# 🚀 Building a Kafka-Integrated Spring Boot Application: A Step-by-Step Guide

## 🔷 Step 1: Install Apache Kafka

To run Kafka locally, you'll need both **Kafka** and **Zookeeper**.

1. **Download Kafka** from Apache Kafka's website.
2. **Start Zookeeper** (needed by Kafka for managing brokers):

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

3. **Start Kafka Server**:

```
bin/kafka-server-start.sh config/server.properties
```

Kafka will run on **localhost:9092** by default, and Zookeeper on **localhost:2181**.

---

## 🔶 Step 2: Set Up Spring Boot Project with Kafka Dependencies

Create a Spring Boot project using Spring Initializr or your preferred IDE, and add the **Kafka dependency** to the `pom.xml` file.

```
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
```

---

## 🔷 Step 3: Configure Kafka Properties

In `application.yml` or `application.properties`, configure Kafka by setting the bootstrap server, serializers, and deserializers.

```yaml
spring:
  kafka:
    bootstrap-servers: localhost:9092
    consumer:
      group-id: my-group
      auto-offset-reset: earliest
      key-deserializer:
org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer:
org.apache.kafka.common.serialization.StringDeserializer
    producer:
      key-serializer:
org.apache.kafka.common.serialization.StringSerializer
      value-serializer:
org.apache.kafka.common.serialization.StringSerializer
```

### 🔶 Step 4: Create Kafka Producer

The **Producer** will send messages to a specified Kafka topic. Here's a simple producer service with a method to publish messages:

```java
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Service;

@Service
public class KafkaProducerService {

    private static final Logger logger =
```

```
LoggerFactory.getLogger(KafkaProducerService.class);
    private static final String TOPIC = "my-topic"; // Replace with
your topic

    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    public void sendMessage(String message) {
        logger.info("Sending message: {}", message);
        kafkaTemplate.send(TOPIC, message);
    }
}
```

### 🔷 Step 5: Create Kafka Consumer

The **Consumer** will listen to a specific topic and process incoming messages. In this case, we'll set up a listener method using `@KafkaListener`.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;

@Service
public class KafkaConsumerService {

    private static final Logger logger =
LoggerFactory.getLogger(KafkaConsumerService.class);

    @KafkaListener(topics = "my-topic", groupId = "my-group")
    public void listen(String message) {
```

```
        logger.info("Received message: {}", message);
    }
}
```

### 🔶 Step 6: Expose a REST API for Sending Messages

To make it easier to test the producer, create a simple controller that sends messages to Kafka using the producer service.

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/kafka")
public class MessageController {

    @Autowired
    private KafkaProducerService kafkaProducerService;

    @PostMapping("/publish")
    public ResponseEntity<String> publishMessage(@RequestParam String
message) {
        kafkaProducerService.sendMessage(message);
        return ResponseEntity.ok("Message published!");
    }
}
```

### 🔷 Step 7: Testing the Setup

1. **Start Kafka and Zookeeper** if not already running.
2. **Run the Spring Boot Application**.

To test the message flow:

- **Send a POST request** to `http://localhost:8080/api/kafka/publish` with a `message` parameter.
- **Observe the Kafka logs and the consumer logs** in your Spring Boot console to verify that the message is consumed.

You can use a tool like **Postman** or **cURL** to send a request:

```
curl -X POST "http://localhost:8080/api/kafka/publish?message=HelloKafka"
```

---

## 🔶 Step 8: Advanced Kafka Configurations

1. **Custom Topics**: By default, the producer sends to `"my-topic"`. You can create a custom topic in Kafka:

   ```
   bin/kafka-topics.sh --create --topic my-topic --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1
   ```

2. **Error Handling**: Implement `ErrorHandler` in your consumer to handle message processing failures gracefully.
3. **Message Retention**: Configure the retention policy for your Kafka topic in Kafka's server properties:

   ```
   properties
   Copy code
   log.retention.hours=168  # Retains messages for 7 days
   ```

---

## 🔷 Example of a Full Producer-Consumer Flow with Custom Configurations

Below is a more flexible setup that allows dynamic topics and error handling.

### Updated KafkaProducerService with Dynamic Topics

```
@Service
public class KafkaProducerService {

    private final KafkaTemplate<String, String> kafkaTemplate;

    @Autowired
    public KafkaProducerService(KafkaTemplate<String, String>
kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

    public void sendMessage(String topic, String message) {
        kafkaTemplate.send(topic, message).addCallback(
            success -> System.out.println("Message sent to topic " +
topic + ": " + message),
            failure -> System.err.println("Failed to send message: "
+ failure.getMessage())
        );
    }
}
```

### Updated KafkaConsumerService with Error Handling

```
@Service
public class KafkaConsumerService {

    private static final Logger logger =
LoggerFactory.getLogger(KafkaConsumerService.class);

    @KafkaListener(topics = "my-topic", groupId = "my-group")
```

```
    public void listen(String message) {
        try {
            logger.info("Received message: {}", message);
            // Process message (e.g., save to DB, etc.)
        } catch (Exception e) {
            logger.error("Failed to process message: {}", message,
e);
        }
    }
}
```

---

## 🔶 Key Takeaways

1. **Kafka** is ideal for high-throughput data streaming.
2. **Spring Boot** offers seamless integration with Kafka through `spring-kafka`.
3. Use **@KafkaListener** for consuming messages and **KafkaTemplate** for producing.
4. Configure properties in `application.yml` to control consumer group, deserialization, etc.