# Kafka Consumer Efficiency

How to improve efficient of kafka consumer with Spring Boot

To improve the efficiency of a Kafka consumer in a Spring Boot application, you can optimize the consumer configuration, fine-tune resource usage, and apply various best practices. Here are some strategies:

### 1. Concurrency & Partitioning

- **Increase Consumer Concurrency**: By increasing the number of consumer threads (`concurrency`), you can process messages in parallel. However, ensure the number of consumers matches or is less than the number of partitions in the topic.

  java

  ```java
  @Bean
  public ConcurrentKafkaListenerContainerFactory<String, String>
  kafkaListenerContainerFactory() {
      ConcurrentKafkaListenerContainerFactory<String, String> factory =
          new ConcurrentKafkaListenerContainerFactory<>();
      factory.setConsumerFactory(consumerFactory());
      factory.setConcurrency(3); // Number of consumers per listener
      return factory;
  }
  ```

- **Partitioning**: Ensure that your Kafka topic is partitioned properly so that multiple consumers can be assigned partitions and scale horizontally.

### 2. Batch Processing

- **Enable Batch Listener**: Instead of processing messages one-by-one, configure the consumer to process batches of records. This reduces overhead and increases throughput.

  java

  ```java
  @KafkaListener(topics = "topicName", containerFactory = "batchFactory")
  public void listen(List<String> messages) {
      messages.forEach(message -> process(message));
  }
  ```

- **Batch Size**: Configure the batch size in `ConsumerConfig` to control how many records to fetch at once.

```properties
spring.kafka.consumer.fetch-min-size=500000
spring.kafka.consumer.max-poll-records=1000
```

### 3. Optimal Polling

- **Adjust `max.poll.interval.ms`**: Set the maximum time between poll requests to prevent consumer rebalances due to slower processing times.

  ```properties
  spring.kafka.consumer.max-poll-interval-ms=300000
  ```

- **Increase `max.poll.records`**: Fetch more records in a single poll to process more data at once.

  ```properties
  spring.kafka.consumer.max-poll-records=500
  ```

### 4. Tuning Offsets

- **Enable Async Offset Commits**: Asynchronous offset commits allow for better throughput, especially for high-throughput consumers.

  ```java
  factory.getContainerProperties().setAckMode(AckMode.MANUAL_IMMEDIATE);
  ```

- **Manual Offset Management**: For better control, commit offsets manually after processing messages.

  ```java
  @KafkaListener(topics = "topicName", containerFactory = "kafkaListenerContainerFactory")
  public void listen(ConsumerRecords<String, String> records, Acknowledgment acknowledgment) {
      records.forEach(record -> process(record));
      acknowledgment.acknowledge(); // Manual acknowledgment
  }
  ```

### 5. Connection Pooling

Use connection pooling for Kafka producers if your consumers also produce messages. This avoids the overhead of creating new Kafka producers each time.

### 6. Resource Tuning

- **Increase Heap Size**: Adjust JVM memory settings to allocate sufficient heap space for the Kafka consumer, especially when consuming large volumes of messages.
- **Use Efficient Deserialization**: If possible, optimize your deserialization logic, or use fast serializers/deserializers like Avro or Protobuf.

### 7. Compression

- **Enable Compression**: If message size is large, use Kafka's built-in compression (e.g., ``` `gzip` ```, ``` `snappy` ```, or ``` `lz4` ```) to reduce network usage and improve consumer efficiency.

```properties
spring.kafka.producer.compression-type=gzip
```

### 8. Enable Parallel Processing

If your message processing is CPU-bound or can be done in parallel, offload the actual message handling to another thread pool.

```java
@KafkaListener(topics = "topicName")
public void listen(String message) {
    taskExecutor.submit(() -> processMessage(message));
}
```

### 9. Monitoring & Metrics

- **Enable Monitoring**: Use Kafka metrics to monitor consumer performance and tune the application accordingly (e.g., consumer lag, throughput, errors).
- **Adjust Logging Levels**: Reduce unnecessary logging that might introduce I/O bottlenecks during high-volume processing.

### 10. Leverage Backpressure Handling

- Implement mechanisms like buffering or rate-limiting to prevent your consumers from being overwhelmed during spikes in message traffic.

### 11. Kafka Broker Optimization

Ensure that the Kafka brokers are optimally configured for partition replication, compression, and throughput, as this indirectly impacts the consumer's efficiency.
By combining these techniques, you can significantly improve the efficiency of Kafka consumers in your Spring Boot application.

