Apache Kafka

List of Contents

Introduction

Apache Kafka is a publish-subscribe based fault tolerant messaging system. It is fast, scalable and distributed by design. It was initially thought of as a message queue and open-sourced by LinkedIn in 2011. Kafka works by organizing data into topics, which are similar to channels or queues in a messaging system. Producers write data to topics and consumers read from topics. Its community evolved Kafka to provide key capabilities:

- Publish and Subscribe to streams of records, like a message queue.
- Storage system so messages can be consumed asynchronously.
- Kafka writes data to a scalable disk structure and replicates for fault tolerance. Producers can wait to write acknowledgements.
- Stream processing with Kafka Streams API enables complex aggregations or joins of input streams onto an output stream of processed data.

How Kafka Works

Kafka works by having producers write data to a topic, which is a specific category or stream of data, and consumers read from that topic. Topics are partitioned and replicated across multiple Kafka brokers, which are servers that run Kafka. This allows for horizontal scalability and fault tolerance.

Data is written to a topic in the form of records, which consist of a key, a value, and a timestamp. Producers write records to a specific partition within a topic, and each partition is ordered so that records are written in the order they were received. Consumers read records from a partition in the order they were written.

Kafka also has the ability to retain records for a specified amount of time, allowing for real-time data processing and analysis.

In summary, Apache Kafka is a distributed streaming platform that allows for handling real-time data feeds, it is a fault-tolerant system by partitioning and replicating topics across multiple brokers, the data is written in the form of records which consist of key, value, and timestamp and the records are written in order it was received. Consumers read records in the order they were written and kafka retains records for a specified amount of time.

Advantages Over REST API

- Loose Coupling - Neither service knows about each other regarding data update.
- Durability - Guarantees that the message will be delivered even if the consumer is down. Whenever the consumer gets up again, all messages will be there.
- Scalability - Since the messages get stored in a bucket, there is no need to wait for responses. We create asynchronous communication between all services.
- Flexibility - The sender of the message has no idea who is going to consume it. Meaning we can easily add new consumers with less work.

ActiveMQ/RabbitMQ vs Kafka

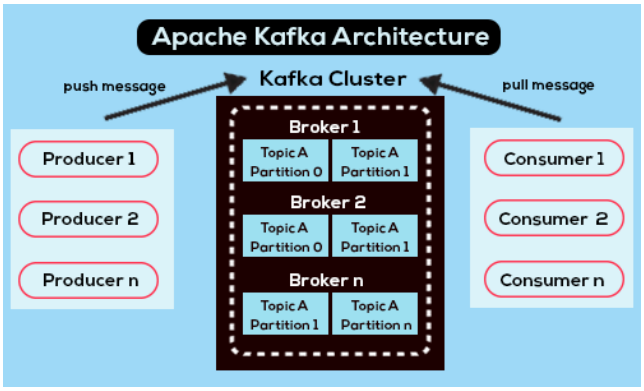
ActiveMQ/RabbitMQ	Kafka
P2P messaging system.	Publish-Subscribe. Kafka allows publishing and subscribing to streams of records.
Traditional messaging system. Deals with a small amount of data.	Distributed publish-subscribe messaging and logging system with message persistence capability.
One or more consumers are connected to the queue while the broker uses the round robin approach to direct messages to specific consumers.	Producers push event streams to the brokers, and consumers pull the data from brokers.
Lower throughput since each delivery message state is maintained.	Higher throughput since producers don’t wait for acknowledgments from brokers. So brokers can write messages at a very high rate.
Cannot assure the order of messages to remain the same.	Can assure that the messages retain the order in which they are sent. Order of message is important in

	logging/monitoring services.
Push type messaging platform wherein the providers push the messages to consumers.	Pull type messaging platform wherein the consumers pull the messages from the brokers.
Responsibility of producers to ensure delivery of messages. Dumb consumers, smart queues.	Responsibility of consumers to consume messages. Dumb brokers, smart consumers.
No chances of horizontal scalability and replication.	Scalable and available because of replication of partitions.
Performance slows down as the number of consumers starts increasing. 4K-10K messages per second.	Performance remains effective and good even if there are newer consumers being added. 1M messages per second.
Supports both synchronous and asynchronous.	Supports asynchronous.
Acknowledgement based. Messages are removed from the queue once they are processed and acknowledged.	Policy based. Messages are always forever saved until the retention time expires.

