Spring Boot Kafka Integration

Production Ready Spring Boot Kafka Integration
Error Handling
Proper Logging
Proper exception Handling
Retry mechanism if connection interrupts till connect
Retry mechanism if cluster is down
Retry mechanism if there is no activity like weekend, holiday
Dead Letter Topic
Avro, String and JSON Support with Schema Registry
Proper polling to keep alive the session
Add support for concurrency
Multithreading
Offset Manual Commit
Cover all aspects of Resiliency
Health Acutuator Check for each consumer/cluster

In bank, there is no activity on saturday and sunday. So, no message broadcst to kafka cluster. All the nodes disconnected due to rebalancing.. And the connection never established again.. How to overcome?
User can provide any number of consumer from different cluster. I am not aware of number.
How can we customize?
Write complete code in Spring Boot

Certainly! Building a production-ready Spring Boot application that integrates with Apache Kafka involves several critical components to ensure reliability, resiliency, and scalability. Below, I'll guide you through the key aspects of such an application, including code snippets to illustrate how to implement each feature.

---

1. Project Setup

First, set up your Spring Boot project with the necessary dependencies in your `pom.xml`:

```xml
<dependencies>
    <!-- Spring Boot Kafka -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-artemis</artifactId>
    </dependency>

    <!-- Kafka Client -->
    <dependency>
```

```
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-clients</artifactId>
    </dependency>

    <!-- Confluent Schema Registry (for Avro) -->
    <dependency>
        <groupId>io.confluent</groupId>
        <artifactId>kafka-avro-serializer</artifactId>
        <version>${confluent.version}</version>
    </dependency>

    <!-- Actuator for Health Checks -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
</dependencies>
```

---

2. Configuration

Kafka Configuration

Create a configuration class to set up Kafka producer and consumer factories:

```java
@Configuration
@EnableKafka
public class KafkaConfig {

    @Value("${kafka.bootstrap-servers}")
    private String bootstrapServers;

    // Producer Configuration
    @Bean
    public ProducerFactory<String, Object> producerFactory() {
        Map<String, Object> configProps = new HashMap<>();
        configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        // Add serializers based on data type (String, JSON, Avro)
        configProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        // For JSON
        configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
JsonSerializer.class);
        // For Avro
        // configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
KafkaAvroSerializer.class);
        // configProps.put("schema.registry.url", "http://localhost:8081");
        return new DefaultKafkaProducerFactory<>(configProps);
    }

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

    // Consumer Configuration
    @Bean
    public ConsumerFactory<String, Object> consumerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        // Add deserializers based on data type (String, JSON, Avro)
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        // For JSON
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, JsonDeserializer.class);
        // For Avro
        // props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, 
KafkaAvroDeserializer.class);
        // props.put("schema.registry.url", "http://localhost:8081");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "your-group-id");
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        // Enable manual offset commit
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
        return new DefaultKafkaConsumerFactory<>(props);
    }

    // Listener Container Factory
    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, Object> 
kafkaListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory());
        // Add support for concurrency
        factory.setConcurrency(3);
        // Set manual acknowledgment mode
        factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL);
        return factory;
    }
}
```

---

3. Error Handling and Retry Mechanism

Implement a custom error handler to manage exceptions and retries:

```java
@Component
public class CustomKafkaErrorHandler implements ErrorHandler {

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

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    @Override
    public void handle(Exception thrownException, ConsumerRecord<?, ?> data) {
        logger.error("Error in process with Exception {} and the record is {}", 
thrownException, data);

        // Send the record to a Dead Letter Topic
        kafkaTemplate.send("dead-letter-topic", data.key(), data.value());
    }
}
```

Update the `kafkaListenerContainerFactory` to use the custom error handler and retry mechanism:

```java
@Bean
public ConcurrentKafkaListenerContainerFactory<String, Object> 
kafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<String, Object> factory =
        new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory());
    factory.setConcurrency(3);
    factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL);
```

```
        // Set custom error handler
        factory.setErrorHandler(new SeekToCurrentErrorHandler());

        // Retry mechanism
        factory.setRetryTemplate(retryTemplate());

        return factory;
    }

    private RetryTemplate retryTemplate() {
        RetryTemplate retryTemplate = new RetryTemplate();
        FixedBackOffPolicy backOffPolicy = new FixedBackOffPolicy();
        backOffPolicy.setBackOffPeriod(1000); // Retry every 1 second
        retryTemplate.setBackOffPolicy(backOffPolicy);

        SimpleRetryPolicy retryPolicy = new SimpleRetryPolicy();
        retryPolicy.setMaxAttempts(5); // Retry up to 5 times
        retryTemplate.setRetryPolicy(retryPolicy);

        return retryTemplate;
    }
```

---

4. Kafka Listener with Manual Offset Commit

Create a Kafka listener that processes messages and manually acknowledges them:

```java

    @Service
    public class KafkaConsumerService {

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

        @KafkaListener(topics = "your-topic", containerFactory =
    "kafkaListenerContainerFactory")
        public void consume(ConsumerRecord<String, Object> record, Acknowledgment
    acknowledgment) {
            try {
                // Process the message
                logger.info("Consumed message: {}", record.value());

                // Manually acknowledge
                acknowledgment.acknowledge();

            } catch (Exception e) {
                logger.error("Error processing message: {}", record.value(), e);
                throw e; // Let the error handler manage the exception
            }
        }
    }
```

---

5. Keeping the Session Alive During Inactivity

To prevent the consumer from disconnecting due to inactivity (e.g., on weekends), you can adjust the session timeout and heartbeat interval:

```properties
# application.properties

# Set session timeout to 1 hour (default is 10 seconds)
kafka.consumer.session.timeout.ms=3600000

# Set heartbeat interval to 30 seconds
kafka.consumer.heartbeat.interval.ms=30000

# Adjust max poll interval
kafka.consumer.max.poll.interval.ms=3600000
```

Alternatively, you can implement a scheduled task that polls the Kafka cluster periodically:

```java
@Component
public class KeepAliveScheduler {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    @Scheduled(fixedRate = 60000) // Every 60 seconds
    public void sendKeepAlive() {
        kafkaTemplate.send("your-topic", "keep-alive-key", "KEEP_ALIVE");
    }
}
```

---

6. Handling Rebalancing and Connection Issues

To handle rebalancing and ensure the consumer re-establishes the connection, you can implement the `` `ConsumerRebalanceListener` ``:

```java
factory.getContainerProperties().setConsumerRebalanceListener(new
ConsumerRebalanceListener() {
    @Override
    public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
        logger.info("Partitions revoked: {}", partitions);
    }

    @Override
    public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
        logger.info("Partitions assigned: {}", partitions);
    }
});
```

---

7. Health Checks with Actuator

Implement health checks for each consumer and cluster using Spring Boot Actuator:

```java
@Component
public class KafkaHealthIndicator implements HealthIndicator {

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    @Override
    public Health health() {
        try {
            List<PartitionInfo> partitions = kafkaTemplate.partitionsFor("your-topic");
            if (partitions != null && !partitions.isEmpty()) {
                return Health.up().withDetail("partitions", partitions.size()).build();
            } else {
                return Health.down().withDetail("error", "No partitions found").build();
            }
        } catch (Exception e) {
            return Health.down(e).build();
        }
    }
}
```

---

8. Supporting Multiple Clusters and Consumers

To handle any number of consumers from different clusters, you can define multiple `ConsumerFactory` and `KafkaListenerContainerFactory` beans with different qualifiers:

```java
@Configuration
public class MultiClusterKafkaConfig {

    @Bean
    @Qualifier("cluster1ConsumerFactory")
    public ConsumerFactory<String, Object> cluster1ConsumerFactory() {
        // Configure for cluster 1
    }

    @Bean
    @Qualifier("cluster1KafkaListenerContainerFactory")
    public ConcurrentKafkaListenerContainerFactory<String, Object>
cluster1KafkaListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
            new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(cluster1ConsumerFactory());
        // Additional configurations
        return factory;
    }

    // Repeat for cluster 2, 3, etc.
}
```

---

9. Concurrency and Multithreading

The `ConcurrentKafkaListenerContainerFactory`'s `setConcurrency()` method allows you to run multiple threads for consuming messages:

```java
factory.setConcurrency(5); // Number of threads
```