Can you include error handling also.. And Can you avoid annotation

**Memory updated**

Here's how you can improve the efficiency of a Kafka consumer in a Spring Boot application without using annotations and with robust error handling:

### 1. Creating Kafka Consumer without Annotations

Instead of using `@KafkaListener`, you can configure the consumer manually using `KafkaConsumer<K, V>`.

java

```java
@Bean
public KafkaConsumer<String, String> kafkaConsumer() {
    Properties props = new Properties();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    props.put(ConsumerConfig.GROUP_ID_CONFIG, "group_id");
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
    props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false); // Disable auto-commit for
manual control
    props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, "1000");

    KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
    consumer.subscribe(Collections.singletonList("topicName"));
    return consumer;
}
```

### 2. Poll & Process Records

You can now manually poll records from Kafka and process them in batches.

java

```java
public void consumeMessages(KafkaConsumer<String, String> consumer) {
    try {
        while (true) {
            ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(100));
            if (!records.isEmpty()) {
                processRecords(records);
                consumer.commitSync(); // Manually commit after processing
            }
        }
    } catch (Exception e) {
        handleError(e);
    } finally {
        consumer.close();
    }
}
```

### 3. Efficient Batch Processing

Process records in batches

Use Spring Boot, without annotation

To set up a Kafka consumer in a Spring Boot application without using annotations, you can configure everything programmatically. Here’s a step-by-step approach, including error handling:

### 1. Configure Kafka Consumer Properties

Define your Kafka consumer properties in a configuration class.

java

```
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.util.Properties;

@Configuration
public class KafkaConsumerConfig {

    @Bean
    public Properties kafkaConsumerProperties() {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "group_id");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false"); // Disable auto-commit
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, "1000");
        props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "15000");
        return props;
    }
}
```

### 2. Create the Kafka Consumer

Create a Kafka consumer bean and manage it manually.

java

```
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.common.TopicPartition;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import java.time.Duration;
import java.util.Collections;
import java.util.Properties;

@Configuration
public class KafkaConsumerManager {

    @Autowired
    private Properties kafkaConsumerProperties;

    private Consumer<String, String> consumer;

    @PostConstruct
    public void start() {
        consumer = new KafkaConsumer<>(kafkaConsumerProperties);
        consumer.subscribe(Collections.singletonList("topicName"));

        new Thread(() -> {
            while (true) {
                try {
                    ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(1000));
                    records.forEach(record -> {
                        try {
                            processRecord(record);
                        } catch (Exception e) {
                            handleError(record, e);
                        }
                    });
                    consumer.commitSync();
                } catch (Exception e) {
                    // Log and handle polling exceptions
                    e.printStackTrace();
                }
            }
        }).start();
    }

    @PreDestroy
    public void stop() {
        if (consumer != null) {
            consumer.close();
        }
    }

    private void processRecord(ConsumerRecord<String, String> record) {
        // Process your record here
        System.out.printf("Consumed record with key %s and value %s%n", record.key(),
record.value());
    }

    private void handleError(ConsumerRecord<String, String> record, Exception e) {
        // Handle errors during processing
        System.err.printf("Error processing record with key %s and value %s: %s%n",
record.key(), record.value(), e.getMessage());
        // Implement error handling logic (e.g., retry logic, logging, etc.)
    }
}
```

### 3. Error Handling Strategies

- **Logging**: Log errors for further analysis and debugging.
- **Retry Logic**: Implement retry logic in the `handleError` method if needed.
- **Dead Letter Queue**: Send problematic records to a dead letter queue for manual inspection.

### 4. Consumer Lifecycle Management

- **Graceful Shutdown**: Ensure that the consumer is closed properly on application shutdown to avoid resource leaks.

### 5. Testing and Monitoring

- **Unit Testing**: Test your consumer logic and error handling in isolation.
- **Monitoring**: Use tools to monitor Kafka consumer metrics (e.g., lag, throughput) and set up alerts for any issues.

By following these steps, you'll have a fully functional Kafka consumer in a Spring Boot application, without relying on annotations, and with comprehensive error handling.

