Kafka Configurations Overview

You said:

Spring Kafka

Is it possible to write only configuration based code having multiple cluster, multiple topics and multiple listener..

Zero Code

ChatGPT said:

Yes, it is possible to write configuration-based code with Spring Kafka to manage multiple clusters, topics, and listeners without writing much business logic code. This approach typically relies on using Spring's powerful configuration capabilities, such as YAML or properties files, along with Spring Boot's auto-configuration and Spring Kafka's declarative configuration support.
Here's how you can approach this:

1. Multiple Kafka Clusters:

- You can define multiple Kafka clusters in your application properties or YAML file. Spring Boot allows you to define different Kafka clusters using `` `KafkaProperties` `` or by defining multiple `` `KafkaTemplate` `` and `` `ConsumerFactory` `` beans.

2. Multiple Topics and Listeners:

- You can configure multiple topics and listeners by using the `` `@KafkaListener` `` annotation with `` `topics` `` parameter, or by configuring listener containers programmatically.
- You can also set up listeners for different topics across clusters using configurations and bean definitions.

3. Zero Business Logic Code:

- To achieve "zero code" in terms of business logic, you'd mostly rely on annotations and configuration files. However, while you won't write much Java code, you'll still need to define your setup in YAML/properties and possibly some minimal Java configuration classes to tie everything together.

Example Configuration in YAML:

yaml

```yaml
spring:
  kafka:
    consumer:
      group-id: group1
      auto-offset-reset: earliest
    producer:
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      value-serializer: org.apache.kafka.common.serialization.StringSerializer

    clusters:
      cluster1:
        bootstrap-servers: broker1:9092,broker2:9092
      cluster2:
        bootstrap-servers: broker3:9092,broker4:9092

    listeners:
      listener1:
        id: listener1
        topics: topic1
        containerFactory: kafkaListenerContainerFactoryCluster1
      listener2:
        id: listener2
        topics: topic2
        containerFactory: kafkaListenerContainerFactoryCluster2
```

Example Java Configuration:

java

```java
@Configuration
public class KafkaConfig {

    @Bean
    public ConsumerFactory<String, String> consumerFactoryCluster1() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092,broker2:9092");
        // Add other necessary properties
        return new DefaultKafkaConsumerFactory<>(props);
    }

    @Bean
    public KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<String,
String>> kafkaListenerContainerFactoryCluster1() {
        ConcurrentKafkaListenerContainerFactory<String, String> factory =
                new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactoryCluster1());
        return factory;
    }

    @Bean
    public ConsumerFactory<String, String> consumerFactoryCluster2() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker3:9092,broker4:9092");
        // Add other necessary properties
        return new DefaultKafkaConsumerFactory<>(props);
    }

    @Bean
    public KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<String,
String>> kafkaListenerContainerFactoryCluster2() {
        ConcurrentKafkaListenerContainerFactory<String, String> factory =
                new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactoryCluster2());
        return factory;
    }
}
```

Setting Up Listeners:

```java
@Service
public class KafkaListeners {

    @KafkaListener(id = "listener1", topics = "topic1", containerFactory =
"kafkaListenerContainerFactoryCluster1")
    public void listenCluster1(String message) {
        // Process message
    }

    @KafkaListener(id = "listener2", topics = "topic2", containerFactory =
"kafkaListenerContainerFactoryCluster2")
    public void listenCluster2(String message) {
        // Process message
    }
}
```

Advantages:

- **Separation of Concerns**: Business logic is kept separate from configuration.
- **Declarative Setup**: Makes use of Spring's declarative configuration abilities, reducing boilerplate code.
- **Reusability**: The setup is reusable and can be easily scaled.

Limitations:

- **Minimal Java Code Required**: While you aim for zero code, some minimal Java code is typically required for configuring beans and setting up listeners.
- **Complexity**: Managing multiple clusters and topics can get complex, and careful attention is needed to avoid misconfigurations.

Would this approach work for your use case, or do you need further customization?

You said:

Can we generalize kafkalistener based on some configuraion

ChatGPT said:

Yes, you can generalize `@KafkaListener` based on configuration, allowing dynamic creation of listeners for different topics and clusters based on properties or YAML configuration. This approach leverages Spring's flexible configuration capabilities to define listeners dynamically.

Steps to Generalize `@KafkaListener`:

1. **Define Configuration Properties**:
    - You can create a custom configuration class to read cluster and topic configurations from properties or YAML files.
2. **Create Dynamic Kafka Listeners**:
    - Use Spring's `@PostConstruct` or `ApplicationListener<ApplicationReadyEvent>` to programmatically register listeners after the application context is initialized.
3. **Use a `MessageListener` with a Generic Kafka Listener Container**:
    - Instead of hardcoding `@KafkaListener`, create a generic method to configure and start listener containers dynamically.