Architecture & Key Terminologies

Apache Kafka has several key terminologies that are important to understand in order to effectively use the platform. Here is a brief overview of some of the key terminologies:

- **Topic**: A topic is a specific category or stream of data in Kafka. Producers write data to topics and consumers read data from topics.
- **Broker**: A broker is a Kafka server that stores and manages the data in topics. A Kafka cluster typically consists of multiple brokers for fault tolerance and scalability.
- **Producer**: A producer is an application or process that writes data to a topic. Producers send data to a broker, which then stores the data in the topic.
- **Consumer**: A consumer is an application or process that reads data



from a topic. Consumers read data from a broker, which retrieves the data from the topic.

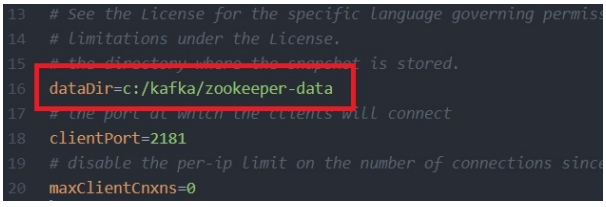
- **Partition**: A partition is a unit of storage in a topic. Topics are divided into partitions to allow for parallelism in the storage and retrieval of data.
- **Offset**: An offset is a unique identifier for a message in a partition. It is used to track the position of a consumer in a partition and to ensure that messages are read in the correct order.
- **Replication Factor**: Replication factor is the number of replicas of a topic that are stored across the cluster. It ensures that data is not lost even if a broker goes down.
- **Cluster**: A cluster is a group of brokers that work together to store and manage data in topics.
- **Zookeeper**: Zookeeper is a distributed coordination service that is used by Kafka to manage its cluster. It is responsible for maintaining the state of the cluster and for electing a leader for partitions.
- **Connectors**: Kafka Connect is a tool for scalable and reliable streaming data between Apache Kafka and other systems. It's used to import and export data to/from Kafka.
- **Streams**: Kafka Streams is a client library for building applications and microservices, where the input and output data are stored in Kafka clusters.

Installation & Configuration

Apache Kafka can be downloaded from: https://kafka.apache.org/downloads

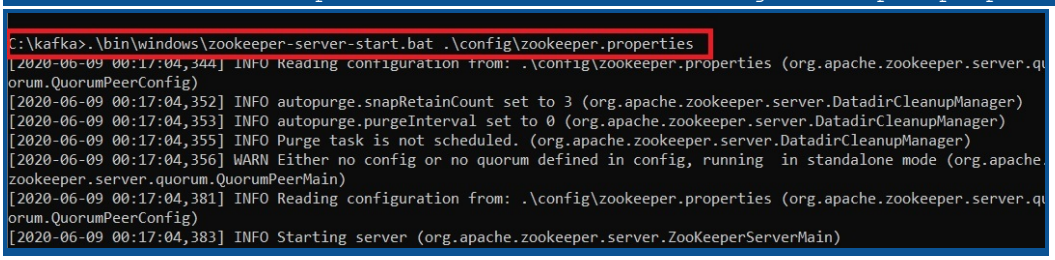
For the installation process, follow the steps given below:

- Go to the Downloads folder and select the downloaded Binary file.
- Extract the file and move the extracted folder to the directory where you wish to keep the files.
- Copy the path of the Kafka folder. Now go to config inside the kafka folder and open `zookeeper.properties` file. Copy the path against the field `dataDir` and add `/zookeeper-data` to the path.