---

10. Schema Registry and Avro Support

Configure the producer and consumer to use the Schema Registry for Avro serialization/deserialization:

```java
// Producer Configuration
configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, KafkaAvroSerializer.class);
configProps.put("schema.registry.url", "http://localhost:8081");

// Consumer Configuration
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, KafkaAvroDeserializer.class);
props.put("schema.registry.url", "http://localhost:8081");
```

---

11. Dead Letter Topic

Ensure that failed messages are sent to a Dead Letter Topic for further analysis:

```java
@Bean
public DeadLetterPublishingRecoverer deadLetterPublishingRecoverer() {
    return new DeadLetterPublishingRecoverer(kafkaTemplate());
}

@Bean
public SeekToCurrentErrorHandler errorHandler() {
    return new SeekToCurrentErrorHandler(deadLetterPublishingRecoverer(), new
FixedBackOff(1000L, 2));
}
```

---

12. Resiliency

Implement circuit breakers and bulkheads using Resilience4j or similar libraries to enhance resiliency:

```xml
<!-- Add Resilience4j dependency -->
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-spring-boot2</artifactId>
    <version>1.7.0</version>
</dependency>
```

Use annotations like `@Retry`, `@CircuitBreaker`, `@Bulkhead` on your service methods.

---

13. Proper Logging

Utilize a logging framework like SLF4J with Logback or Log4j2. Ensure logs are appropriately leveled (INFO, DEBUG, ERROR) and include contextual information.

```java
private static final Logger logger = LoggerFactory.getLogger(YourClass.class);

logger.info("Informational message");
logger.debug("Debugging details");
logger.error("Error message", exception);
```

---

14. Exception Handling

Implement global exception handling using `@ControllerAdvice` for REST controllers and handle exceptions within your Kafka listeners.

```java
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleAllExceptions(Exception ex) {
        logger.error("Unhandled exception occurred", ex);
        return new ResponseEntity<>("An error occurred",
HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

---

15. Addressing the Weekend Inactivity Issue

To prevent consumers from disconnecting due to inactivity over weekends:

- **Increase the `max.poll.interval.ms`**: This setting controls the maximum delay between invocations of `poll()`. By increasing it, you give your consumer more time before the broker considers it dead.

- **Implement a Keep-Alive Mechanism**: As shown earlier, send periodic keep-alive messages or implement a scheduled task that performs a light poll.

- **Use `auto.offset.reset=earliest`**: This ensures that when the consumer restarts, it picks up messages from the earliest offset if no committed offsets are found.

Complete Code Structure

Due to space constraints, the complete code isn't provided here, but the components above can be assembled into a full Spring Boot application. Here's how you might organize your code:

- **Application Class**: Contains the `main` method to run the Spring Boot application.
- **Configuration Classes**: For Kafka, multiple clusters, and any additional beans.
- **Service Classes**: Contain business logic, Kafka listeners, and producers.
- **Error Handling**: Custom error handlers and exception classes.
- **Logging**: Configuration files for Logback or Log4j2.
- **Health Checks**: Custom health indicators using Actuator.
- **Resources**:
    - `application.properties` or `application.yml` for configuration.
    - Log configuration files.

Customizing for Unknown Number of Consumers and Clusters

If the number of consumers and clusters is dynamic and not known at compile-time, you can:

- **Use a Map of Consumer Factories**: Load configurations from a database or external source and create consumer factories at runtime.

- **Dynamic Bean Registration**: Use Spring's `BeanDefinitionRegistry` to register beans dynamically.

Final Notes

- **Testing**: Ensure you write unit and integration tests for your components.
- **Documentation**: Comment your code and maintain proper documentation.
- **Security**: Secure your application, especially when dealing with sensitive data like in a banking environment.

By following the guidelines and code snippets provided, you can build a robust, production-ready Spring Boot application that integrates seamlessly with Kafka and handles the scenarios you've described.

User can provide any number of consumer from different cluster. I am not aware of number.
How can we customize

Certainly! Handling an **unknown number of consumers from different Kafka clusters** requires a dynamic and flexible approach. Since you don't know the number of clusters or consumers at compile time, you need to design your application to configure and manage them at runtime.

Here's how you can customize your Spring Boot application to handle any number of consumers from different clusters:

---

1. Externalize Cluster and Consumer Configurations

Use external configuration files (`application.yml` or `application.properties`) to define your Kafka clusters and consumers. This allows you to add or remove clusters and consumers without changing the application code.

Example `application.yml`:

```yaml
kafka:
  clusters:
    - name: cluster1
      bootstrap-servers: cluster1.kafka.server:9092
      group-id: group1
      topics:
        - topicA
        - topicB
    - name: cluster2
      bootstrap-servers: cluster2.kafka.server:9092
      group-id: group2
      topics:
        - topicC
        - topicD
    # Additional clusters can be added here
```

---

2. Create Configuration Classes

Cluster Configuration Class:

```java
public class ClusterConfig {

    private String name;
    private String bootstrapServers;
    private String groupId;
    private List<String> topics;

    // Getters and Setters
}
```

Main Configuration Class:

```java
@Configuration
@ConfigurationProperties(prefix = "kafka")
public class KafkaClustersConfig {

    private List<ClusterConfig> clusters;

    // Getters and Setters
}
```

---

3. Dynamically Register Beans for Each Cluster

Use the `ApplicationContext` to dynamically create `ConsumerFactory` and `ConcurrentKafkaListenerContainerFactory` beans for each cluster.

Dynamic Kafka Configuration:

```java
@Configuration
public class DynamicKafkaConfiguration implements ApplicationContextAware {

    private ApplicationContext applicationContext;

    @Autowired
    private KafkaClustersConfig kafkaClustersConfig;

    @PostConstruct
    public void setupKafkaListeners() {
        for (ClusterConfig cluster : kafkaClustersConfig.getClusters()) {
            createKafkaListenerContainerFactory(cluster);
        }
    }

    private void createKafkaListenerContainerFactory(ClusterConfig clusterConfig) {
        // Consumer Properties
        Map<String, Object> consumerProps = new HashMap<>();
        consumerProps.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
clusterConfig.getBootstrapServers());
        consumerProps.put(ConsumerConfig.GROUP_ID_CONFIG, clusterConfig.getGroupId());
        consumerProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
JsonDeserializer.class);
        consumerProps.put(JsonDeserializer.TRUSTED_PACKAGES, "*");
        consumerProps.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        consumerProps.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);

        // Consumer Factory
        ConsumerFactory<String, Object> consumerFactory = new
DefaultKafkaConsumerFactory<>(consumerProps);

        // Kafka Listener Container Factory
        ConcurrentKafkaListenerContainerFactory<String, Object> factory = new
ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory);
        factory.setConcurrency(3); // or any other number based on your requirement
        factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL);

        // Register the factory bean dynamically
        String beanName = clusterConfig.getName() + "KafkaListenerContainerFactory";
        ((GenericApplicationContext) applicationContext).registerBean(beanName,
ConcurrentKafkaListenerContainerFactory.class, () -> factory);
```

```
    }

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws
BeansException {
        this.applicationContext = applicationContext;
    }
}
```

---

4. Create a Dynamic Kafka Listener

Implement a custom annotation and a bean post-processor to register Kafka listeners dynamically for each cluster and topic.

Custom Annotation:

```java
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface DynamicKafkaListener {
    String cluster();
    String[] topics();
}
```

Bean Post-Processor:

```java
@Component
public class DynamicKafkaListenerBeanPostProcessor implements BeanPostProcessor {

    @Autowired
    private KafkaClustersConfig kafkaClustersConfig;

    @Autowired
    private ApplicationContext applicationContext;

    @Autowired
    private KafkaListenerEndpointRegistry kafkaListenerEndpointRegistry;

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws
BeansException {
        Class<?> targetClass = AopUtils.getTargetClass(bean);

        for (Method method : targetClass.getDeclaredMethods()) {
            DynamicKafkaListener annotation = AnnotationUtils.findAnnotation(method,
DynamicKafkaListener.class);
            if (annotation != null) {
                String clusterName = annotation.cluster();
                ClusterConfig clusterConfig = kafkaClustersConfig.getClusters().stream()
                        .filter(c -> c.getName().equals(clusterName))
                        .findFirst()
                        .orElseThrow(() -> new IllegalArgumentException("Cluster not
found: " + clusterName));