Use ConcurrentKafkaListenerContainerFactory
Create Bean Programatically
No use of annotation because I have to created resource based on dynamic configuration
Error Handling
And Improve efficiency
Best Practice

**Memory updated**

To create a Kafka consumer programmatically using `ConcurrentKafkaListenerContainerFactory` while incorporating error handling and improving efficiency, follow these steps:

### 1. Configure Kafka Consumer Properties

Define Kafka consumer properties programmatically.

java

```java
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.util.HashMap;
import java.util.Map;

@Configuration
public class KafkaConfig {

    @Bean
    public Map<String, Object> kafkaConsumerProperties() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "dynamic-group-id");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false"); // Disable auto-
commit for manual control
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, "1000");
        props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "15000");
        return props;
    }
}
```

## 2. Create `ConcurrentKafkaListenerContainerFactory` Programmatically

Set up the `ConcurrentKafkaListenerContainerFactory` to use your properties.

```java
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.core.DefaultKafkaConsumerFactory;
import org.springframework.kafka.listener.ConcurrentMessageListenerContainer;
import org.springframework.kafka.listener.config.ContainerProperties;
import org.springframework.kafka.listener.SeekToCurrentErrorHandler;
import org.springframework.kafka.support.serializer.ErrorHandlingDeserializer;
import org.springframework.kafka.support.serializer.JsonDeserializer;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.util.Map;

@Configuration
public class KafkaConsumerFactoryConfig {

    @Autowired
    private Map<String, Object> kafkaConsumerProperties;

    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, String>
kafkaListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, String> factory =
            new ConcurrentKafkaListenerContainerFactory<>();

        factory.setConsumerFactory(consumerFactory());
        factory.setConcurrency(3); // Number of concurrent consumers
        factory.getContainerProperties().setPollTimeout(3000);
        factory.setBatchListener(true); // Process messages in batches

        // Error handling
        factory.setErrorHandler(new SeekToCurrentErrorHandler((record, exception) -> {
            // Handle the error here
            System.err.printf("Error processing record with key %s and value %s: %s%n",
record.key(), record.value(), exception.getMessage());
        }, 3)); // Retry up to 3 times before sending to DLQ

        return factory;
    }

    @Bean
    public ConsumerFactory<String, String> consumerFactory() {
        return new DefaultKafkaConsumerFactory<>(kafkaConsumerProperties);
    }
}
```

## 3. Consume Messages Programmatically

Create a class to consume messages using the `ConcurrentKafkaListenerContainerFactory`.

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.listener.ConcurrentMessageListenerContainer;
import org.springframework.kafka.listener.MessageListener;
import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

@Component
public class KafkaConsumer {

    @Autowired
    private ConcurrentKafkaListenerContainerFactory<String, String>
kafkaListenerContainerFactory;

    private ConcurrentMessageListenerContainer<String, String> container;

    @PostConstruct
    public void start() {
        container = kafkaListenerContainerFactory.createContainer("topicName");
        container.setMessageListener(new MessageListener<String, String>() {
            @Override
            public void onMessage(ConsumerRecord<String, String> record) {
                try {
                    processRecord(record);
                } catch (Exception e) {
                    handleError(record, e);
                }
            }
        });
        container.start();
    }

    @PreDestroy
    public void stop() {
        if (container != null) {
            container.stop();
        }
    }

    private void processRecord(ConsumerRecord<String, String> record) {
        // Process your record here
        System.out.printf("Consumed record with key %s and value %s%n", record.key(),
record.value());
    }

    private void handleError(ConsumerRecord<String, String> record, Exception e) {
        // Handle errors during processing
        System.err.printf("Error processing record with key %s and value %s: %s%n",
record.key(), record.value(), e.getMessage());
        // Implement error handling logic (e.g., retry logic, logging, etc.)
    }
}
```

## 4. Efficiency Improvements

- **Concurrency**: Increase concurrency in `ConcurrentKafkaListenerContainerFactory` to process messages in parallel.
- **Batch Processing**: Enable batch processing for higher throughput.
- **Manual Offset Management**: Manually commit offsets after successful processing to avoid message loss.
- **Error Handling**: Use `SeekToCurrentErrorHandler` to handle errors and retries effectively.

## 5. Dynamic Configuration

To support dynamic configuration, update `kafkaConsumerProperties` as needed and recreate the `ConcurrentKafkaListenerContainerFactory` bean if necessary.
By following these steps, you can efficiently set up a Kafka consumer programmatically in Spring Boot, handle errors gracefully, and dynamically manage configurations.

Thanks.. Could you please include below:
Enable Parallel Processing
Compression
 Async Offset Commits
Optimal Polling

Certainly! Let's incorporate parallel processing, compression, asynchronous offset commits, and optimal polling into your Kafka consumer setup.

## 1. Parallel Processing

For parallel processing, you can use a thread pool to handle messages asynchronously. In the `MessageListener`, submit tasks to an executor service.

java

```java
import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

@Component
public class KafkaConsumer {

    @Autowired
    private ConcurrentKafkaListenerContainerFactory<String, String>
kafkaListenerContainerFactory;

    private ConcurrentMessageListenerContainer<String, String> container;
    private ExecutorService executorService;