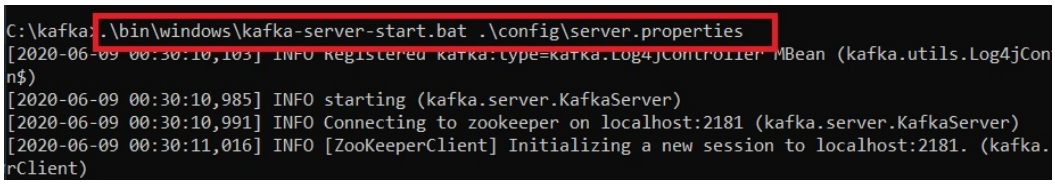
- Now in the same folder config open `server.properties` and scroll down to log.dirs and paste the path. To the path add `/kafka-logs`
- This completes the configuration of zookeeper and kafka server. Now open the command prompt and change the directory to the kafka folder. First start zookeeper using the command given below:

```
.\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties
```



- Now open another command prompt and change the directory to the kafka folder. Run kafka server using the command:

```
.\bin\windows\kafka-server-start.bat .\config\server.properties
```



Now kafka is running and ready to stream data.

Verify Installation

The default port that Apache Kafka runs on is port 9092. When we start a Kafka broker, it will listen on port 9092 for incoming connections from producers and consumers. If you need to change the port that Kafka uses, you can do so by editing the `server.properties` configuration file and updating the listeners property.

Creating a Topic

To create a new topic, use the following command:

```
"bin/kafka-topics.sh --create --bootstrap-server localhost:9092
--replication-factor 1 --partitions 1 --topic <topic_name>"
```

Replace `"topic_name"` with the desired name of your topic.

Producing a Message

To write messages to a topic, we need a producer.

Use the following command to start a producer that will write messages to the topic:

```
"bin/kafka-console-producer.sh --broker-list localhost:9092 --topic
<topic_name>"
```

Type in the messages you want to send and press enter to send them.

Consuming a Message

To read messages from a topic, we need a consumer. Use the following command to start a consumer that will read messages from the topic:

```
"bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic
<topic_name> --from-beginning"
```

The consumer will display the messages that are written to the topic in real-time.

Simple Producer Consumer Example Using Spring Boot Application

We will create an application where we can purchase a product. When purchased we'll create a purchase record and publish a message to kafka. A consumer will decrement the product inventory and send an email to the user.

First, add the following dependencies to your Spring Boot project's `pom.xml` file:

```
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

Let's add the following configuration in the `application.yml`:

```
spring:
 kafka:
   bootstrap-servers: localhost:9092
   consumer:
     group-id: inventory-group
     auto-offset-reset: earliest
     enable-auto-commit: false
     key-serializer: org.apache.kafka.common.serialization.StringSerializer
     value-serializer:
org.springframework.kafka.support.serializer.JsonSerializer
   producer:
     key-serializer: org.apache.kafka.common.serialization.StringSerializer
     value-serializer:
org.springframework.kafka.support.serializer.JsonSerializer
```

We'll now create a topic by using the `KafkaAdmin` class provided by Spring Kafka. We need to define the topic configuration in the `application.yml` file and create a `KafkaAdmin` bean in the configuration class.

```
@Configuration
public class KafkaTopicConfig {

   @Value("${kafka.topic.name}")
   private String topicName;

   @Bean
   public NewTopic topic() {
```

```
        return new NewTopic(topicName, 1, (short) 1);
    }

    @Bean
    public KafkaAdmin kafkaAdmin() {
        Map<String, Object> configs = new HashMap<>();
        configs.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:9092");
        return new KafkaAdmin(configs);
    }
}
```

Add the topic name in `application.yml`:

```
kafka:
 topic:
   name: product_purchased
```

Create a `Product` domain:

```
public class Product {
    private Long id;
    private String category;
    private String name;
    private Integer quantity;
    // getters and setters
}
```

Create a Purchase domain:

```
public class Purchase {
    private Long id;
    private Product product;
    private int quantity;
    private LocalDateTime purchasedAt;
    // getters and setters
}
```

Create `Product` and `Purchase` repository classes respectively:

```
@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {
}

@Repository
public interface PurchaseRepository extends JpaRepository<Purchase, Long> {
}
```

Write a REST API for purchasing product in `ProductController`:

```
@RestController
@RequestMapping("/api/products")
public class ProductController {
   private final ProductService productService;

   @Autowired
   public ProductController(ProductService productService) {
       this.productService = productService;
   }

   @PostMapping("/purchase")
   public ResponseEntity<String> purchaseProduct(@RequestBody Product product)
{
       productService.purchaseProduct(product);
       return new ResponseEntity<>("Product purchased successfully",
HttpStatus.OK);
   }
}
```

The `ProductService` will have the `purchaseProduct` method that will create a purchase record and publish a message to the kafka broker.

