8

Exploring Event-Driven Systems with Kafka

In this chapter, we will delve into the mechanics of creating an event-driven system using Kafka and Spring Boot. Here, we'll discover how to configure Kafka and ZooKeeper on your computer using Docker, laying the foundation for developing microservices that can seamlessly communicate through events. You'll get hands-on experience with building two Spring Boot applications: one for generating events and the other for consuming them, simulating the functions of a sender and receiver in a messaging framework.

The ultimate aim of this chapter is to equip you with the skills to design, deploy, and monitor an **event-driven architecture (EDA)** that harnesses the capabilities of Kafka combined with the simplicity of Spring Boot. This knowledge is not crucial for your progress in this book's journey but invaluable in real-world scenarios where scalable and responsive systems are not just preferred but expected.

Mastering these principles and tools is essential for creating applications that are adaptable, scalable, and capable of meeting the evolving demands of contemporary software environments. By the conclusion of this chapter, you will have an event-driven setup on your local machine, boosting your confidence to tackle more complex systems.

The following are the main topics of this chapter that you'll explore:

- Introduction to event-driven architecture
- Setting up Kafka and ZooKeeper for local development
- Building an event-driven application with Spring Boot messaging
- Monitoring event-driven systems

Technical requirements

For this chapter, we are going to need to configure some settings on our local machines:

- **Java Development Kit 17 (JDK 17)**
- A modern **integrated development environment (IDE)**; I recommend IntelliJ IDEA

- You can clone all repositories related to *Chapter 8* from the GitHub repository here: https://github.com/PacktPublishing/Mastering-Spring-Boot-3.0/
- Docker Desktop

Introduction to event-driven architecture

Event-driven architecture, also known as **EDA**, is a design approach widely used in software development. It focuses more on triggering actions based on events than following a strict step-by-step process. In EDA, when a specific event occurs, the system reacts promptly by carrying out the action or series of actions. This method differs from models that rely on request-response patterns and offers a more dynamic and real-time system behavior.

EDA is significant in the era we're living in where data is constantly being generated and updated. The ability to promptly respond to changes is invaluable in such a fast-paced environment. EDA empowers businesses to seize opportunities and address challenges swiftly compared to conventional systems. This agility is particularly crucial in industries such as

finance, real-time analytics, the **Internet of Things** (**IoT**), and other areas where rapid changes occur frequently and the timeliness of information holds importance.

Moving to EDA can significantly change how a company functions, offering the following benefits:

- **Responsiveness**: By handling events in real time, event-driven systems offer immediate feedback or action, which is crucial for time-sensitive tasks.
- **Scalability**: Event-driven setups can manage a number of events without causing delays in processing. This scalability is important for businesses dealing with increasing data volume and complexity.
- **Flexibility**: As components in EDA are loosely connected, they can be updated or replaced independently without impacting the system. This flexibility makes upgrades and the integration of features simpler.
- **Efficiency**: Minimizing the need for checking for new data through polling or querying reduces resource consumption, improving overall system efficiency.
- **Enhanced user experience**: In applications requiring real-time information, such as gaming and live updates, EDA contributes to providing a dynamic user experience.

These benefits highlight why many organizations are moving toward EDA to meet the demands of modern technological challenges.

In EDA, we need a **message broker**. A message broker helps us to distribute the message between the components. In this chapter, we will use Apache Kafka as a message broker. Kafka is an open source stream-processing platform. It was initially developed by LinkedIn and later donated to the Apache Software Foundation. Kafka primarily functions as a message broker adept at handling substantial data volumes efficiently.

Its design features facilitate durable message storage and high-throughput event processing for effective EDA implementations. This platform allows distributed data streams to be consumed in time, making it an optimal solution for applications requiring extensive data-processing and transfer capabilities.

With Kafka, developers can seamlessly transfer data between components of an event-driven system, ensuring the preservation of event integrity and order even in complex transaction scenarios. This feature positions Kafka as a component in the architecture of many modern high-performance applications that rely on real-time data processing.

Now that we have a grasp of what EDA entails and the benefits it brings, along with understanding Kafka's role in such systems, we will go through the process of setting up Kafka on Docker. This setup creates a controlled and reproducible environment for the exploration of Kafka's capabilities within EDA. Our aim is to equip you with the tools and knowledge to deploy Kafka efficiently, enabling you to harness the potential of real-time data processing in your projects.

By mastering the deployment of Kafka using Docker, you will acquire the experience essential for comprehending and managing the intricacies of event-driven systems. This hands-on approach not only reinforces theoretical concepts but also readies you to effectively handle real-world applications.

Setting up Kafka and ZooKeeper for local development

Kafka plays a role in an event-driven system, facilitating smooth communication among different components. It enables services to communicate through message exchange, like how people use messaging apps to stay connected. This

architecture promotes the development of scalable applications by allowing various parts of the system to function autonomously and respond promptly to events. We will also mention Kafka and its role in the *Understanding Kafka brokers and their role in event-driven systems* section in more detail.

However, Kafka doesn't work alone; it collaborates with **ZooKeeper**, which serves as its overseer. ZooKeeper monitors Kafka's brokers to ensure they're functioning. Think of it as having a coordinator who assigns tasks and ensures operations. ZooKeeper is essential for managing the background processes that uphold Kafka's stability and reliability during peak loads.

After talking about the components we need, I will also mention the installation. We will use Docker as we did in previous chapters. Docker simplifies the setup of Kafka and ZooKeeper on your machine. It provides a portable version of the entire configuration that you can easily launch whenever needed, hassle-free.

This method of setting up Kafka and ZooKeeper isn't just for convenience; it's also about ensuring that you can explore, create, and test your event-driven systems without having to worry about intricate installation procedures or variations between setups. As we delve into the steps of setting up Kafka

and ZooKeeper using Docker, remember that this forms the groundwork. You're establishing an adaptable infrastructure for your applications—one that will facilitate effective communication and seamless scalability. Let's proceed and get your local development environment ready for EDA.