                String containerFactoryBeanName = clusterConfig.getName() +
"KafkaListenerContainerFactory";
                ConcurrentKafkaListenerContainerFactory<?, ?> factory =
                        (ConcurrentKafkaListenerContainerFactory<?, ?>)
applicationContext.getBean(containerFactoryBeanName);

                // Register Kafka Listener
                MethodKafkaListenerEndpoint<String, Object> endpoint = new
MethodKafkaListenerEndpoint<>();
                endpoint.setMethod(method);
                endpoint.setBean(bean);
                endpoint.setId(beanName + "_" + method.getName() + "_" + clusterName);
                endpoint.setGroupId(clusterConfig.getGroupId());
                endpoint.setTopics(annotation.topics());
                endpoint.setConcurrency(3); // or based on your needs

endpoint.setBeanFactory(applicationContext.getAutowireCapableBeanFactory());

                kafkaListenerEndpointRegistry.registerListenerContainer(endpoint,
factory);
            }
        }
        return bean;
    }
}
```

---

5. Implement the Listener Methods

Use the `@DynamicKafkaListener` annotation on methods that should consume messages from specific clusters and topics.

Example Consumer Service:

```java
@Service
public class KafkaConsumerService {

    @DynamicKafkaListener(cluster = "cluster1", topics = {"topicA", "topicB"})
    public void listenCluster1(ConsumerRecord<String, Object> record, Acknowledgment
acknowledgment) {
        // Process the message from cluster1
        System.out.println("Cluster1 Message: " + record.value());
        acknowledgment.acknowledge();
    }

    @DynamicKafkaListener(cluster = "cluster2", topics = {"topicC"})
    public void listenCluster2(ConsumerRecord<String, Object> record, Acknowledgment
acknowledgment) {
        // Process the message from cluster2
        System.out.println("Cluster2 Message: " + record.value());
        acknowledgment.acknowledge();
    }

    // Additional listener methods can be added as needed
}
```

---

6. Handle an Unknown Number of Clusters and Consumers

Since you don't know the number of clusters or consumers ahead of time, your application needs to be able to adapt at runtime.

- **Configuration-Driven**: By defining clusters and consumers in external configuration files, you can update the configurations without changing the code.
- **Dynamic Registration**: Using dynamic bean registration, your application can create the necessary beans based on the configurations provided at startup.

---

7. Use a Generic Message Handler (Optional)

If you want to avoid writing multiple listener methods, you can implement a generic handler that processes messages from all clusters and topics.

Generic Listener Method:

```java
@Service
public class GenericKafkaConsumerService {

    @Autowired
    private KafkaClustersConfig kafkaClustersConfig;

    @PostConstruct
    public void setupGenericListeners() {
        for (ClusterConfig clusterConfig : kafkaClustersConfig.getClusters()) {
            registerGenericListener(clusterConfig);
        }
    }

    private void registerGenericListener(ClusterConfig clusterConfig) {
        String containerFactoryBeanName = clusterConfig.getName() +
"KafkaListenerContainerFactory";
        ConcurrentKafkaListenerContainerFactory<?, ?> factory =
                (ConcurrentKafkaListenerContainerFactory<?, ?>)
applicationContext.getBean(containerFactoryBeanName);

        // Define the listener endpoint
        MethodKafkaListenerEndpoint<String, Object> endpoint = new
MethodKafkaListenerEndpoint<>();
        endpoint.setMethod(getGenericListenerMethod());
        endpoint.setBean(this);
        endpoint.setId("genericListener_" + clusterConfig.getName());
        endpoint.setGroupId(clusterConfig.getGroupId());
        endpoint.setTopics(clusterConfig.getTopics().toArray(new String[0]));
        endpoint.setConcurrency(3);
        endpoint.setBeanFactory(applicationContext.getAutowireCapableBeanFactory());

        kafkaListenerEndpointRegistry.registerListenerContainer(endpoint, factory);
    }

    private Method getGenericListenerMethod() {
        try {
            return this.getClass().getMethod("handleMessage", ConsumerRecord.class,
Acknowledgment.class);
        } catch (NoSuchMethodException e) {
            throw new RuntimeException("Failed to get generic listener method", e);
        }
    }

    public void handleMessage(ConsumerRecord<String, Object> record, Acknowledgment
acknowledgment) {
        // Generic message processing
        System.out.println("Received message from " + record.topic() + ": " +
record.value());
        acknowledgment.acknowledge();
        }
    }
```

---

8. Refresh Configurations at Runtime (Advanced)

If configurations might change while the application is running, consider implementing a mechanism to refresh the configurations and re-register listeners.

- **Spring Cloud Config**: Use Spring Cloud Config to manage configurations centrally.
- **Refresh Scope**: Use `@RefreshScope` to reload beans when configurations change.
- **Manual Refresh**: Implement endpoints to trigger a manual refresh.

---

9. Ensure Thread Safety and Concurrency

Since consumers are dynamic and may increase, make sure your application can handle concurrent processing.

- **Concurrency Settings**: Adjust the `setConcurrency()` method based on expected load.
- **Thread Pool Management**: Configure thread pools if necessary to manage resources.

---

10. Error Handling and Retries

Implement error handling mechanisms that work with dynamically registered listeners.

- **Custom Error Handlers**: Create error handlers that can be applied to each `ConcurrentKafkaListenerContainerFactory`.
- **Retry Mechanisms**: Use `RetryTemplate` or Spring Retry to handle transient errors.

Example Error Handler:

```java
    private void createKafkaListenerContainerFactory(ClusterConfig clusterConfig) {
        // Existing code...

        // Error Handler
        factory.setErrorHandler(new SeekToCurrentErrorHandler(
            new DeadLetterPublishingRecoverer(kafkaTemplate()), new FixedBackOff(1000L, 3)));

        // Continue with bean registration...
    }
```

---

11. Health Checks for Each Cluster

Use Spring Boot Actuator to expose health indicators for each Kafka cluster.

Custom Health Indicator:

```java
@Component
public class KafkaClustersHealthIndicator implements HealthIndicator {

    @Autowired
    private KafkaClustersConfig kafkaClustersConfig;

    @Override
    public Health health() {
        for (ClusterConfig clusterConfig : kafkaClustersConfig.getClusters()) {
            try (AdminClient adminClient = AdminClient.create(Map.of(
                    AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG,
clusterConfig.getBootstrapServers()))) {
                DescribeClusterResult clusterInfo = adminClient.describeCluster();
                clusterInfo.clusterId().get();
            } catch (Exception e) {
                return Health.down()
                        .withDetail("Cluster", clusterConfig.getName())
                        .withException(e)
                        .build();
            }
        }
        return Health.up().build();
    }
}
```

---

12. Logging and Monitoring

Ensure that logging includes cluster and consumer details for easier troubleshooting.

- **Log Patterns**: Include cluster name and group ID in log messages.
- **Monitoring Tools**: Use tools like Prometheus and Grafana to monitor consumer metrics.

---

13. Security Considerations

Secure sensitive configurations like bootstrap servers and credentials.

- **Externalize Secrets**: Use environment variables or secret management tools.
- **Encryption**: Encrypt sensitive properties if stored in configuration files.

---

14. Testing

Write tests to ensure that dynamic configurations and listeners work as expected.

- **Integration Tests**: Use `EmbeddedKafka` for testing Kafka interactions.
- **Configuration Tests**: Validate that configurations are loaded correctly.

---

15. Documentation and Maintenance

- **Document Configuration Formats**: Clearly explain how to add or modify clusters and consumers.
- **Maintenance Guidelines**: Provide instructions for operators to manage configurations.

Summary

By externalizing configurations and using dynamic bean registration, your application can handle any number of consumers from different clusters. This approach allows for flexibility and scalability without requiring code changes when clusters or consumers are added or removed.

Key Steps:

1. **Externalize Configurations**: Define clusters and consumers in configuration files.
2. **Dynamic Bean Registration**: Create beans for consumers and listener factories at runtime.
3. **Custom Annotations and Post-Processing**: Use annotations to associate listener methods with clusters.
4. **Generic Handlers (Optional)**: Implement generic message handlers for flexibility.
5. **Error Handling and Health Checks**: Ensure robust error management and monitoring.
6. **Testing and Documentation**: Validate functionality and provide clear documentation.

Feel free to ask if you need further clarification or assistance with implementing this solution!

Using Generic Listener and factory, Could you please bundle with all aspects of resiliency and high availability

Certainly! Let's create a **Spring Boot application** that uses a **Generic Kafka Listener** and **factory** to handle an unknown number of consumers from different clusters, incorporating all aspects of **resiliency** and **high availability**.

We'll cover:

- Dynamic configuration for multiple clusters and topics
- Generic Kafka listener to process messages from all clusters
- Error handling with retries and dead-letter topics
- Support for Avro, String, and JSON serialization with Schema Registry
- Keep-alive mechanisms for inactivity periods
- Health checks using Spring Boot Actuator
- Concurrency and multithreading
- Proper logging and exception handling
- Resiliency patterns (circuit breakers, bulkheads)

Project Setup

1. Dependencies

Add the following dependencies to your `pom.xml`:

xml