```
@Service
public class ProductService {

   private final ProductRepository productRepository;
   private final PurchaseRepository purchaseRepository;
   private final KafkaTemplate<Product, String> kafkaTemplate;

   @Autowired
   public ProductService(ProductRepository productRepository,
PurchaseRepository purchaseRepository,
                         KafkaTemplate<Product, String> kafkaTemplate) {
       this.productRepository = productRepository;
       this.purchaseRepository = purchaseRepository;
       this.kafkaTemplate = kafkaTemplate;
   }

   public void purchaseProduct(Product product, int quantity, String email) {
       Product dbProduct = productRepository.findById(product.getId()).get();

       Purchase purchase = new Purchase();
       purchase.setProduct(dbProduct);
       purchase.setQuantity(quantity);
       purchase.setPurchasedAt(LocalDateTime.now());
       purchaseRepository.save(purchase);
```

```
        kafkaTemplate.send("product_purchased", email, purchase);
    }
}
```

In the above code, a purchase record is created in the database when a product is purchased, and the product and email are sent as the message to the Kafka topic using `KafkaTemplate`.

We'll now write a consumer `InventoryConsumer` that will listen to the topic `product_purchased`, using `KafkaListener`. It will consume the message from the topic and first decrement from the inventory and then send an email to the user regarding the purchase.

```
@Component
public class InventoryConsumer {

    private final ProductRepository productRepository;
    private final EmailService emailService;

    @Autowired
    public InventoryConsumer(ProductRepository productRepository, EmailService
emailService) {
        this.productRepository = productRepository;
        this.emailService = emailService;
    }

    @KafkaListener(topics = "product_purchased")
    public void decrementInventoryAndSendEmail(Purchase purchase,
                    @Header(KafkaHeaders.RECEIVED_MESSAGE_KEY) String email) {
        Product dbProduct =
             productRepository.findById(purchase.getProduct().getId()).get();
        dbProduct.setQuantity(dbProduct.getQuantity() -
                              purchase.getQuantity());
        productRepository.save(dbProduct);

        emailService.sendProductPurchaseEmail(dbProduct, email);
    }
```

In the above example, the email address of the user is passed as a `header` to the Kafka consumer, and the email service uses that header to send an email to the user about the successful product purchase.

SMTP Configuration in `application.yml`:

```
spring:
 mail:
```

```
    host: smtp.gmail.com
    port: 587
    username: your.email@gmail.com
    password: your-email-password
    properties:
      mail:
        smtp:
          auth: true
          starttls:
            enable: true
```

Email Service Class:

```
@Service
public class EmailService {

    private final JavaMailSender javaMailSender;

    @Value("${spring.mail.username}")
    private String emailFrom;

    @Autowired
    public EmailService(JavaMailSender javaMailSender) {
        this.javaMailSender = javaMailSender;
    }

    public void sendProductPurchaseEmail(Product product, String toEmail) {
        SimpleMailMessage mailMessage = new SimpleMailMessage();
        mailMessage.setFrom(emailFrom);
        mailMessage.setTo(toEmail);
        mailMessage.setSubject("Product Purchase Successful");
        mailMessage.setText("You have successfully purchased the product: " +
product.getName());

        javaMailSender.send(mailMessage);
    }
}
```

Now we can run the application and test the feature by invoking the REST API using Postman or any other client.

References:

1. https://www.geeksforgeeks.org/how-to-install-and-run-apache-kafka-on-windows/
2. https://kafka.apache.org/documentation/