Understanding Kafka brokers and their role in event-driven systems

In the changing world of EDA, **Kafka brokers** serve as efficient hubs carefully managing the reception routing and delivery of messages to their designated destinations. Within the Kafka ecosystem, a Kafka broker plays a role as part of a group of brokers that work together to oversee message traffic. In simple terms, imagine these brokers as diligent postal workers handling the messages from producers and organizing them into topics similar to specific mailboxes or addresses. These topics can be divided into sections to facilitate scalable message processing.

Let's see how a Kafka cluster works in *Figure 8.1*.

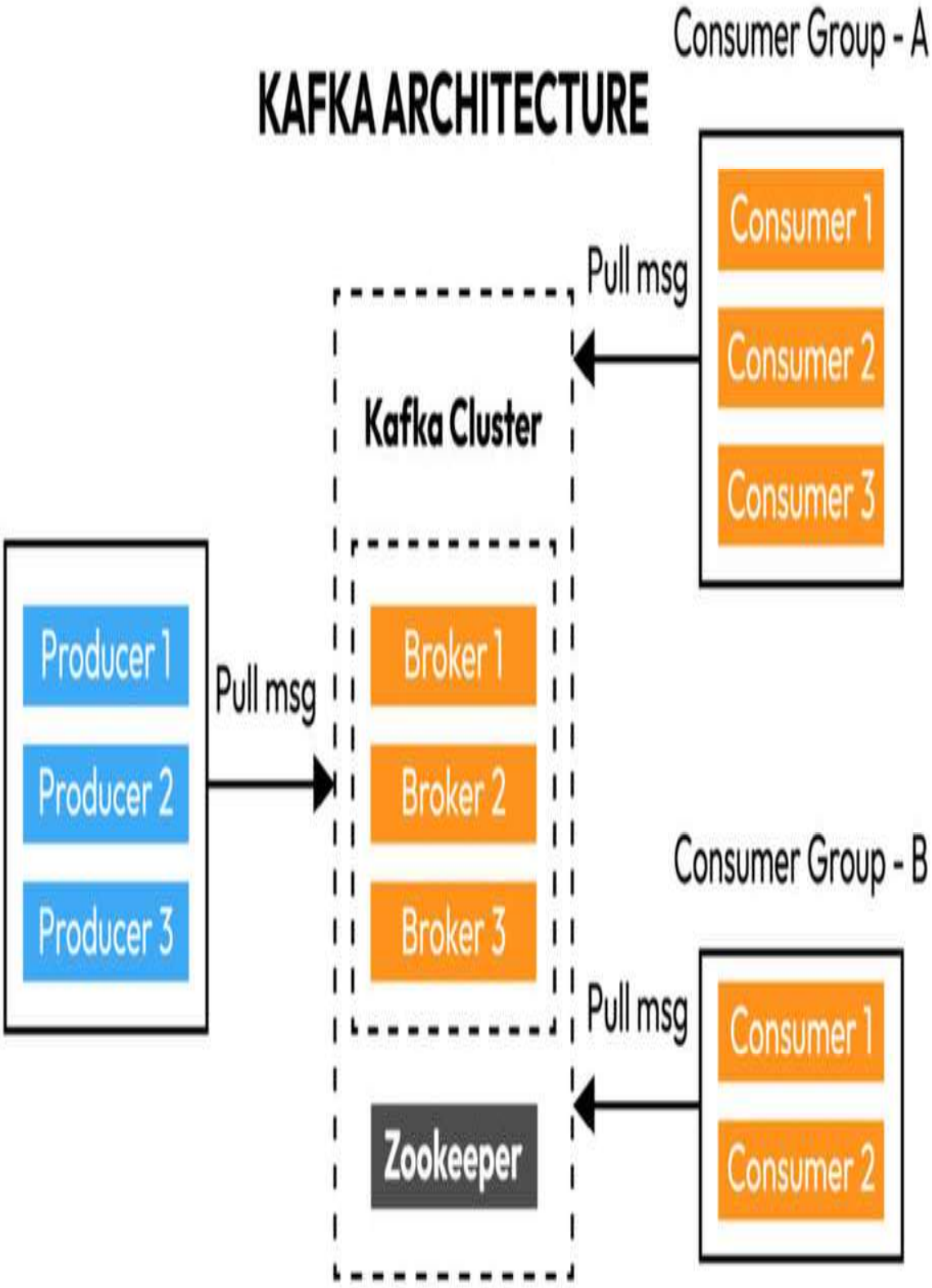
KAFKA ARCHITECTURE
Consumer Group - A
Consumer 1
Consumer 2
Consumer 3
Pull msg
Kafka Cluster
Producer 1
Producer 2
Producer 3
Pull msg
Broker 1
Broker 2
Broker 3
Consumer Group - B
Pull msg
Consumer 1
Consumer 2
Zookeeper

Figure 8.1: Kafka cluster architecture

In this diagram, you can see how Kafka organizes its workflow. Producers are the sources that send data to the Kafka system. They push messages into the Kafka cluster, which consists of multiple brokers (**Broker 1**, **Broker 2**, and **Broker 3**). These brokers store the messages and make them available for consumption. ZooKeeper acts as the manager of this cluster, keeping track of the state of brokers and performing other coordination tasks. Consumer groups, labeled **Group-A** and **Group-B**, pull messages from the brokers depending on their needs.

The true magic of Kafka brokers lies in their adeptness at managing these topic sections. When a message arrives, the broker determines where to place it within a section based on criteria such as its importance level. This method ensures a distribution of messages and groups similar ones (those sharing common attributes) in one section. This partitioning process is essential for distributing workloads and enables consumer applications to process messages concurrently for more streamlined data handling.

Furthermore, another critical function of Kafka brokers is ensuring message duplication across the Kafka system,

safeguarding against data loss in case of broker malfunctions. This duplication process acts as a safety measure by creating copies of sections across different brokers. If a broker goes offline, another can step in, smoothly keeping the system strong and flexible.

Brokers are skilled at storing and providing messages for consumers. They use offsets to track which messages consumers have read, allowing consumers to resume right where they left off in the message stream. This ensures that every message is handled and gives consumers the flexibility to manage messages at their own pace.

