway**  **✅ Best Practices**   * **Centralize token validation at the gateway to reduce duplication.** * **Use custom claims in JWTs to identify token groups.** * **Implement role-based access control (RBAC) using token claims.** * **Ensure token expiration and revocation mechanisms are in place.** * **Use HTTPS for all communications to prevent token leakage.** * **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***   **🔍 How to Check Token Group in JWT**  **✅ Step 1: Decode the JWT**  **Use any of these online tools:**   * **JWT.io** * [**CodeShack JWT Decoder**](https://codeshack.io/jwt-decoder/) * [**CodLab JWT Decoder**](https://codlab.dev/decoder/jwt-decoder)   **Paste your token and inspect the payload section.**  **🧩 Key Claims to Identify Token Group**  **Look for these claims in the decoded payload:**   | **Claim** | **Description** | | --- | --- | | **groups** | **Lists group IDs the user belongs to (e.g., security groups, distribution lists) — common in Azure AD** | | **roles** | **Application roles assigned to the user or group (e.g., Reviewer, Approver) — useful for RBAC** | | **wids** | **Directory roles (e.g., Application Developer) — specific to Microsoft Entra ID** | | **hasgroups** | **Indicates group overage (too many groups to fit in token) — use Graph API to fetch full list** | | **aud** | **Audience — helps identify which service the token is intended for.** | | **iss** | **Issuer — helps identify the source of the token (e.g., Azure AD, HSBC IAM).** | | **scp or scope** | **Permissions granted to the token — often used to infer token group.** |   **🛠️ Handling Group Overages**  **If the user belongs to too many groups (e.g., more than 200 for JWT), the token may omit the groups claim and instead include:**   * **hasgroups: true** * **\_claim\_names with a reference to fetch groups via Microsoft Graph API**   **🧠 Best Practices**   * **Use application roles instead of raw group IDs for better scalability and clarity**   **1**  **.**   * **For real-time group membership, query Microsoft Graph API.** * **In Spring Boot, use a JWT filter to extract and validate group claims for authorization.**   **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*** |

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **DB related all implementation**   * 1. **Connection pool**   2. **JPA**   3. **Hibernate**   4. **DB types**   5. **Query optimization**   6. **SQL query interview and project level** * **Connection pools**   1. **Hikari**   2. **DBCP2**   In Spring Boot microservices, several types of database connection pools are commonly used to manage and optimize database connections. Each has its own strengths and configuration style. Here's a breakdown of the most popular ones:  🔹 1. **HikariCP (Default in Spring Boot)**   * Overview: High-performance, lightweight connection pool. * Pros:   + Fastest among all pools.   + Low memory footprint.   + Built-in leak detection. * Configuration: * Use Case: Recommended for most Spring Boot applications   1  .  🔹 2. **Apache Commons DBCP2**   * Overview: Mature and stable connection pool from Apache. * Pros:   + Fine-grained control over pool behavior.   + Good for legacy systems. * Configuration: * Use Case: Suitable for applications needing detailed tuning   2  .  🔹 3. **Tomcat JDBC Connection Pool**   * Overview: Built-in with Apache Tomcat, used in older Spring Boot versions. * Pros:   + Integrated with Tomcat.   + Good performance and reliability. * Configuration: * Use Case: Useful when deploying on Tomcat or for backward compatibility   3  .  **🔹 4. C3P0**   * Overview: Older connection pool library. * Pros:   + Easy to configure.   + Supports automatic recovery. * Cons:   + Slower than HikariCP and DBCP2. * Use Case: Legacy applications or simple setups.   \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Here’s a **step-by-step guide** to implement **HikariCP** in a **Spring Boot microservice**:  **✅ Step-by-Step Implementation of HikariCP**  **🔹 Step 1: Create a Spring Boot Project**  You can use Spring Initializr or your IDE to create a new project with dependencies like:   * Spring Web * Spring Data JPA * MySQL Driver (or your preferred DB)   **🔹 Step 2: Add Dependencies (if not already included)**  In pom.xml (for Maven):  Spring Boot includes HikariCP by default, so you usually don’t need to add it manually.  **🔹 Step 3: Configure application.properties**  You can also use application.yml if preferred.  **🔹 Step 4: Verify HikariCP is Active**  Spring Boot logs the connection pool type on startup. You should see something like:  HikariPool-1 - Starting...  HikariPool-1 - Start completed.  Or you can log it manually:  **🔹 Step 5: Use JPA or JDBC Normally**  You can now use Spring Data JPA or JDBC templates as usual. HikariCP will manage the connections behind the scenes.  **🔹 Step 6: Optional Monitoring**  HikariCP supports JMX monitoring. You can enable it by adding:  Then use tools like JConsole or VisualVM to monitor pool metrics.  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Here’s a **step-by-step guide** to implement **Apache Commons DBCP2** connection pooling in a **Spring Boot microservice**:  **✅ Step-by-Step Implementation of DBCP2 in Spring Boot**  **🔹 Step 1: Create a Spring Boot Project**  Use Spring Initializr or your IDE to create a new project with:   * Spring Web * Spring Data JPA * MySQL/PostgreSQL Driver (or your preferred DB)   **🔹 Step 2: Add DBCP2 Dependency**  In pom.xml (for Maven):  Spring Boot does not include DBCP2 by default, so this step is necessary.  **🔹 Step 3: Configure application.properties**  **🔹 Step 4: Verify DBCP2 is Active**  You can log the datasource type to confirm it's using DBCP2:  Expected output:  Using DataSource: org.apache.commons.dbcp2.BasicDataSource  **🔹 Step 5: Use JPA or JDBC Normally**  You can now use Spring Data JPA or JDBC templates as usual. DBCP2 will manage the connections behind the scenes.  **🔹 Step 6: Optional Monitoring and Leak Detection**  Enable abandoned connection cleanup:  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  To integrate **Apache Commons DBCP2** with **PostgreSQL** in a **Spring Boot microservice**, follow these steps:  **✅ Step-by-Step Integration of DBCP2 with PostgreSQL**  **🔹 Step 1: Create a Spring Boot Project**  Use Spring Initializr or your IDE and include:   * Spring Web * Spring Data JPA * PostgreSQL Driver   **🔹 Step 2: Add Required Dependencies**  In pom.xml:  **🔹 Step 3: Configure application.properties**  **🔹 Step 4: Verify DBCP2 is Active**  Add a simple log to confirm the pool type:  Expected output:  Using DataSource: org.apache.commons.dbcp2.BasicDataSource  **🔹 Step 5: Use JPA or JDBC Normally**  You can now use Spring Data JPA repositories or JdbcTemplate as usual. DBCP2 will manage the connections.  **🔹 Step 6: Optional – Enable Leak Detection**  spring.datasource.dbcp2.remove-abandoned-on-borrow=true  spring.datasource.dbcp2.remove-abandoned-on-maintenance=true  spring.datasource.dbcp2.remove-abandoned-timeout=60  spring.datasource.dbcp2.log-abandoned=true  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  To implement **JPA (Java Persistence API)** in a **Spring Boot microservice**, especially with a connection pool like **DBCP2**, you follow a structured approach that integrates database access with entity management. Here's a detailed step-by-step guide:  **✅ Step-by-Step JPA Implementation in Spring Boot Microservices**  **🔹 Step 1: Create a Spring Boot Project**  Use Spring Initializr or your IDE and include:   * Spring Web * Spring Data JPA * PostgreSQL/MySQL Driver (based on your DB) * Optional: Spring Boot DevTools for hot reload   **🔹 Step 2: Add Dependencies in pom.xml**  <!-- Spring Data JPA -->  <dependency>      <groupId>org.springframework.boot</groupId>      <artifactId>spring-boot-starter-data-jpa</artifactId>  </dependency>  <!-- PostgreSQL Driver -->  <dependency>      <groupId>org.postgresql</groupId>      <artifactId>postgresql</artifactId>      <version>42.7.1</version>  </dependency>  <!-- Apache Commons DBCP2 -->  <dependency>      <groupId>org.apache.commons</groupId>      <artifactId>commons-dbcp2</artifactId>      <version>2.9.0</version>  </dependency>  **🔹 Step 3: Configure application.properties**  **🔹 Step 4: Create Entity Class**  **🔹 Step 5: Create Repository Interface**  **🔹 Step 6: Create Service Layer**  **🔹 Step 7: Create REST Controller**  **🔹 Step 8: Run and Test**   * Start your Spring Boot application. * Use Postman or curl to test endpoints:   + GET /api/employees   + POST /api/employees   \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  The comparison between **JPA (Java Persistence API)** and **Hibernate** often causes confusion because they are closely related. Here's a clear breakdown to help you understand which one is "better" depending on your use case:  **🔍 JPA vs Hibernate: Key Differences**   | **Feature** | **JPA** | **Hibernate** | | --- | --- | --- | | **Type** | Specification (standard API) | Implementation of JPA + additional features | | **Ownership** | Maintained by Oracle (Java EE standard) | Developed by Red Hat | | **Purpose** | Defines how Java objects map to DB tables | Provides actual code to perform persistence | | **Flexibility** | Can switch between providers (e.g., EclipseLink, Hibernate) | Tied to Hibernate-specific features | | **Features** | Basic ORM operations | Advanced ORM features (caching, filters, etc.) | | **Usage in Spring Boot** | Used via Spring Data JPA abstraction | Hibernate is the default JPA provider |   **✅ What Is JPA?**   * **JPA** is a **standard API** for object-relational mapping (ORM). * It defines annotations like @Entity, @Table, @Id, etc. * It does **not** perform any persistence itself—it needs a provider like Hibernate.   **✅ What Is Hibernate?**   * **Hibernate** is a **JPA provider** and a **full-featured ORM framework**. * It implements JPA and adds many powerful features:   + First-level and second-level caching   + Lazy/eager loading   + Custom SQL queries   + Advanced mapping strategies   **🧠 Which One Should You Use?**  **✅ Use JPA if:**   * You want **portability** across different ORM providers. * You prefer **standardized code** and future-proofing. * You're using **Spring Data JPA**, which abstracts JPA and works seamlessly with Hibernate.   **✅ Use Hibernate directly if:**   * You need **advanced features** not available in JPA (e.g., filters, interceptors). * You want **fine-grained control** over the persistence layer. * You're working on a complex legacy system already using Hibernate.   **🔧 In Practice (Spring Boot)**  Most Spring Boot applications use **Spring Data JPA**, which:   * Uses **JPA annotations and interfaces**. * Uses **Hibernate as the default provider** under the hood.   So you're often using **both**: JPA for abstraction, Hibernate for execution.  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Implementing **Hibernate in Spring Boot microservices** involves integrating Hibernate as the JPA provider to handle database operations. Here's a **step-by-step guide** to help you set it up:  **✅ Step 1: Set Up Your Spring Boot Project**  You can use Spring Initializr to generate a base project.  **Dependencies to include:**   * Spring Web * Spring Data JPA * Hibernate (comes with Spring Data JPA) * H2/MySQL/PostgreSQL (depending on your DB)   **✅ Step 2: Configure application.properties or application.yml**  For example, using MySQL:  **✅ Step 3: Create Entity Classes**  Hibernate works with entity classes annotated with @Entity.  **✅ Step 4: Create a Repository Interface**  Use Spring Data JPA to abstract Hibernate operations.  **✅ Step 5: Create a Service Layer**  Encapsulate business logic here.  **✅ Step 6: Create a REST Controller**  Expose endpoints for your microservice.  **✅ Step 7: Test Your Microservice**  Run the application and test endpoints using:   * Postman * Swagger (add Springdoc OpenAPI dependency) * Curl   **✅ Step 8: Optional Enhancements**   * Add DTOs and ModelMapper for clean API responses. * Use @Transactional for managing transactions. * Add exception handling with @ControllerAdvice. |

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

|  |
| --- |
| **Annotation level all implementation in project**   1. **Request filter** 2. **@PostConstruct** 3. **@Retryable** 4. **@Recover** 5. **@Bean** 6. **@Controller** 7. **@Qualifier** 8. **@Resource** 9. **@PathVariable** 10. **@QueryParam** 11. **@Query** 12. **@ControllerAdvice** 13. **@Services** 14. **@repository** 15. **@Asynch** 16. **@Synch** 17. **@Autowired**   **🔹 1. RequestFilter**  Purpose: Intercepts HTTP requests before they reach the controller.  Usage:  @Component  public class CustomRequestFilter implements Filter {      public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) {          // Pre-processing logic          chain.doFilter(request, response);      }  }  **Used in:** Logging, authentication, modifying headers.  Here’s a detailed explanation of each annotation you listed, describing its **purpose**, **usage**, and **where it fits** in a typical Spring Boot + Hibernate REST API project:  **🔹 2. @PostConstruct**  **Purpose:** Executes a method after dependency injection is done.  **Usage:**  @PostConstruct  public void init() {      // Initialization logic  }  **Used in:** Setting up resources, loading config, initializing beans.  **🔹 3. @Retryable**  **Purpose:** Automatically retries a method on failure.  **Usage:**  @Retryable(value = Exception.class, maxAttempts = 3)  public void fetchData() {      // Retry logic  }  **Used in:** Network calls, external service integration.  **🔹 4. @Recover**  **Purpose:** Defines fallback logic after retries fail.  **Usage:**  @Recover  public void recover(Exception e) {      // Fallback logic  }  **Used in:** Graceful degradation, error handling.  **🔹 5. @Bean**  **Purpose:** Declares a method that returns a Spring-managed bean.  **Usage:**  @Bean  public RestTemplate restTemplate() {      return new RestTemplate();  }  **Used in:** Configuration classes.  **🔹 6. @Controller**  **Purpose:** Marks a class as a Spring MVC controller.  **Usage:**  **Used in:** Web applications (returns views).  **🔹 7. @Qualifier**  **Purpose:** Specifies which bean to inject when multiple candidates exist.  **Usage:**  @Autowired  @Qualifier("employeeService")  private Service service;  **Used in:** Resolving bean conflicts.  **🔹 8. @Resource**  **Purpose:** Injects a bean by name (JDK standard).  **Usage:**  @Resource(name = "employeeService")  private Service service;  **Used in:** Alternative to @Autowired.  **🔹 9. @PathVariable**  **Purpose:** Binds URL path segments to method parameters.  **Usage:**  **Used in:** RESTful endpoints.  **🔹 10. @QueryParam *(from JAX-RS, not Spring)***  **Purpose:** Binds query parameters in JAX-RS.  **Spring Equivalent:**  **🔹 11. @Query**  **Purpose:** Defines custom JPQL or native SQL queries.  **Usage:**  **Used in:** Repository interfaces.  **🔹 12. @ControllerAdvice**  **Purpose:** Global exception handling and model binding.  **Usage:**  **Used in:** Centralized error handling.  **🔹 13. @Service**  **Purpose:** Marks a class as a service layer component.  **Usage:**  **Used in:** Business logic layer.  **🔹 14. @Repository**  **Purpose:** Marks a class as a DAO component.  **Usage:**  **Used in:** Data access layer.  **🔹 15. @Async**  **Purpose:** Executes methods asynchronously.  **Usage:**  **Used in:** Background tasks, notifications.  **🔹 16. @Sync *(Not a standard Spring annotation)***  There is **no official @Sync annotation** in Spring. You might be referring to:   * synchronized keyword in Java * Custom annotations for locking   Let me know if you meant something else here.  **🔹 17. @Autowired**  **Purpose:** Automatically injects dependencies.  **Usage:**  **Used in:** All layers (Controller, Service, Repository). |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |
| --- |
| Transaction management   1. Programmatic transaction management 2. Declaration Transaction management 3. Advance questions on it |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |
| --- |
| Microservices design pattern in project   1. API gateway design pattern 2. Circuit breaker design pattern |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |
| --- |
| Exceptionmanagement |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |
| --- |
| Cache management   1. Default cache management 2. Custom cache management |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |
| --- |
| **Sonar & Sona type & Checkmarx** |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |
| --- |
| **Prod readiness and its related tools**   1. **Technical Architect** 2. **BA** 3. **Leads** 4. **Tester**    1. **Mannual testing**    2. **E2e testing**    3. **Sanity testing**    4. **Prod comparison testing**    5. **Automation testing**    6. **PT testing** 5. **Technical**    1. **Technical lead**    2. **Developers**    3. **Junit testing**    4. **Commit & push-mechanism and commands**    5. **Swagger/open api and its alternatives** 6. **DevOps**    1. **Pipeline**    2. **Jenking**    3. **IKP**    4. **Docker**    5. **Kubernates**    6. **Helm,CICD**    7. **Load balancer**    8. **Cluster** 7. **Prod readiness**    1. **FQDN**    2. **Production configmap configuration**    3. **Code quality**    4. **Code review**    5. **Golive**    6. **GONOGO**    7. **Product engineer**    8. **RunBook(technical and business)**    9. **Technical live & Business live**    10. **Post live—KT tech leads, BA, Architect**    11. **SignOff(BA,Architect,Tester,Channle)**    12. **PVT** |