Example Configuration:

YAML Configuration:

yaml

```yaml
kafka:
  clusters:
    - name: cluster1
      bootstrap-servers: broker1:9092
      topics:
        - name: topic1
          group-id: group1
        - name: topic2
          group-id: group2
    - name: cluster2
      bootstrap-servers: broker3:9092
      topics:
        - name: topic3
          group-id: group3
```

Configuration Class:

java

```java
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

import java.util.List;

@Component
@ConfigurationProperties(prefix = "kafka")
public class KafkaClusterConfig {

    private List<Cluster> clusters;

    public static class Cluster {
        private String name;
        private String bootstrapServers;
        private List<Topic> topics;

        // Getters and setters...

        public static class Topic {
            private String name;
            private String groupId;

            // Getters and setters...
        }
    }

    // Getters and setters...
}
```

Dynamic Listener Registration:

```java
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.event.ContextRefreshedEvent;
import org.springframework.context.event.EventListener;
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.MessageListener;
import org.springframework.kafka.listener.config.ContainerProperties;
import org.springframework.stereotype.Component;

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

@Component
public class KafkaListenerInitializer {

    @Autowired
    private KafkaClusterConfig kafkaClusterConfig;

    @EventListener(ContextRefreshedEvent.class)
    public void onApplicationEvent() {
        for (KafkaClusterConfig.Cluster cluster : kafkaClusterConfig.getClusters()) {
            for (KafkaClusterConfig.Cluster.Topic topic : cluster.getTopics()) {
                createListenerForTopic(cluster.getBootstrapServers(), topic.getName(),
topic.getGroupId());
            }
        }
    }

    private void createListenerForTopic(String bootstrapServers, String topic, String
groupId) {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        props.put(ConsumerConfig.GROUP_ID_CONFIG, groupId);
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);

        ConsumerFactory<String, String> consumerFactory = new
DefaultKafkaConsumerFactory<>(props);
        ConcurrentKafkaListenerContainerFactory<String, String> factory = new
ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory);

        MessageListener<String, String> messageListener = record -> {
            // Process the record
            System.out.println("Received message: " + record.value() + " from topic: " +
record.topic());
        };

        ContainerProperties containerProps = new ContainerProperties(topic);
        containerProps.setMessageListener(messageListener);

        ConcurrentMessageListenerContainer<String, String> container =
factory.createContainer(containerProps);
        container.start();
        }
    }
```

Explanation:

- `KafkaClusterConfig`: This is a custom configuration class that binds to the properties in the YAML file.
- `KafkaListenerInitializer`: This component listens for the Spring context to be refreshed and then registers Kafka listeners dynamically based on the configuration.
- `createListenerForTopic` **Method**: It sets up a `MessageListener` and a `ConcurrentMessageListenerContainer` for each topic dynamically. This method takes the `bootstrapServers`, `topic`, and `groupId` from the configuration and creates a Kafka consumer for that topic.

Advantages:

- **Dynamic and Scalable**: You can add or remove clusters, topics, and listeners without changing the Java code. Just update the YAML or properties file.
- **Centralized Configuration**: All Kafka-related configurations are centralized, making it easier to manage.

Considerations:

- **Complexity**: Although this approach reduces hardcoded listeners, it introduces complexity in managing and troubleshooting Kafka consumers.
- **Resource Management**: Ensure that the number of consumers and threads does not overwhelm your application or Kafka brokers.

This generalized approach should provide the flexibility and scalability needed for managing multiple Kafka clusters, topics, and listeners dynamically. Would you like to dive deeper into any specific part of this solution?

You said:

Spring Kafka Default is required??

ChatGPT said:

No, a default `KafkaListener` or `KafkaTemplate` is not strictly required in Spring Kafka. However, having a default can simplify the configuration and usage in cases where you don't need to manage multiple Kafka clusters or topics with different configurations.

When a Default Configuration is Useful:

- **Single Cluster and Simple Setup**: If your application interacts with a single Kafka cluster and you don't have complex requirements like multiple consumers with different configurations, a default setup is very convenient.
- **Reduced Boilerplate**: A default configuration minimizes the need to specify configuration details every time you create a listener or producer. Spring will automatically wire up the necessary beans based on default configurations.

When a Default Configuration is Not Needed:

- **Multiple Clusters or Custom Configurations**: If your application needs to handle multiple Kafka clusters, topics with different configurations, or you need fine-grained control over consumer and producer settings, you might not use the default configuration at all. Instead, you can create custom `` `KafkaListenerContainerFactory` ``, `` `ConsumerFactory` ``, and `` `KafkaTemplate` `` beans.
- **Programmatic Setup**: If you're setting up listeners and producers programmatically (e.g., dynamically based on external configuration), you may bypass default configurations and manage everything explicitly.

Example Without Default:

You can skip default configurations by explicitly creating and managing your Kafka-related beans.

Custom Consumer Factory:

```java
@Bean
public ConsumerFactory<String, String> consumerFactoryCluster1() {
    Map<String, Object> props = new HashMap<>();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
    props.put(ConsumerConfig.GROUP_ID_CONFIG, "group1");
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
    return new DefaultKafkaConsumerFactory<>(props);
}

@Bean
public KafkaListenerContainerFactory<ConcurrentMessageListenerContainer<String, String>>
kafkaListenerContainerFactoryCluster1() {
    ConcurrentKafkaListenerContainerFactory<String, String> factory =
            new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactoryCluster1());
    return factory;
}
```

Using Custom Factory:

```java
@KafkaListener(topics = "topic1", containerFactory =
"kafkaListenerContainerFactoryCluster1")
public void listen(String message) {
    System.out.println("Received message: " + message);
}
```

Avoiding Default Kafka Configuration:

If you want to ensure that no default configuration is applied, you can avoid setting properties like `` `spring.kafka.bootstrap-servers` ``, `` `spring.kafka.consumer.group-id` ``, etc., in your application properties. Instead, you manage all Kafka-related beans explicitly.