The orchestration of messages in a Kafka cluster, overseen by brokers, is a process that combines efficiency with reliability. This intricate coordination carried out by brokers enables event-driven systems to function efficiently, managing large amounts of data with precision. By utilizing the features of Kafka brokers, developers can create systems that are not only scalable and resilient but also capable of processing messages swiftly and accurately to meet the demands of today's fast-paced digital landscape.

As we further explore the aspects of setting up and using Kafka, the role of brokers as the foundation for reliable and efficient

message distribution becomes increasingly clear. Their ability to handle and direct messages serves as the core of any EDA, ensuring that information is delivered accurately to its intended destination on time.

Running Kafka and ZooKeeper with Docker

Running Kafka and ZooKeeper on your computer through Docker can be a game-changer for developers. It streamlines what was once a setup process into something simple and easy to handle. Docker containers serve as transportable spaces that can be swiftly initiated, halted, and deleted, making them ideal for development and testing purposes. This arrangement enables you to recreate a production-level environment on your machine without the need for setup or specialized hardware.

You will be familiar with Docker Compose since we have used it in almost all the previous chapters. We will use Docker Compose to run both services with a single command. Here's a simple **`docker-compose.yml`** file example that sets up Kafka and ZooKeeper:

```
version: '2'
services:
  zookeeper:
    image: zookeeper
    ports:
      - "2181:2181"
    networks:
      - kafka-network
  kafka:
    image: confluentinc/cp-kafka
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
    environment:
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS:
PLAINTEXT://localhost:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    networks:
      - kafka-network
networks:
  kafka-network:
    driver: bridge
```

The **`docker-compose.yml`** file is like a recipe that tells Docker exactly how to run your Kafka and ZooKeeper containers. It

tells Docker which images to use, how the containers should talk to each other on a network, which ports to open, and what environment variables to set. In this file, we have told Docker to run ZooKeeper on port `2181` and Kafka on port `9092`. Using this file, we streamline the whole process, making it as easy as pressing a button to get your setup running. It's a brilliant tool for developers, cutting down on the manual steps and letting you focus on the fun part—building and experimenting with your event-driven applications.

Save this file as `docker-compose.yml` and run it using this command:

```
docker-compose up -d
```

This command pulls the necessary Docker images, creates the containers, and starts Kafka and ZooKeeper in detached mode, leaving them running in the background.

By following these steps, you've just set up a robust, scalable messaging backbone for your applications to build upon. This foundation not only supports the development of event-driven systems but also paves the way for experimenting with Kafka's powerful features in a controlled local environment.

Finishing our exploration of configuring Kafka using Docker, it's evident how this pairing removes the obstacles in running Kafka on your computer. Docker's container magic has turned what might have been a laborious task into a straightforward process, allowing you to concentrate more on the creative aspects of developing applications rather than getting caught up in setup intricacies. This simplified setup isn't only about convenience; it's also about democratizing technology and simplifying its management, empowering developers to experiment and innovate with EDA without dealing with overly complicated configurations.

As we shift from the aspects of setting up Kafka and ZooKeeper to delving into the exciting realm of constructing an event-driven application using Spring Boot messaging, we're transitioning from laying the groundwork for infrastructure to engaging in the artistry of application design. In this section, you'll witness firsthand how your Kafka setup empowers you as we walk you through the creation of applications that generate and consume messages with Spring Boot. This is where abstract concepts materialize into creations allowing you to fully leverage the capabilities of event-driven systems.

Building an event-driven application with Spring Boot messaging

Crafting an event-driven application using Spring Boot involves building a system that's responsive, scalable, and equipped to handle the complexities of modern software requirements. Essentially, an event-driven application responds to events, ranging from user interactions to messages from external systems. This methodology enables components of your application to interact and operate independently, enhancing flexibility and efficiency. With Spring Boot, setting up such an application is made easier due to its philosophy of convention over configuration and the array of tools it provides from the start.

Throughout this journey, we will take a hands-on approach by introducing two Spring Boot projects—one will focus on generating events while the other will concentrate on consuming them. This segregation mirrors real-life scenarios where producers and consumers are often located in systems or microservices highlighting the decentralized nature of contemporary applications. By working on these projects, you

will gain experience in configuring a producer for sending messages and a consumer for reacting to those messages, within the context of Spring Boot and Kafka. This method not only strengthens your comprehension of event-driven systems but also equips you with the resources needed to create and enhance your own scalable applications.

As we move forward, we'll dive into the details of creating a Spring Boot project for Kafka integration. This will establish the foundation for our event-based applications, walking you through the process of configuring a Spring Boot project to send and receive messages using Kafka. You'll gain insights into the settings, libraries, and initial code structures required to kick start the implementation. Here is where our theoretical ideas transform into executable code. So, let's get started and embark on this journey of developing robust interactive applications with Spring Boot and Kafka.

Creating a Spring Boot project for Kafka integration

Starting a project in Spring Boot that is specifically tailored for integrating with Kafka is the practical step toward unlocking the capabilities of event-driven applications. This step combines the ease and adaptability of Spring Boot with the messaging

features of Kafka, allowing developers to create scalable and agile applications. Through this integration, we are establishing a base that facilitates communication and the management of large data volumes and operations in a distributed setting. The objective is to establish a framework that addresses message production and consumption requirements while also seamlessly expanding as the application evolves.

We will need two different projects to demonstrate the consumer and producer. So, you will need to follow the same steps twice to create the two projects. But it would be better to choose a different name when entering the project metadata in *step 2*.

In *Figure 8.2*, we can see how our applications will communicate with each other.