```xml
<dependencies>
    <!-- Spring Boot Starter -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>

    <!-- Spring for Apache Kafka -->
    <dependency>
        <groupId>org.springframework.kafka</groupId>
        <artifactId>spring-kafka</artifactId>
    </dependency>

    <!-- Confluent Schema Registry (for Avro support) -->
    <dependency>
        <groupId>io.confluent</groupId>
        <artifactId>kafka-avro-serializer</artifactId>
        <version>7.0.1</version> <!-- Use the appropriate version -->
    </dependency>

    <!-- Spring Boot Actuator -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>

    <!-- Resilience4j for Resiliency Patterns -->
    <dependency>
        <groupId>io.github.resilience4j</groupId>
        <artifactId>resilience4j-spring-boot2</artifactId>
        <version>1.7.1</version>
    </dependency>

    <!-- Logging -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-log4j2</artifactId>
    </dependency>
</dependencies>
```

---

2. External Configuration

application.yml

We'll define Kafka clusters and topics in `application.yml` for dynamic configuration:

yaml

```yaml
kafka:
  clusters:
    - name: cluster1
      bootstrapServers: localhost:9092
      schemaRegistryUrl: http://localhost:8081
      groupId: group1
      topics:
        - topicA
        - topicB
    - name: cluster2
      bootstrapServers: localhost:9093
      schemaRegistryUrl: http://localhost:8082
      groupId: group2
      topics:
        - topicC
        - topicD
```

---

3. Configuration Classes

Cluster Configuration Model

```java
public class ClusterConfig {

    private String name;
    private String bootstrapServers;
    private String schemaRegistryUrl;
    private String groupId;
    private List<String> topics;

    // Getters and Setters
}
```

Main Configuration Class

```java
@Configuration
@ConfigurationProperties(prefix = "kafka")
public class KafkaClustersConfig {

    private List<ClusterConfig> clusters;

    // Getters and Setters
}
```

---

4. Dynamic Kafka Configuration

We will dynamically create `ConsumerFactory` and `KafkaListenerContainerFactory` beans for each cluster.

```java
@Configuration
@EnableKafka
public class DynamicKafkaConfiguration implements ApplicationContextAware {

    private ApplicationContext applicationContext;

    @Autowired
    private KafkaClustersConfig kafkaClustersConfig;

    @Autowired
    private KafkaTemplate<String, Object> kafkaTemplate;

    @PostConstruct
    public void setupKafkaListeners() {
        for (ClusterConfig cluster : kafkaClustersConfig.getClusters()) {
            createKafkaListenerContainerFactory(cluster);
        }
    }

    private void createKafkaListenerContainerFactory(ClusterConfig clusterConfig) {
        // Consumer Properties
        Map<String, Object> consumerProps = new HashMap<>();
        consumerProps.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
clusterConfig.getBootstrapServers());
        consumerProps.put(ConsumerConfig.GROUP_ID_CONFIG, clusterConfig.getGroupId());
        consumerProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        // Support for String, JSON, and Avro
        consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
KafkaAvroDeserializer.class);
        consumerProps.put("schema.registry.url", clusterConfig.getSchemaRegistryUrl());
        consumerProps.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        consumerProps.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
        consumerProps.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 60000); // 1 minute
        consumerProps.put(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG, 300000); // 5
minutes

        // Consumer Factory
        DefaultKafkaConsumerFactory<String, Object> consumerFactory = new
DefaultKafkaConsumerFactory<>(consumerProps);

        // Kafka Listener Container Factory
        ConcurrentKafkaListenerContainerFactory<String, Object> factory = new
ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory);
        factory.setConcurrency(3);

factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL_IMMEDIATE);

        // Error Handling
        factory.setErrorHandler(errorHandler());
        factory.setRetryTemplate(retryTemplate());
        factory.setRecoveryCallback(retryContext -> {
            ConsumerRecord<?, ?> record = (ConsumerRecord<?, ?>)
retryContext.getAttribute("record");
            sendToDeadLetterTopic(record);
            return null;
        });

        // Register the factory bean dynamically
        String beanName = clusterConfig.getName() + "KafkaListenerContainerFactory";
        ((GenericApplicationContext) applicationContext).registerBean(beanName,
ConcurrentKafkaListenerContainerFactory.class, () -> factory);
    }

    private ErrorHandler errorHandler() {
        return new SeekToCurrentErrorHandler(new
DeadLetterPublishingRecoverer(kafkaTemplate), new FixedBackOff(1000L, 2));
    }

    private RetryTemplate retryTemplate() {
        RetryTemplate retryTemplate = new RetryTemplate();

        FixedBackOffPolicy backOffPolicy = new FixedBackOffPolicy();
        backOffPolicy.setBackOffPeriod(2000L); // 2 seconds
        retryTemplate.setBackOffPolicy(backOffPolicy);

        SimpleRetryPolicy retryPolicy = new SimpleRetryPolicy();
        retryPolicy.setMaxAttempts(3);
        retryTemplate.setRetryPolicy(retryPolicy);

        return retryTemplate;
    }

    private void sendToDeadLetterTopic(ConsumerRecord<?, ?> record) {
        kafkaTemplate.send("dead-letter-topic", record.key(), record.value());
    }

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws
BeansException {
        this.applicationContext = applicationContext;
    }
}
```

---

5. Generic Kafka Listener

We'll implement a generic listener method that can process messages from any cluster and topic.

Kafka Consumer Service

```java
@Service
public class GenericKafkaConsumerService {

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

    @Autowired
    private ApplicationContext applicationContext;

    @Autowired
    private KafkaClustersConfig kafkaClustersConfig;

    @Autowired
    private KafkaListenerEndpointRegistry kafkaListenerEndpointRegistry;

    @PostConstruct
    public void setupGenericListeners() {
        for (ClusterConfig clusterConfig : kafkaClustersConfig.getClusters()) {
            registerGenericListener(clusterConfig);
        }
    }

    private void registerGenericListener(ClusterConfig clusterConfig) {
        String containerFactoryBeanName = clusterConfig.getName() +
"KafkaListenerContainerFactory";
        ConcurrentKafkaListenerContainerFactory<String, Object> factory =
                (ConcurrentKafkaListenerContainerFactory<String, Object>)
applicationContext.getBean(containerFactoryBeanName);

        // Define the listener endpoint
        MethodKafkaListenerEndpoint<String, Object> endpoint = new
MethodKafkaListenerEndpoint<>();
        endpoint.setMethod(getGenericListenerMethod());
        endpoint.setBean(this);
        endpoint.setId("genericListener_" + clusterConfig.getName());
        endpoint.setGroupId(clusterConfig.getGroupId());
        endpoint.setTopics(clusterConfig.getTopics());
        endpoint.setBeanFactory(applicationContext.getAutowireCapableBeanFactory());

        kafkaListenerEndpointRegistry.registerListenerContainer(endpoint, factory, true);
    }

    private Method getGenericListenerMethod() {
        try {
            return this.getClass().getMethod("handleMessage", ConsumerRecord.class,
Acknowledgment.class, String.class);
        } catch (NoSuchMethodException e) {
            throw new RuntimeException("Failed to get generic listener method", e);
        }
    }
    public void handleMessage(ConsumerRecord<String, Object> record, Acknowledgment
acknowledgment, @Header(KafkaHeaders.RECEIVED_TOPIC) String topic) {
        try {
            // Process the message
            logger.info("Received message from topic {}: {}", topic, record.value());
            // Implement your business logic here

