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

**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. |

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

|  |
| --- |
| **DB related all implementation**   * 1. **Connection pool**   2. **JPA**   3. **Hibernate**   4. **DB types**   5. **Query optimization**   6. **SQL query interview and project level** |

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

|  |
| --- |
| **Annotation level all implementation in project**   1. **@Retryable** 2. **@Recover** 3. **@Bean** 4. **@Controller** 5. **@Qualifier** 6. **@Resource** 7. **@PathVariable** 8. **@QueryParam** 9. **@Query** 10. **@ControllerAdvice** 11. **@Services** 12. **@repository** 13. **@Asynch** 14. **@Synch** 15. **@Autowired** |

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