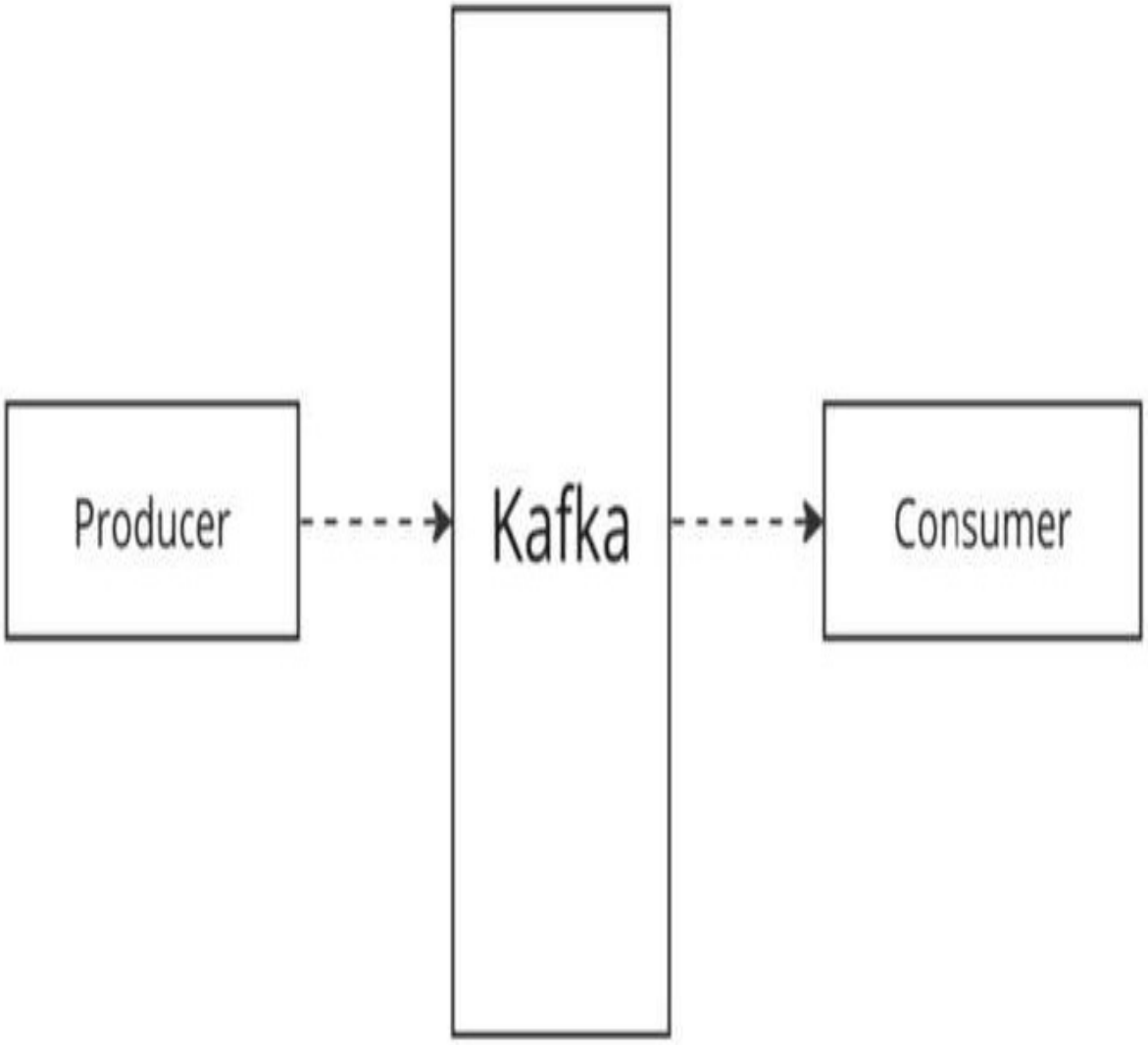

Figure 8.2: How our apps communicate with each other

As you can see in *Figure 8.2*, there is no direct call between the producer application and the consumer application. The producer application sends a message to Kafka and Kafka publishes this message to the consumer application.

Here is a step-by-step guide to creating a Spring Boot project:

1. Navigate to Spring Initializr (https://start.spring.io/) to bootstrap your project. It's an online tool that lets you generate a Spring Boot project with your chosen dependencies quickly.
2. Enter your project's metadata, such as **Group**, **Artifact**, and **Description**. Give different names for consumer and producer projects. Choose either **Maven** or **Gradle** as your build tool according to your preference. In our example, we will use **Gradle**.
3. Select your dependencies. For a Kafka project, you need to add **Spring for Apache Kafka** under the **Messaging** category. We need to add `Spring Web` for the producer project. This dependency includes the necessary libraries to integrate Kafka with Spring Boot.
4. Generate the project. Once you've filled in all the details and selected your dependencies, click on **Generate** to download your project template.
   In *Figure 8.3*, we can see which dependencies we need and how to configure Spring Initializr.

Figure 8.3: Screenshot of Spring Initialzr

5. Extract the downloaded ZIP file and open the project in your favorite IDE, such as IntelliJ IDEA, Eclipse, or VS Code.

6. Update the **`application.properties`** file using the following line. Use different ports for consumer and publisher projects:

```
server.port:8181
```

When integrating Kafka with a Spring Boot project, a key component is **Spring Kafka**, which is added by Spring Initializr as Spring for Apache Kafka. This library simplifies the handling of messaging solutions based on Kafka by providing a user abstraction. It streamlines the process of sending and receiving messages between your Spring Boot application and Kafka brokers. By abstracting the complexities of producer and consumer configurations, it enables you to focus on implementing business logic rather than dealing with repetitive code for message handling.

With your Spring Boot project configured and essential Kafka integration dependencies in place, you are now ready to delve into the details of producing and consuming messages. This setup serves as a starting point for exploring communication and EDAs, offering an effective approach to managing data flow in your applications.

Moving on to building the producer application in the next subsection marks a shift from setup to implementation. Here, we will guide you through setting up a Kafka producer within your Spring Boot project. This is where all your foundational work begins to take shape, allowing you to send messages to Kafka topics and kickstart the communication process for any event-driven system. Get ready to translate theory into action and witness how your application can engage with Kafka.