    @PostConstruct
    public void start() {
        executorService = Executors.newFixedThreadPool(10); // Adjust thread pool size as
needed

        container = kafkaListenerContainerFactory.createContainer("topicName");
        container.setMessageListener(new MessageListener<String, String>() {
            @Override
            public void onMessage(ConsumerRecord<String, String> record) {
                executorService.submit(() -> {
                    try {
                        processRecord(record);
                    } catch (Exception e) {
                        handleError(record, e);
                    }
                });
            }
        });
        container.start();
    }

    @PreDestroy
    public void stop() {
        if (container != null) {
            container.stop();
        }
        if (executorService != null) {
            executorService.shutdown();
        }
    }

    private void processRecord(ConsumerRecord<String, String> record) {
        // Process your record here
        System.out.printf("Consumed record with key %s and value %s%n", record.key(),
record.value());
    }

    private void handleError(ConsumerRecord<String, String> record, Exception e) {
        // Handle errors during processing
        System.err.printf("Error processing record with key %s and value %s: %s%n",
record.key(), record.value(), e.getMessage());
        // Implement error handling logic (e.g., retry logic, logging, etc.)
    }
}
```

## 2. Compression

To enable compression for Kafka producers, you can configure it in the producer properties. This helps reduce the amount of data sent over the network and can be set up in your producer configuration.

```java
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.springframework.kafka.core.DefaultKafkaProducerFactory;
import org.springframework.kafka.core.ProducerFactory;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.util.HashMap;
import java.util.Map;

@Configuration
public class KafkaProducerConfig {

    @Bean
    public Map<String, Object> producerProperties() {
        Map<String, Object> props = new HashMap<>();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
        props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "gzip"); // Enable compression
        return props;
    }

    @Bean
    public ProducerFactory<String, String> producerFactory() {
        return new DefaultKafkaProducerFactory<>(producerProperties());
    }

    @Bean
    public KafkaTemplate<String, String> kafkaTemplate() {
        return new KafkaTemplate<>(producerFactory());
    }
}
```

## 3. Asynchronous Offset Commits

Enable asynchronous offset commits by configuring the `ConcurrentKafkaListenerContainerFactory` to use manual acknowledgment and commit offsets asynchronously.

```java
import org.springframework.kafka.listener.AcknowledgingMessageListener;
import org.springframework.kafka.listener.MessageListener;
import org.springframework.kafka.listener.config.ContainerProperties;
import org.springframework.kafka.listener.SeekToCurrentErrorHandler;

@Bean
public ConcurrentKafkaListenerContainerFactory<String, String>
kafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<String, String> factory =
        new ConcurrentKafkaListenerContainerFactory<>();

    factory.setConsumerFactory(consumerFactory());
    factory.setConcurrency(3); // Number of concurrent consumers
    factory.setBatchListener(true); // Process messages in batches
    factory.setErrorHandler(new SeekToCurrentErrorHandler((record, exception) -> {
        // Handle the error here
        System.err.printf("Error processing record with key %s and value %s: %s%n",
record.key(), record.value(), exception.getMessage());
    }, 3)); // Retry up to 3 times before sending to DLQ


factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL_IMMEDIATE);
// Enable manual, immediate acknowledgment
    return factory;
}
```

## 4. Optimal Polling

Configure optimal polling settings by adjusting properties like `max.poll.records` and `max.poll.interval.ms` in the Kafka consumer configuration.

```java
@Bean
public Map<String, Object> kafkaConsumerProperties() {
    Map<String, Object> props = new HashMap<>();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    props.put(ConsumerConfig.GROUP_ID_CONFIG, "dynamic-group-id");
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
    props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false"); // Disable auto-commit
for manual control
    props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, "1000"); // Fetch more records in
one poll
    props.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, "15000");
    props.put(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG, "300000"); // Increase max poll
interval
```

```
    return props;
}
```

## Summary

1. **Parallel Processing**: Use an executor service to process records in parallel.
2. **Compression**: Configure Kafka producers to use compression.
3. **Asynchronous Offset Commits**: Enable manual, immediate acknowledgment for asynchronous offset commits.
4. **Optimal Polling**: Adjust properties like `` `max.poll.records` `` and `` `max.poll.interval.ms` `` for optimal performance.

By following these practices, you can ensure your Kafka consumers are efficient, resilient, and capable of handling high throughput effectively.

Could you please write generic class for commit strategies

Certainly! Here's a generic class for managing commit strategies in Kafka consumers, allowing you to easily switch between different commit strategies (e.g., manual, auto, or async commits).

### 1. Commit Strategy Interface

Define a common interface for commit strategies.