            // Acknowledge the message
            acknowledgment.acknowledge();
        } catch (Exception e) {
            logger.error("Error processing message: {}", record.value(), e);
            throw e; // Let the error handler handle the exception
        }
    }
}
```

---

6. Producer Configuration

Producer Factory and Kafka Template

java

```java
@Configuration
public class KafkaProducerConfig {

    @Bean
    public ProducerFactory<String, Object> producerFactory(KafkaClustersConfig
kafkaClustersConfig) {
        Map<String, Object> configProps = new HashMap<>();
        // Assuming using the first cluster's bootstrap servers and schema registry
        ClusterConfig clusterConfig = kafkaClustersConfig.getClusters().get(0);

        configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
clusterConfig.getBootstrapServers());
        configProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
KafkaAvroSerializer.class);
        configProps.put("schema.registry.url", clusterConfig.getSchemaRegistryUrl());
        configProps.put(ProducerConfig.ACKS_CONFIG, "all");

        return new DefaultKafkaProducerFactory<>(configProps);
    }

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

---

7. Error Handling and Resiliency

Custom Error Handler

Already included in the dynamic configuration:

- **SeekToCurrentErrorHandler**: Resets offsets to retry message consumption.
- **RetryTemplate**: Configured to retry failed messages.

- **DeadLetterPublishingRecoverer**: Publishes failed messages to a dead-letter topic after retries are exhausted.

Dead Letter Topic Listener

Create a listener to monitor the dead-letter topic for further analysis.

```java
@Service
public class DeadLetterTopicListener {

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

    @KafkaListener(topics = "dead-letter-topic")
    public void listenDeadLetterTopic(ConsumerRecord<String, Object> record) {
        logger.error("Received message on dead-letter-topic: {}", record.value());
        // Implement alerting or further processing
    }
}
```

---

8. Keep-Alive Mechanism

To prevent consumers from disconnecting due to inactivity over weekends:

- **Adjust Consumer Configurations**:

  ```java
  consumerProps.put(ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG, 3600000); // 1 hour
  consumerProps.put(ConsumerConfig.HEARTBEAT_INTERVAL_MS_CONFIG, 30000); // 30 seconds
  consumerProps.put(ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG, 7200000); // 2 hours
  ```

- **Implement Scheduled Polling**:

  ```java
  @Component
  public class KeepAliveScheduler {

      @Autowired
      private KafkaClustersConfig kafkaClustersConfig;

      @Autowired
      private KafkaTemplate<String, Object> kafkaTemplate;

      @Scheduled(fixedRate = 3600000) // Every hour
      public void sendKeepAlive() {
          for (ClusterConfig clusterConfig : kafkaClustersConfig.getClusters()) {
              for (String topic : clusterConfig.getTopics()) {
                  kafkaTemplate.send(topic, "keep-alive-key", "KEEP_ALIVE");
              }
          }
  ```

```
        }
    }
```

---

9. Health Checks with Actuator

Implement custom health indicators for each Kafka cluster.

Kafka Health Indicator

```java
@Component
public class KafkaClustersHealthIndicator implements HealthIndicator {

    @Autowired
    private KafkaClustersConfig kafkaClustersConfig;

    @Override
    public Health health() {
        Map<String, Object> details = new HashMap<>();
        boolean allUp = true;

        for (ClusterConfig clusterConfig : kafkaClustersConfig.getClusters()) {
            try (AdminClient adminClient = AdminClient.create(Map.of(
                    AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG,
clusterConfig.getBootstrapServers()))) {
                DescribeClusterResult clusterInfo = adminClient.describeCluster();
                clusterInfo.clusterId().get(5, TimeUnit.SECONDS);
                details.put(clusterConfig.getName(), "UP");
            } catch (Exception e) {
                allUp = false;
                details.put(clusterConfig.getName(), "DOWN");
            }
        }

        return allUp ? Health.up().withDetails(details).build() :
Health.down().withDetails(details).build();
    }
}
```

---

10. Resiliency Patterns with Resilience4j

Use Resilience4j annotations to implement circuit breakers and retries.

Enable Resilience4j

```java
@EnableCircuitBreaker
@Configuration
public class Resilience4jConfig {
    // Configuration if needed
}
```

Apply Circuit Breaker to Message Processing

```java
@Service
public class GenericKafkaConsumerService {

    // Existing code...

    @CircuitBreaker(name = "kafkaProcessingCircuitBreaker", fallbackMethod =
"fallbackProcess")
    public void handleMessage(ConsumerRecord<String, Object> record, Acknowledgment
acknowledgment, @Header(KafkaHeaders.RECEIVED_TOPIC) String topic) {
        // Existing processing code
    }

    public void fallbackProcess(ConsumerRecord<String, Object> record, Acknowledgment
acknowledgment, String topic, Throwable throwable) {
        logger.error("Fallback processing for message: {}", record.value(), throwable);
        // Handle fallback logic, e.g., send to dead-letter topic
        sendToDeadLetterTopic(record);
        acknowledgment.acknowledge();
    }

    private void sendToDeadLetterTopic(ConsumerRecord<?, ?> record) {
        kafkaTemplate.send("dead-letter-topic", record.key(), record.value());
    }
}
```

---

11. Concurrency and Multithreading

Already set in the dynamic configuration:

```java
factory.setConcurrency(3); // Adjust as needed
```

Ensure that your message processing is thread-safe if it shares resources.

---

12. Proper Logging and Exception Handling

Logging Configuration

Use Log4j2 or Logback for logging with appropriate log levels.

Example Logback Configuration (`logback-spring.xml`):

```xml
<configuration>
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <!-- Layout configuration -->
    </appender>

    <logger name="org.springframework.kafka" level="INFO"/>
    <logger name="your.package.name" level="DEBUG"/>

    <root level="INFO">
```

```
        <appender-ref ref="STDOUT"/>
    </root>
</configuration>
```

Exception Handling

Handle exceptions within your listener method and use proper logging.

```java
public void handleMessage(ConsumerRecord<String, Object> record, Acknowledgment
acknowledgment, @Header(KafkaHeaders.RECEIVED_TOPIC) String topic) {
    try {
        // Process the message
    } catch (SpecificException e) {
        logger.error("Specific exception occurred: {}", e.getMessage());
        // Handle specific exception
    } catch (Exception e) {
        logger.error("Unhandled exception occurred: {}", e.getMessage(), e);
        throw e; // Let the error handler handle it
    }
}
```

---

13. Support for Avro, String, and JSON Serialization

Deserializer Configuration

Modify the deserializer in the consumer properties to handle multiple serialization formats.

Option 1: Use ByteArrayDeserializer and determine the type at runtime.

```java
consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
ByteArrayDeserializer.class);
```

In your listener:

```java
public void handleMessage(ConsumerRecord<String, byte[]> record, Acknowledgment
acknowledgment, @Header(KafkaHeaders.RECEIVED_TOPIC) String topic) {
    byte[] data = record.value();
    // Determine the type of data (e.g., by inspecting headers or data)
    Object deserializedObject = deserializeData(data);
    // Process the deserialized object
}
```

Option 2: Use a custom deserializer that can handle multiple types.

Create a custom deserializer that attempts to deserialize using different formats.

14. Handling Inactivity and Rebalancing

To prevent rebalancing due to inactivity:

- **Set `max.poll.interval.ms` to a higher value** to accommodate long processing times or inactivity.
- **Implement a Keep-Alive Consumer**

```java
@Component
public class KeepAliveConsumer {

    @KafkaListener(topics = "#{'${kafka.clusters.*.topics}'.split(',')}", autoStartup
= "true")
    public void keepAliveListener() {
        // This listener does nothing but keeps the consumer alive
    }
}
```

- **Adjust Consumer Configurations**

```java
consumerProps.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 1);
```

---

15. Testing and Validation

Ensure to write unit tests and integration tests to validate the behavior.