Building the producer application

Creating the producer application is like establishing a broadcasting hub within your event-based framework, where your Spring Boot setup is all set to dispatch messages out to the world—or, precisely, to a Kafka topic. This stage holds importance as it marks the beginning of information flow within your system, ensuring that data reaches its intended destination at the right moment.

Creating a Kafka producer in Spring Boot involves a few straightforward steps. First, you need to configure your application to connect to Kafka. This is done in the `application.properties` file in your producer Spring Boot project. You'll specify details such as the Kafka server's address and the default topic to which you want to send messages.

Here’s how we will implement a Kafka producer in a Spring Boot application:

```
@RestController
public class EventProducerController {
    private final KafkaTemplate<String, String>
kafkaTemplate;
    @Autowired
    public
EventProducerController(KafkaTemplate<String,
String> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }
    @GetMapping("/message/{message}")
    public String trigger(@PathVariable String
message) {
        kafkaTemplate.send("messageTopic",
message);
        return "Hello, Your message has been
published: " + message;
    }
}
```

In this code, **`KafkaTemplate`** is a Spring-provided class that simplifies sending messages to a Kafka topic. We inject this template into our **`MessageProducer`** service and use its **`send`**

method to publish messages. The **`send`** method takes two parameters—the name of the topic and the message itself.

To ensure your producer application can successfully send messages to Kafka, you'll need to add some configurations to your **`application.properties`** file:

```
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.producer.key-
serializer=org.apache.kafka.common.serialization.Stri

spring.kafka.producer.value-
serializer=org.apache.kafka.common.serialization.Stri
```

These configurations help Spring Boot identify the location of your Kafka server (Bootstrap servers) and how to convert the messages into a format for transmission over the network (key serializer and value serializer). Serialization involves converting your message, in this case, a string, into a format that can be transmitted over the network.

By setting up and configuring your Kafka producer, you have taken a step toward developing an event-driven application. This configuration allows your application to initiate conversations within your distributed system by sending out

messages that other parts of your system can respond to and handle.

Moving forward, let's shift our focus to the counterpart of this interaction: building the consumer application. This involves creating listeners that anticipate and react to messages dispatched by our producer. It plays a role in closing the communication loop within our EDA, transforming our system into a dynamic network of services capable of responding to real-time data. Let's proceed with our exploration and uncover how we can unleash the potential of event-driven applications.

Building the consumer application

Once we've got our broadcasting station set up using our producer application, it's time to tune to the correct frequency by developing the consumer application. This step ensures that the messages sent out by the producer aren't just lost in space but are actually received, understood, and put into action. In our event-driven structure, the consumer application acts like a listener in a crowd catching signals meant for it and handling them accordingly. By incorporating a Kafka consumer into a Spring Boot application, we establish an element that eagerly waits for messages and is prepared to process them as soon as they come through. This ability plays a role in creating systems

that are truly interactive and can respond promptly to changes and events in real time.

To set up a Kafka consumer in Spring Boot, you first need to configure your application to listen to the Kafka topics of interest. This involves specifying in the **`application.properties`** file where your Kafka server is located and which topics your application should subscribe to.

Here's how we will implement a Kafka consumer in our Spring Boot application:

```
import
org.springframework.kafka.annotation.KafkaListener;

import org.springframework.stereotype.Component;
@Component
public class MessageConsumer {
    @KafkaListener(topics = "messageTopic",
groupId = "consumer_1_id")
    public void listen(String message) {
        System.out.println("Received message: " +
message);
    }
}
```

In this snippet, the `@KafkaListener` annotation marks the `listen` method as a listener for messages on `messageTopic`. The `groupId` is used by Kafka to group consumers that should be considered as a single unit. This setup allows your application to automatically pick up and process messages from the specified topic.

To make sure your consumer application consumes messages efficiently, add the following configurations to your `application.properties` file:

```
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.consumer.group-id= consumer_1_id

spring.kafka.consumer.auto-offset-reset=earliest
spring.kafka.consumer.key-
deserializer=org.apache.kafka.common.serialization.St

spring.kafka.consumer.value-
deserializer=org.apache.kafka.common.serialization.St
```

These configurations make sure your user connects to the Kafka server (Bootstrap servers) and properly decodes the messages it receives (key deserializer and value deserializer). The `auto-offset-reset` option guides Kafka on where to

begin reading messages if there's no offset for your users group; by setting it to `earliest`, our application will start to consume from the beginning of the event topic.

Once your consumer application is active, your event-driven system is now fully operational, capable of both sending and receiving messages through the Kafka messaging pipeline. This two-way communication framework lays the foundation for scalable applications that can handle real-time data streams and respond promptly to events as they occur.

Looking forward, the next critical step involves testing both the producer and consumer applications to ensure their integration. This phase bridges theory with practice, allowing you to witness the outcomes of your efforts. Testing serves not only to verify individual component functionalities but also to validate the overall responsiveness and efficiency of the system. Let's progress by initiating tests on our event-driven applications, ensuring they're primed to manage any challenges that may arise.

Testing the whole stack – bringing your event-driven architecture to life

After configuring our event-based system using Kafka, Spring Boot, and Docker, we reach a pivotal moment as we test the entire setup to witness our system in operation. This critical stage confirms that our separate elements, the producer and consumer applications, are properly set up and communicating as intended while also ensuring that Kafka, managed by Docker, effectively transmits messages between them. This testing phase represents the culmination of our work, allowing us to directly observe the dynamic exchange of messages that serves as the core of any event-driven system.

Here are the instructions to run the whole stack:

1. **Docker Compose for Kafka and ZooKeeper**: Begin by starting Kafka and ZooKeeper using Docker Compose. Navigate to the directory containing your `docker-compose.yml` file that defines the Kafka and ZooKeeper services and run the following:

   ```
   docker-compose up -d
   ```

   This command starts Kafka and ZooKeeper in detached mode, setting the stage for message brokering.
2. **Running the producer application**: Launch your producer Spring Boot application, ensuring it runs on port `8282`. This

can be configured in the **`application.properties`** file with the following line:

```
server.port=8282
```

Start the application through your IDE or by running **`./gradlew bootRun`** in the terminal within the project directory.