```java
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerRecord;

public interface CommitStrategy<K, V> {
    void commitOffsets(Consumer<K, V> consumer, Iterable<ConsumerRecord<K, V>> records);
}
```

### 2. Auto Commit Strategy

Implement auto commit strategy.

```java
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerRecord;

public class AutoCommitStrategy<K, V> implements CommitStrategy<K, V> {
    @Override
    public void commitOffsets(Consumer<K, V> consumer, Iterable<ConsumerRecord<K, V>> records) {
        // Auto commit is handled by Kafka itself based on the `enable.auto.commit` configuration
    }
}
```

## 3. Manual Commit Strategy

Implement manual commit strategy.

java

```
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerRecord;

public class ManualCommitStrategy<K, V> implements CommitStrategy<K, V> {
    @Override
    public void commitOffsets(Consumer<K, V> consumer, Iterable<ConsumerRecord<K, V>> records) {
        // Commit offsets manually
        consumer.commitSync();
    }
}
```

## 4. Async Commit Strategy

Implement asynchronous commit strategy.

java

```
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.OffsetCommitCallback;

public class AsyncCommitStrategy<K, V> implements CommitStrategy<K, V> {
    @Override
    public void commitOffsets(Consumer<K, V> consumer, Iterable<ConsumerRecord<K, V>> records) {
        // Commit offsets asynchronously
        consumer.commitAsync(new OffsetCommitCallback() {
            @Override
            public void onComplete(Map<TopicPartition, OffsetAndMetadata> offsets, Exception exception) {
                if (exception != null) {
                    System.err.printf("Failed to commit offsets: %s%n", exception.getMessage());
                } else {
                    System.out.println("Offsets committed successfully");
                }
            }
        });
    }
}
```

## 5. Kafka Consumer Configuration

Create a class to integrate the commit strategy with Kafka consumers.

java

```java
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.util.Collections;
import java.util.Map;
import java.util.Properties;

@Configuration
public class KafkaConsumerConfig {

    @Autowired
    private Map<String, Object> kafkaConsumerProperties;

    @Bean
    public Consumer<String, String> kafkaConsumer() {
        Properties props = new Properties();
        props.putAll(kafkaConsumerProperties);
        return new KafkaConsumer<>(props);
    }

    @Bean
    public CommitStrategy<String, String> commitStrategy() {
        // Choose the desired commit strategy
        return new ManualCommitStrategy<>();
    }
}
```

### 6. Kafka Consumer Implementation

Implement the consumer logic using the chosen commit strategy.

java

```java
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.common.TopicPartition;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.listener.ConcurrentMessageListenerContainer;
import org.springframework.kafka.listener.MessageListener;
import org.springframework.kafka.listener.config.ContainerProperties;
import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import java.util.Collections;
import java.util.Map;

@Component
public class KafkaConsumerManager {

    @Autowired
    private Consumer<String, String> kafkaConsumer;

    @Autowired
    private CommitStrategy<String, String> commitStrategy;

    @PostConstruct
    public void start() {
        kafkaConsumer.subscribe(Collections.singletonList("topicName"));
        new Thread(() -> {
            while (true) {
                try {
                    var records = kafkaConsumer.poll(Duration.ofMillis(1000));
                    if (records.count() > 0) {
                        // Process records
                        for (ConsumerRecord<String, String> record : records) {
                            processRecord(record);
                        }
                        // Commit offsets using the chosen strategy
                        commitStrategy.commitOffsets(kafkaConsumer, records);
                    }
                } catch (Exception e) {
                    // Handle exceptions during polling
                    e.printStackTrace();
                }
            }
        }).start();
    }

    @PreDestroy
    public void stop() {
        if (kafkaConsumer != null) {
            kafkaConsumer.close();
        }
    }

    private void processRecord(ConsumerRecord<String, String> record) {
        // Process the record
        System.out.printf("Consumed record with key %s and value %s%n", record.key(),
record.value());
    }
}
```

## Summary

1. **Commit Strategy Interface**: Define a common interface for commit strategies.
2. **Auto Commit Strategy**: Handles auto-commit offsets.
3. **Manual Commit Strategy**: Commits offsets manually.
4. **Async Commit Strategy**: Commits offsets asynchronously.
5. **Kafka Consumer Configuration**: Set up the Kafka consumer and inject the commit strategy.
6. **Kafka Consumer Implementation**: Use the commit strategy in your consumer logic.

This setup allows you to easily switch between different commit strategies and maintain a clean and flexible Kafka consumer implementation.

ChatGPT can make mistakes. Check important info.