- Use **EmbeddedKafka** for integration testing.

Complete Code Structure

Due to space constraints, only key components are provided. You can assemble the code into a full application with the following structure:

- **Application Class**: Contains the `main` method to run the Spring Boot application.

```java
@SpringBootApplication
public class KafkaApplication {

    public static void main(String[] args) {
        SpringApplication.run(KafkaApplication.class, args);
    }
}
```

- **Configuration Classes**: `` `KafkaClustersConfig` ``, `` `DynamicKafkaConfiguration` ``, `` `KafkaProducerConfig` ``, etc.

- **Service Classes**: `` `GenericKafkaConsumerService` ``, `` `DeadLetterTopicListener` ``, `` `KeepAliveScheduler` ``.

- **Model Classes**: `` `ClusterConfig` ``.

- **Resources**:
  - `` `application.yml` `` for configurations.
  - Logging configuration file.

- **Error Handling**: Implemented in `` `DynamicKafkaConfiguration` `` and `` `GenericKafkaConsumerService` ``.

- **Resiliency**: Using Resilience4j in `` `GenericKafkaConsumerService` ``.

- **Health Checks**: `` `KafkaClustersHealthIndicator` ``.

- **Testing**: Write test classes using JUnit and `` `@SpringBootTest` ``.

Final Notes

- **Security**: If your Kafka clusters require authentication (e.g., SSL, SASL), include the necessary properties in your configuration.

- **Scaling**: Monitor your application's performance and adjust concurrency and resource allocations as needed.

- **Monitoring**: Use tools like **Spring Boot Actuator**, **Prometheus**, and **Grafana** to monitor application health and metrics.

- **Documentation**: Keep your code well-documented and maintain clear instructions for configuration.

By following this approach, you can build a robust, resilient, and highly available Spring Boot application that can handle any number of consumers from different Kafka clusters using a generic listener and factory.

Feel free to customize and extend the code to fit your specific requirements. Let me know if you need further assistance or clarification on any part!

Can i download it as zip

I'm glad to hear that you're interested in implementing the solution! Unfortunately, I don't have the capability to create or send files, including ZIP archives. However, I can guide you through assembling the code into a complete Spring Boot project on your local machine.

Here's how you can proceed to build the application with all the aspects of resiliency and high availability:

---

1. Set Up a New Spring Boot Project

You can create a new Spring Boot project using one of the following methods:

- **Spring Initializr Website

Message need to be transmitted to multiple eventbridge

Certainly! If you need to transmit messages to multiple **AWS EventBridge** instances from your Spring Boot application, you can integrate the AWS SDK for Java into your application. This allows your application to send events to EventBridge buses, enabling you to build scalable and event-driven architectures.

In the context of your existing Kafka consumer application, you can consume messages from Kafka and then forward them to multiple EventBridge buses as required.

Overview

- **Goal**: Modify the Spring Boot application to transmit messages to multiple AWS EventBridge instances.
- **Approach**:
  - Use AWS SDK for Java to interact with EventBridge.
  - Configure multiple EventBridge clients if necessary.
  - Ensure the application handles retries, error handling, and resiliency when communicating with EventBridge.
  - Keep the application scalable and maintain high availability.

Prerequisites

1. **AWS Credentials**: Ensure that your application has access to AWS credentials with permissions to publish events to EventBridge.

2. **AWS SDK for Java Dependency**: Add the AWS SDK dependency to your project.

3. **EventBridge Buses**: Identify the EventBridge event buses you need to send messages to.

1. Add AWS SDK Dependency

Add the AWS SDK for EventBridge to your ``pom.xml``:

```xml
<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>eventbridge</artifactId>
    <version>2.20.14</version> <!-- Use the latest version -->
</dependency>
```

---

2. Configure AWS Credentials

AWS SDK for Java will automatically pick up credentials from:

- Environment variables (``AWS_ACCESS_KEY_ID``, ``AWS_SECRET_ACCESS_KEY``)
- AWS configuration files (``~/.aws/credentials`` and ``~/.aws/config``)
- IAM roles if running on AWS EC2 or ECS instances.

Alternatively, you can configure credentials programmatically.

---

3. Create EventBridge Client Configuration

EventBridgeClientFactory

Create a factory to manage multiple ``EventBridgeClient`` instances for different regions or configurations.

```java
import software.amazon.awssdk.services.eventbridge.EventBridgeClient;
import software.amazon.awssdk.regions.Region;
import org.springframework.stereotype.Component;

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

@Component
public class EventBridgeClientFactory {

    private Map<String, EventBridgeClient> clients = new HashMap<>();

    public EventBridgeClient getClient(String region) {
        return clients.computeIfAbsent(region, this::createClient);
    }

    private EventBridgeClient createClient(String region) {
        return EventBridgeClient.builder()
                .region(Region.of(region))
                .build();
    }
}
```

4. Update Configuration for Multiple EventBridge Buses

Add EventBridge configurations to your `` `application.yml` `` or `` `application.properties` ``.

application.yml

```yaml
eventbridge:
  buses:
    - name: bus1
      region: us-east-1
      eventBusName: custom-bus1
    - name: bus2
      region: us-west-2
      eventBusName: custom-bus2
    # Add more buses as needed
```

EventBusConfig

```java
public class EventBusConfig {
    private String name;
    private String region;
    private String eventBusName;

    // Getters and Setters
}
```

EventBridgeConfig

```java
@Configuration
@ConfigurationProperties(prefix = "eventbridge")
public class EventBridgeConfig {
    private List<EventBusConfig> buses;

    // Getters and Setters
}
```

---

5. Implement Message Publisher Service

Create a service to publish messages to multiple EventBridge buses.

```java
import software.amazon.awssdk.services.eventbridge.EventBridgeClient;
import software.amazon.awssdk.services.eventbridge.model.PutEventsRequest;
import software.amazon.awssdk.services.eventbridge.model.PutEventsRequestEntry;
import software.amazon.awssdk.services.eventbridge.model.PutEventsResponse;
import software.amazon.awssdk.core.SdkBytes;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class EventBridgePublisher {

    @Autowired
    private EventBridgeClientFactory eventBridgeClientFactory;

    @Autowired
    private EventBridgeConfig eventBridgeConfig;

    public void publishEvent(Object eventDetail) {
        for (EventBusConfig busConfig : eventBridgeConfig.getBuses()) {
            try {
                EventBridgeClient client =
eventBridgeClientFactory.getClient(busConfig.getRegion());

                PutEventsRequestEntry requestEntry = PutEventsRequestEntry.builder()
                        .eventBusName(busConfig.getEventBusName())
                        .detailType("myDetailType")
                        .source("myApp")
                        .detail(convertEventDetailToJson(eventDetail))
                        .build();

                PutEventsRequest request = PutEventsRequest.builder()
                        .entries(requestEntry)
                        .build();

                PutEventsResponse response = client.putEvents(request);

                if (response.failedEntryCount() > 0) {
                    // Handle failures
                    System.err.println("Failed to send event to EventBridge bus: " +
busConfig.getEventBusName());
                }

            } catch (Exception e) {
                // Proper error handling and logging
                System.err.println("Exception while sending event to EventBridge: " +
e.getMessage());
                // Implement retry logic if necessary
            }
        }
    }

    private String convertEventDetailToJson(Object eventDetail) {
        // Convert your eventDetail object to a JSON string
        // You can use your preferred JSON library (e.g., Jackson)
        ObjectMapper objectMapper = new ObjectMapper();
        try {
            return objectMapper.writeValueAsString(eventDetail);
        } catch (JsonProcessingException e) {
            throw new RuntimeException("Failed to convert event detail to JSON", e);
        }
    }
}
```

---

6. Update Kafka Consumer to Publish to EventBridge

Modify your Kafka listener to use the `EventBridgePublisher` service.

```java
@Service
public class GenericKafkaConsumerService {

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

    @Autowired
    private ApplicationContext applicationContext;

    @Autowired
    private KafkaClustersConfig kafkaClustersConfig;

    @Autowired
    private KafkaListenerEndpointRegistry kafkaListenerEndpointRegistry;

    @Autowired
    private EventBridgePublisher eventBridgePublisher;

    @PostConstruct
    public void setupGenericListeners() {
        // Existing code to register listeners
    }

    // Existing methods...

    public void handleMessage(ConsumerRecord<String, Object> record, Acknowledgment
acknowledgment, @Header(KafkaHeaders.RECEIVED_TOPIC) String topic) {
        try {
            // Process the message
            logger.info("Received message from topic {}: {}", topic, record.value());

            // Publish to EventBridge
            eventBridgePublisher.publishEvent(record.value());

            // Acknowledge the message
            acknowledgment.acknowledge();
        } catch (Exception e) {
            logger.error("Error processing message: {}", record.value(), e);
            throw e; // Let the error handler handle the exception
        }
    }
}
```

---

7. Error Handling and Retry Mechanism

Implement proper error handling when publishing events to EventBridge.