3. **Running the consumer Spring Boot application**: Similarly, launch the consumer application, configured to run on port **`8181`**, by setting this in its **`application.properties`** file:

```
server.port=8181
```

Use your IDE or the **`Gradle`** command as with the producer to start the consumer application.

4. **Trigger message publishing**: With both applications running, it's time to send messages. Use your web browser or a tool such as cURL to make **`GET`** requests to the producer's message-triggering endpoint:

```
http://localhost:8282/message/hello-world
```

Replace **`hello-world`** with any string you wish to send as a message. Trigger a few different messages to test various

scenarios.

5. **Observe the consumer's log**: Switch to the console or log output of your consumer application. You should see the messages logged as they are consumed, indicating successful communication from the producer, through Kafka, to the consumer. The output will be as follows:

```
Received message: hello-world
Received message: hello-world-2
Received message: hello-world-3
```

Successfully running the test stack and observing the flow of messages from the producer to the consumer via Kafka is an invaluable experience because it showcases the power and flexibility of EDAs. This hands-on testing not only increases your understanding of integrating Kafka with Spring Boot applications but also highlights the importance of seamless communication in distributed systems. As you've seen, Docker plays a pivotal role in simplifying the setup for development and testing environments. After this practical experience, you are ready to delve into sophisticated and scalable event-driven applications, which are requested in modern software developments.

Now, with a fully functional event-driven application in hand, it's time to look ahead. The next step is ensuring our application not only runs but succeeds under various conditions. This means diving into monitoring—a vital component of any application's life cycle. In the upcoming section, we'll explore how to keep a keen eye on our application's performance and how to swiftly address any issues that arise. This knowledge will help not only in maintaining the health of our application but also in optimizing its efficiency and reliability. So, let's move forward, ready to tackle these new challenges with confidence.

Monitoring event-driven systems

In the dynamic world of event-driven systems, where applications communicate through a constant flow of messages, monitoring plays a crucial role in ensuring everything runs smoothly. Just as a busy airport needs air traffic control to keep planes moving safely and efficiently, an EDA relies on monitoring to maintain the health and performance of its components. This oversight is vital for spotting when things go wrong and understanding the overall system behavior under various loads and conditions. It enables developers and

operations teams to make informed decisions, optimize performance, and prevent issues before they impact users.

For applications built with Kafka and Spring Boot, a robust set of monitoring tools and techniques is essential for keeping an eye on the system's pulse. At its core, Kafka is designed to handle high volumes of data, making monitoring aspects such as message throughput, broker health, and consumer lag imperative. Tools such as Apache Kafka's JMX metrics and external utilities such as Prometheus and Grafana offer deep insights into Kafka's performance. These tools can track everything from the number of messages being processed to the time it takes to travel through the system.

As monitoring the Spring Boot application was covered in the *Spring Boot Actuator with Prometheus and Grafana* section of *Chapter 7*, it won't be covered here. We will only focus on monitoring Kafka in this section.

Monitoring your Kafka infrastructure

Monitoring your Kafka setup is like using a tool to closely examine the core functions of your event-driven system. It's all about getting a view of how well your Kafka environment is

running, which is crucial for identifying problems, optimizing resource usage, and ensuring messages are delivered on time and reliably. Given Kafka's role in managing data streams and event processing, any issues or inefficiencies can impact the entire system. Therefore, establishing a monitoring system isn't just helpful; it's necessary for maintaining a strong and efficient architecture.

Here are the key metrics to monitor in Kafka:

- **Broker metrics**: These include the number of active brokers in your cluster and their health status. Monitoring the CPU, memory usage, and disk I/O of each broker helps in identifying resource bottlenecks.
- **Topic metrics**: Important metrics here include message in-rate, message out-rate, and the size of topics. Keeping an eye on these can help in understanding the flow of data and spotting any unusual patterns.
- **Consumer metrics**: Consumer lag, which indicates how far behind a consumer group is in processing messages, is critical for ensuring data is processed in a timely manner. Additionally, monitoring the number of active consumers can help with detecting issues with consumer scalability and performance.

- **Producer metrics**: Monitoring the rate of produced messages, along with error rates, can highlight issues in data generation or submission to Kafka topics.

We will use Kafka Manager (now known as **CMAK**, or **Cluster Manager for Apache Kafka**) to monitor our Kafka server. Running CMAK in the same Docker Compose file as your Kafka and ZooKeeper setup is convenient for managing and monitoring your Kafka cluster locally.

Using CMAK to monitor the Kafka server

Here's how you can include CMAK in your Docker Compose setup and get it running on your local machine:

1. To include CMAK in your existing Docker Compose setup, you'll need to add a new service definition for it. Open your **`docker-compose.yml`** file and append the following service definition:

```
  kafka-manager:
    image: hlebalbau/kafka-manager:latest
    depends_on:
      - zookeeper
```

```
      - kafka
    ports:
      - "9000:9000"
    environment:
      ZK_HOSTS: zookeeper:2181
    networks:
      - kafka-network
```

We have simply introduced the **`kafka-manager`** image in our **`docker-compose.yml`** file—CMAK depends on ZooKeeper and Kafka since it needs to monitor their performance, and it will serve on port **`9000`**.

2. With your **`docker-compose.yml`** file updated, launch the services by running the following command in the terminal, in the directory containing your Docker Compose file:

```
docker compose up -d
```

This command pulls the necessary images and starts the ZooKeeper, Kafka, and Kafka Manager containers. The **`-d`** flag runs them in detached mode, so they'll run in the background.

3. Once all the services are up and running, open a web browser and go to **`http://localhost:9000`**. You should be greeted with the Kafka Manager (CMAK) interface.

To start monitoring your Kafka cluster with Kafka Manager, you'll need to add your cluster to the Kafka Manager UI.

4. Click on the **Add Cluster** button.
5. Fill in the cluster information. For **Cluster Zookeeper Hosts**, you can use `zookeeper:2181` if you're running everything locally, and use the default ZooKeeper setup from your Docker Compose file. Note that since Kafka Manager is running in the same Docker network created by Docker Compose, it can resolve the ZooKeeper hostname directly.