Conclusion:

- **Default Configuration**: Useful for simple, single-cluster setups or when you want to minimize configuration overhead.
- **Custom Configuration**: Necessary when dealing with multiple clusters, topics, or when you require fine-grained control over the Kafka setup.

If your use case involves multiple Kafka clusters and dynamic listener creation, you can safely skip the default configuration and manage everything explicitly through custom beans and configuration classes.

You said:

create bean dynamically containerFactory

ChatGPT said:

To dynamically create Kafka listener container factories in Spring, you'll need to use a combination of configuration classes and programmatic bean registration. This approach allows you to configure multiple Kafka listener containers based on dynamic or external configurations, such as properties loaded from a file or a database.
Here’s a step-by-step guide on how to dynamically create `` `KafkaListenerContainerFactory` `` beans:

Step 1: Define Configuration Properties

Create a configuration properties class to bind external configuration settings.

Example YAML Configuration:

```yaml
kafka:
  clusters:
    - name: cluster1
      bootstrap-servers: broker1:9092
      topics:
        - name: topic1
          group-id: group1
        - name: topic2
          group-id: group2
    - name: cluster2
      bootstrap-servers: broker3:9092
      topics:
        - name: topic3
          group-id: group3
```

Configuration Properties Class:

```java
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

import java.util.List;

@Component
@ConfigurationProperties(prefix = "kafka")
public class KafkaClusterConfig {

    private List<Cluster> clusters;

    public List<Cluster> getClusters() {
        return clusters;
    }

    public void setClusters(List<Cluster> clusters) {
        this.clusters = clusters;
    }

    public static class Cluster {
        private String name;
        private String bootstrapServers;
        private List<Topic> topics;

        // Getters and setters...

        public static class Topic {
            private String name;
            private String groupId;

            // Getters and setters...
        }
    }
}
```

Step 2: Create KafkaListenerContainerFactory Dynamically

You will programmatically create `KafkaListenerContainerFactory` beans based on the configuration properties.

Dynamic Kafka Configuration Class:

```java
import org.apache.kafka.clients.consumer.ConsumerConfig;
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 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.MessageListener;
import java.util.HashMap;
import java.util.Map;

@Configuration
public class KafkaDynamicConfig {

    @Autowired
    private KafkaClusterConfig kafkaClusterConfig;

    @Bean
    public void createKafkaListeners() {
        for (KafkaClusterConfig.Cluster cluster : kafkaClusterConfig.getClusters()) {
            for (KafkaClusterConfig.Cluster.Topic topic : cluster.getTopics()) {
                createListenerForTopic(cluster.getBootstrapServers(), topic.getName(),
topic.getGroupId());
            }
        }
    }

    private void createListenerForTopic(String bootstrapServers, String topic, String
groupId) {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        props.put(ConsumerConfig.GROUP_ID_CONFIG, groupId);
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);

        ConsumerFactory<String, String> consumerFactory = new
DefaultKafkaConsumerFactory<>(props);
        ConcurrentKafkaListenerContainerFactory<String, String> factory = new
ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory);

        // Dynamically create and register the listener container
        ConcurrentMessageListenerContainer<String, String> container =
factory.createContainer(topic);
        container.getContainerProperties().setMessageListener((MessageListener<String,
String>) record -> {
            // Process the record
            System.out.println("Received message: " + record.value() + " from topic: " +
record.topic());
        });
        container.start();
    }
}
```

Step 3: Manage Lifecycle

- **Ensure Proper Shutdown**: If you're dynamically creating listener containers, consider managing their lifecycle properly to ensure resources are cleaned up. You might need to handle stopping the containers on application shutdown or in case of errors.

Explanation:

1. **Configuration Properties**: `KafkaClusterConfig` binds to your YAML configuration and holds cluster and topic details.
2. **Dynamic Creation**: In the `KafkaDynamicConfig` class, you read the configuration properties and create `KafkaListenerContainerFactory` and `ConcurrentMessageListenerContainer` beans dynamically.
3. **Message Listener**: The `MessageListener` implementation is used to process messages from Kafka topics.

Advantages:

- **Flexibility**: Allows you to create listeners dynamically based on runtime or external configuration.
- **Scalability**: Easily adapt to changes in the number of topics or clusters.

Considerations:

- **Complexity**: This approach introduces complexity and requires careful management of Kafka listeners and resources.
- **Testing**: Ensure thorough testing to verify that dynamic listeners are correctly created and managed.

This setup gives you the flexibility to handle a dynamic number of topics and clusters without hardcoding configurations in your application code.

ChatGPT can make mistakes. Check important info.