- **Retry Logic**: Use Resilience4j or Spring Retry to implement retries when publishing to EventBridge fails due to transient errors.

Example with Resilience4j

Add Resilience4j annotations to your `EventBridgePublisher` methods.

```java
import io.github.resilience4j.retry.annotation.Retry;
import io.github.resilience4j.retry.annotation.Backoff;

@Service
public class EventBridgePublisher {
    // Existing code...

    @Retry(name = "eventBridgePublisherRetry", fallbackMethod = "publishEventFallback")
    public void publishEvent(Object eventDetail) {
        // Existing code...
    }

    public void publishEventFallback(Object eventDetail, Throwable throwable) {
        // Handle fallback logic, such as logging the error or storing the event for later
processing
        logger.error("Failed to publish event after retries: {}", throwable.getMessage());
        // Optionally, send to a dead-letter queue or persist in a database
    }
}
```

Resilience4j Configuration

Add configurations for Resilience4j in `` `application.yml` ``:

```yaml
resilience4j:
  retry:
    instances:
      eventBridgePublisherRetry:
        maxAttempts: 3
        waitDuration: 2s
```

---

8. Ensure High Availability

- **Stateless Services**: Ensure your Spring Boot application is stateless so that it can be scaled horizontally.
- **Load Balancing**: Deploy multiple instances behind a load balancer.
- **Graceful Shutdown**: Implement graceful shutdown to handle in-flight messages.

---

9. Implement Health Checks

Add health checks to monitor the status of the EventBridge clients.

EventBridge Health Indicator

```java
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;
import software.amazon.awssdk.services.eventbridge.EventBridgeClient;
import software.amazon.awssdk.services.eventbridge.model.DescribeEventBusRequest;

@Component
public class EventBridgeHealthIndicator implements HealthIndicator {

    @Autowired
    private EventBridgeClientFactory eventBridgeClientFactory;
    @Autowired
    private EventBridgeConfig eventBridgeConfig;

    @Override
    public Health health() {
        for (EventBusConfig busConfig : eventBridgeConfig.getBuses()) {
            try {
                EventBridgeClient client =
eventBridgeClientFactory.getClient(busConfig.getRegion());
                client.describeEventBus(DescribeEventBusRequest.builder()
                        .name(busConfig.getEventBusName())
                        .build());
            } catch (Exception e) {
                return Health.down()
                        .withDetail("EventBridge Bus", busConfig.getEventBusName())
                        .withException(e)
                        .build();
            }
        }
        return Health.up().build();
    }
}
```

10. Security Considerations

- **Credentials Management**: Use AWS IAM roles or AWS Secrets Manager to manage credentials securely.
- **Encryption**: Ensure data is encrypted in transit and at rest.

11. Logging and Monitoring

- **Logging**: Use structured logging to capture important information, including EventBridge responses and any errors.
- **Monitoring**: Integrate with monitoring tools like CloudWatch to track metrics and logs.

12. Testing

- **Unit Tests**: Mock the EventBridge client to test your publisher logic.
- **Integration Tests**: Use AWS SDK's local stack or test containers for integration tests.

13. Handling Dynamic EventBridge Buses

If the number of EventBridge buses is dynamic and may change at runtime, you can:

- **Externalize Configuration**: Keep the configuration in an external source like a database or configuration service.
- **Dynamic Client Creation**: Adjust the `EventBridgeClientFactory` to create clients on-the-fly based on updated configurations.

14. Combining with Kafka Consumers

Your application now:

- Consumes messages from multiple Kafka clusters.
- Processes messages using a generic listener.
- Transmits messages to multiple EventBridge buses.
- Handles errors, retries, and resiliency patterns.
- Maintains high availability and scalability.

Sample Project Structure

- **src/main/java**
  - **com.example.kafkaeventbridge**
    - **Application.java**: Main application class.
    - **config**
      - **KafkaClustersConfig.java**
      - **EventBridgeConfig.java**
    - **factory**
      - **EventBridgeClientFactory.java**
    - **service**
      - **GenericKafkaConsumerService.java**
      - **EventBridgePublisher.java**
    - **health**
      - **EventBridgeHealthIndicator.java**
    - **model**
      - **ClusterConfig.java**
      - **EventBusConfig.java**
- **src/main/resources**
  - **application.yml**

Final Notes

- **Permissions**: Ensure the IAM role or user has the `` `events:PutEvents` `` permission for each EventBridge bus.
- **Error Handling**: Always handle exceptions when interacting with external services.
- **Idempotency**: If necessary, implement idempotency to avoid processing duplicate messages.
- **Serialization**: Ensure the messages sent to EventBridge are correctly serialized as JSON strings.
- **Schema Validation**: If using schemas in EventBridge, validate the messages against the schemas.

Next Steps

1. **Set Up AWS Resources**: Create the necessary EventBridge buses in AWS.

2. **Configure AWS Credentials**: Ensure your application can authenticate with AWS.

3. **Implement the Code**: Use the code snippets provided to implement the solution.

4. **Test the Application**: Test with sample messages to ensure they are correctly forwarded to EventBridge.

5. **Deploy and Monitor**: Deploy the application and monitor its performance and health.

Feel free to ask if you need further clarification or assistance with any part of the implementation!

multiple eventbridge
DLQ for eventbridge on failover

Certainly! It sounds like you need to extend your Spring Boot application to:

1. **Transmit messages to multiple AWS EventBridge instances**.
2. Implement a **Dead Letter Queue (DLQ)** mechanism for EventBridge in case of failures during message transmission.

Below, I'll guide you through the process of configuring your application to send messages to multiple EventBridge buses and implement a DLQ using Amazon SQS. We'll cover:

- Setting up AWS credentials and dependencies.
- Configuring multiple EventBridge clients.
- Implementing message publishing with retries.
- Setting up a DLQ using Amazon SQS.
- Handling failures and sending messages to the DLQ.
- Ensuring high availability and resiliency.

Prerequisites

1. **AWS Account**: Ensure you have access to an AWS account.
2. **AWS Credentials**: Set up AWS credentials with permissions to use EventBridge and SQS.
3. **Spring Boot Application**: Your existing Spring Boot application where you need to implement these features.
4. **Maven or Gradle**: For managing project dependencies.

---

1. Add AWS SDK Dependencies

Include the necessary AWS SDK dependencies in your `pom.xml`:

xml

```xml
<dependencies>
    <!-- AWS SDK for EventBridge -->
```

```
        <dependency>
            <groupId>software.amazon.awssdk</groupId>
            <artifactId>eventbridge</artifactId>
            <version>2.20.14</version> <!-- Use the latest version -->
        </dependency>

        <!-- AWS SDK for SQS (for DLQ) -->
        <dependency>
            <groupId>software.amazon.awssdk</groupId>
            <artifactId>sqs</artifactId>
            <version>2.20.14</version>
        </dependency>

        <!-- Jackson Databind for JSON processing -->
        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
        </dependency>

        <!-- Resilience4j for retries and circuit breakers -->
        <dependency>
            <groupId>io.github.resilience4j</groupId>
            <artifactId>resilience4j-spring-boot2</artifactId>
            <version>1.7.1</version>
        </dependency>

        <!-- Other dependencies -->
        <!-- ... -->
    </dependencies>
```

---

2. Configure AWS Credentials

Ensure your application can access AWS credentials. The AWS SDK looks for credentials in the following order:

- Environment variables (`` `AWS_ACCESS_KEY_ID` `` and `` `AWS_SECRET_ACCESS_KEY` ``)
- Java system properties
- Default credential profiles file (`` `~/.aws/credentials` ``)
- Amazon ECS container credentials
- Instance profile credentials (if running on EC2)