   In *Figure 8.4*, we can see how we can fill the form in by using the **Add Cluster** screen of Kafka Manager.

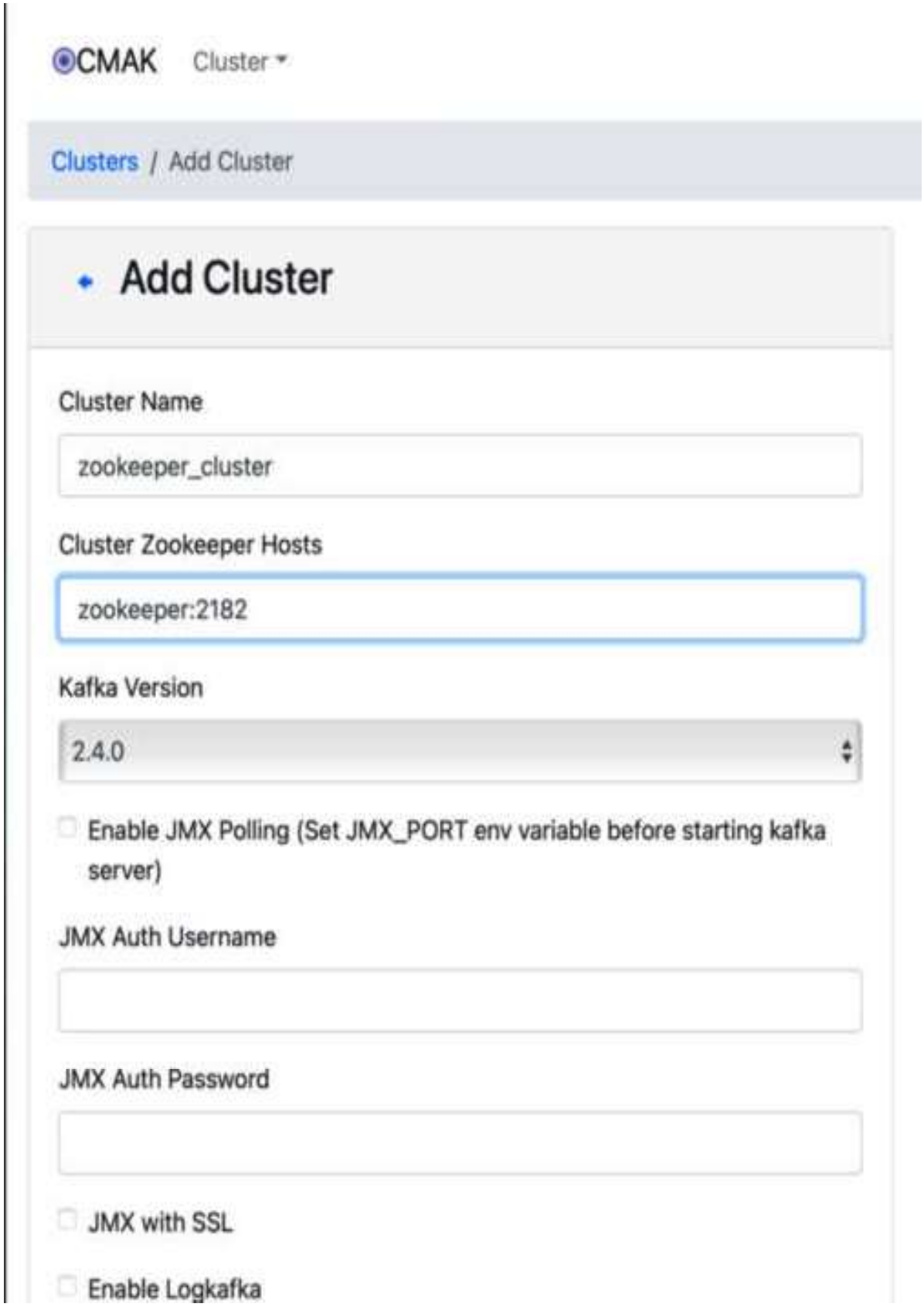
CMAK
Cluster
Clusters / Add Cluster
Add Cluster
Cluster Name
zookeeper_cluster
Cluster Zookeeper Hosts
zookeeper:2182
Kafka Version
2.4.0
Enable JMX Polling (Set JMX_PORT env variable before starting kafka server)
JMX Auth Username
JMX Auth Password
JMX with SSL
Enable Logkafka

Figure 8.4: Screenshot of the Add Cluster screen in the Kafka Manager application

6. Save your cluster configuration.

Now that your Kafka cluster is added to Kafka Manager, you can explore various metrics and configurations, such as topic creation, topic listing, and consumer groups.

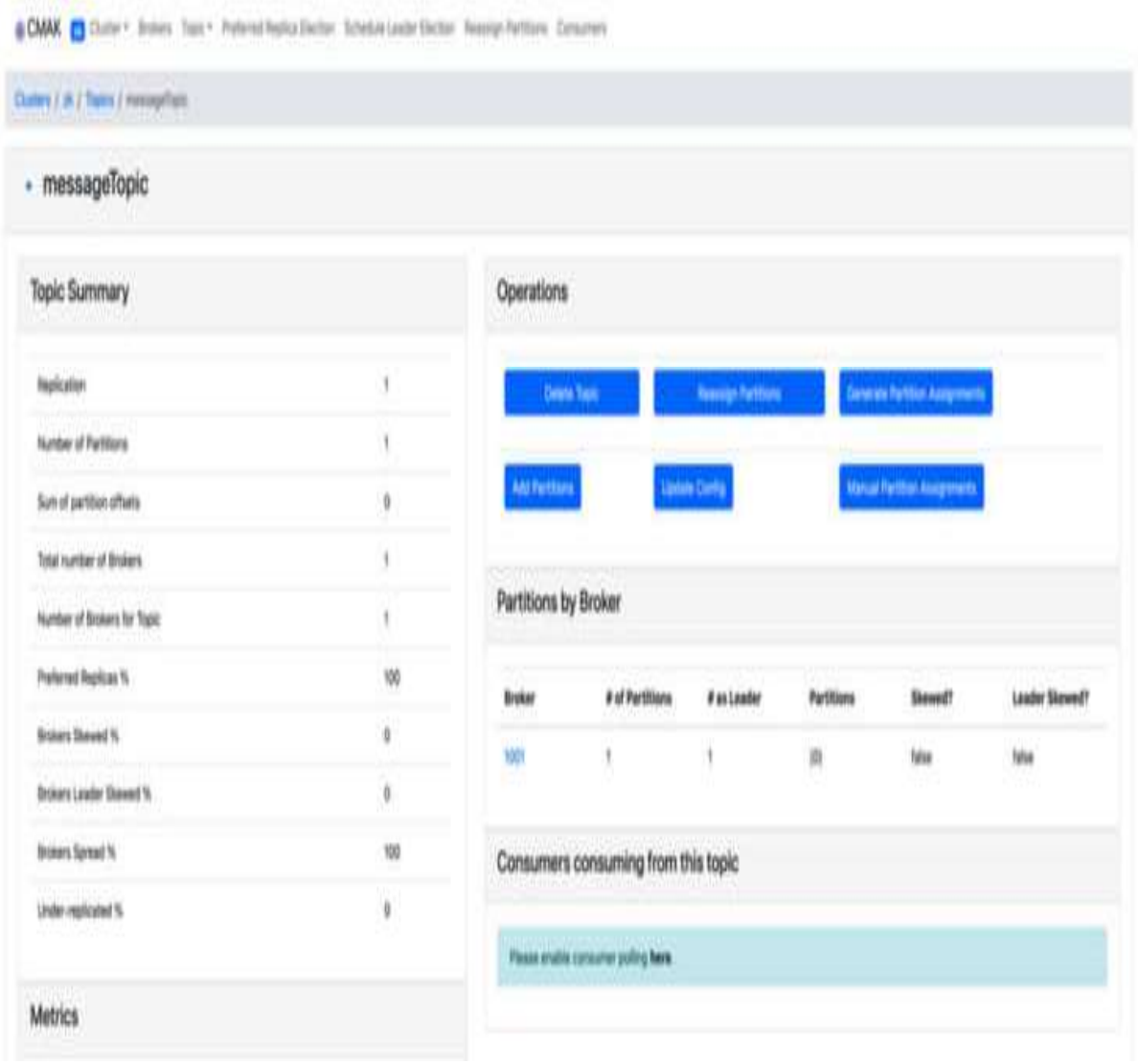

Figure 8.5: Kafka Manager screen for our topic

In *Figure 8.5*, you can see a screenshot of the CMAK dashboard, which gives information about a specific Kafka topic called **`messageTopic`**. The dashboard provides an overview including details on the topic's replication factor, the number of partitions, and the total sum of partition offsets representing the total message count in the topic. Additionally, it offers controls

to manage the topic, such as options to delete the topic, add partitions, or modify the topic's configuration. The dashboard also presents insights into how partitions are distributed across brokers with metrics such as **Preferred Replicas %** and flags any skewed or under-replicated partitions, which are crucial for diagnosing and maintaining optimal health and balance within the Kafka cluster.

This setup allows you to manage and monitor your Kafka cluster locally with ease, providing a powerful interface for handling Kafka configurations and observing cluster performance.

Implementing a monitoring strategy that covers these key metrics and leveraging tools such as Kafka Manager can help you better understand your Kafka infrastructure. This not only aids in proactive maintenance and optimization but also prepares you to react swiftly and effectively to any issues that arise.

In a nutshell, effectively monitoring Kafka is essential for an event-driven system. It's important to keep an eye on key metrics such as broker health, partition balance, message flow, and consumer lag. Tools such as CMAK, Prometheus, and Grafana not only simplify these tasks but also provide in-depth

visibility and analysis to turn raw data into actionable insights. By monitoring, potential issues can be spotted and addressed before they become major problems, ensuring the smooth operation of the Kafka messaging pipeline.

A monitored event-driven system is equipped to handle the complexities of modern data streams and workload requirements. It ensures that every part of the system functions reliably, maintaining the performance needed for today's applications. Ultimately, the strength of the systems lies in paying attention to operational details—where monitoring isn't just a routine but a vital aspect of system health and longevity.

Summary

As we wrap up this chapter, let's take a moment to look back on the journey we've shared. We've dived into the world of Kafka and Spring Boot, putting together each piece of our event-driven system. Here's what we accomplished:

- **Setting up Kafka and ZooKeeper**: We set up Kafka and ZooKeeper on our local machines using Docker, creating a robust backbone for our messaging system.
- **Building Spring Boot applications**: We built two Spring Boot applications from scratch, one as an event producer

and the other as a consumer, learning how they work together to form a responsive EDA.
- **Monitoring the Kafka infrastructure**: We learned the importance of monitoring our Kafka infrastructure, using tools such as CMAK to keep a watchful eye on the health and performance of our system.

The insights explored in this chapter aren't just theoretical; they translate into abilities that you can promptly utilize in real-world scenarios. These competencies are essential for ensuring your systems function and remain resilient, empowering them to adapt to the ever-changing data landscape with agility. The capability to set up, integrate, and manage systems is indispensable in today's rapidly evolving tech arena.

By continuing your learning journey with us, you're not just acquiring tools for your skillset; you're improving your development workflow, making it more seamless and effective. You're also strengthening the durability and manageability of your applications, providing an edge in the competitive technology sector.

As we move forward to the next chapter, we'll delve into the details of advanced Spring Boot features that enhance your development process. You'll discover the art of aspect-oriented

programming for organizing code, leverage the Feign client for seamless HTTP API integration, and harness the capabilities of Spring Boot's sophisticated auto-configuration features. The next chapter focuses on simplifying your tasks as a developer, making them more efficient and productive. Let's move ahead together and expand our knowledge further.