---

3. Define Multiple EventBridge Buses and DLQ Configurations

application.yml

Add configurations for multiple EventBridge buses and the DLQ:

```yaml
aws:
  eventbridge:
    buses:
      - name: bus1
        region: us-east-1
        eventBusName: custom-bus1
      - name: bus2
        region: us-west-2
        eventBusName: custom-bus2
  sqs:
    dlq:
      region: us-east-1
      queueUrl: https://sqs.us-east-1.amazonaws.com/123456789012/YourDLQ
```

Configuration Classes

EventBusConfig.java

```java
public class EventBusConfig {
    private String name;
    private String region;
    private String eventBusName;

    // Getters and Setters
}
```

DLQConfig.java

```java
public class DLQConfig {
    private String region;
    private String queueUrl;

    // Getters and Setters
}
```

AWSConfig.java

```java
@Configuration
@ConfigurationProperties(prefix = "aws")
public class AWSConfig {
    private EventBridgeConfig eventbridge;
    private SQSConfig sqs;

    // Getters and Setters

    public static class EventBridgeConfig {
        private List<EventBusConfig> buses;

        // Getters and Setters
    }

    public static class SQSConfig {
        private DLQConfig dlq;

        // Getters and Setters
    }
}
```

---

4. Create AWS Clients Factory

Create a factory to manage EventBridge and SQS clients for different regions.

AWSClientFactory.java

```java
import software.amazon.awssdk.services.eventbridge.EventBridgeClient;
import software.amazon.awssdk.services.sqs.SqsClient;
import software.amazon.awssdk.regions.Region;
import org.springframework.stereotype.Component;

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

@Component
public class AWSClientFactory {

    private Map<String, EventBridgeClient> eventBridgeClients = new HashMap<>();
    private Map<String, SqsClient> sqsClients = new HashMap<>();

    public EventBridgeClient getEventBridgeClient(String region) {
        return eventBridgeClients.computeIfAbsent(region, this::createEventBridgeClient);
    }

    public SqsClient getSqsClient(String region) {
        return sqsClients.computeIfAbsent(region, this::createSqsClient);
    }

    private EventBridgeClient createEventBridgeClient(String region) {
        return EventBridgeClient.builder()
                .region(Region.of(region))
                .build();
    }

    private SqsClient createSqsClient(String region) {
        return SqsClient.builder()
                .region(Region.of(region))
                .build();
    }
}
```

---

5. Implement the EventBridge Publisher with DLQ Support

EventBridgePublisher.java

```java
import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import io.github.resilience4j.retry.annotation.Retry;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Service;
import software.amazon.awssdk.services.eventbridge.EventBridgeClient;
import software.amazon.awssdk.services.eventbridge.model.PutEventsRequest;
import software.amazon.awssdk.services.eventbridge.model.PutEventsRequestEntry;
import software.amazon.awssdk.services.eventbridge.model.PutEventsResponse;
import software.amazon.awssdk.services.sqs.SqsClient;
import software.amazon.awssdk.services.sqs.model.SendMessageRequest;

import java.util.List;

@Service
```

```
public class EventBridgePublisher {

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

    private final AWSConfig awsConfig;
    private final AWSClientFactory awsClientFactory;
    private final ObjectMapper objectMapper;

    public EventBridgePublisher(AWSConfig awsConfig, AWSClientFactory awsClientFactory,
ObjectMapper objectMapper) {
        this.awsConfig = awsConfig;
        this.awsClientFactory = awsClientFactory;
        this.objectMapper = objectMapper;
    }

    @Retry(name = "eventBridgePublisherRetry", fallbackMethod = "publishEventFallback")
    public void publishEvent(Object eventDetail) {
        List<AWSConfig.EventBridgeConfig.EventBusConfig> buses =
awsConfig.getEventbridge().getBuses();
        for (AWSConfig.EventBridgeConfig.EventBusConfig busConfig : buses) {
            try {
                EventBridgeClient client =
awsClientFactory.getEventBridgeClient(busConfig.getRegion());

                String detailJson = convertEventDetailToJson(eventDetail);

                PutEventsRequestEntry requestEntry = PutEventsRequestEntry.builder()
                        .eventBusName(busConfig.getEventBusName())
                        .detailType("myDetailType")
                        .source("myApp")
                        .detail(detailJson)
                        .build();

                PutEventsRequest request = PutEventsRequest.builder()
                        .entries(requestEntry)
                        .build();

                PutEventsResponse response = client.putEvents(request);

                if (response.failedEntryCount() > 0) {
                    logger.error("Failed to send event to EventBridge bus: {}",
busConfig.getEventBusName());
                    throw new RuntimeException("Failed to send event to EventBridge bus: "
+ busConfig.getEventBusName());
                } else {
                    logger.info("Event sent to EventBridge bus: {}",
busConfig.getEventBusName());
                }

            } catch (Exception e) {
                logger.error("Exception while sending event to EventBridge bus {}: {}",
busConfig.getEventBusName(), e.getMessage());
                throw e; // Trigger retry
            }
        }
    }

    private String convertEventDetailToJson(Object eventDetail) throws
JsonProcessingException {
        return objectMapper.writeValueAsString(eventDetail);
    }

    // Fallback method after retries are exhausted
    public void publishEventFallback(Object eventDetail, Throwable throwable) {
        logger.error("Failed to publish event after retries: {}", throwable.getMessage());
        sendToDLQ(eventDetail);
    }

    private void sendToDLQ(Object eventDetail) {
        try {
            AWSConfig.SQSConfig.DLQConfig dlqConfig = awsConfig.getSqs().getDlq();
            SqsClient sqsClient = awsClientFactory.getSqsClient(dlqConfig.getRegion());

            String messageBody = convertEventDetailToJson(eventDetail);
            SendMessageRequest sendMessageRequest = SendMessageRequest.builder()
                    .queueUrl(dlqConfig.getQueueUrl())
                    .messageBody(messageBody)
                    .build();

            sqsClient.sendMessage(sendMessageRequest);
            logger.info("Message sent to DLQ: {}", dlqConfig.getQueueUrl());

        } catch (Exception e) {
            logger.error("Failed to send message to DLQ: {}", e.getMessage());
            // Consider persisting the message to a database or file system for manual
recovery
        }
    }
}
```

Resilience4j Configuration

Add configurations for Resilience4j in `application.yml`:

```yaml
resilience4j:
  retry:
    instances:
      eventBridgePublisherRetry:
        maxAttempts: 3
        waitDuration: 2s
        retryExceptions:
          - java.io.IOException
          - software.amazon.awssdk.core.exception.SdkException
```

---

6. Update the Kafka Consumer to Use the Publisher

Modify your Kafka listener to use the `EventBridgePublisher` service.

GenericKafkaConsumerService.java

```java
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.springframework.kafka.support.Acknowledgment;
import org.springframework.kafka.support.KafkaHeaders;
import org.springframework.messaging.handler.annotation.Header;

@Service
public class GenericKafkaConsumerService {

    private final EventBridgePublisher eventBridgePublisher;

    public GenericKafkaConsumerService(EventBridgePublisher eventBridgePublisher) {
        this.eventBridgePublisher = eventBridgePublisher;
    }

    @KafkaListener(topics = "#{@kafkaTopicProvider.getAllTopics()}", containerFactory =
"kafkaListenerContainerFactory")
    public void handleMessage(ConsumerRecord<String, Object> record, Acknowledgment
acknowledgment,
                              @Header(KafkaHeaders.RECEIVED_TOPIC) String topic) {
        try {
            // Process the message
            logger.info("Received message from topic {}: {}", topic, record.value());

            // Publish to EventBridge
            eventBridgePublisher.publishEvent(record.value());

            // Acknowledge the message
            acknowledgment.acknowledge();

        } catch (Exception e) {
            logger.error("Error processing message: {}", record.value(), e);
            throw e; // Let the error handler handle the exception
        }
    }
}
```

---

7. Provide Topics Dynamically

Create a bean to provide all topics dynamically.

KafkaTopicProvider.java

```java
import org.springframework.stereotype.Component;

import java.util.List;
import java.util.stream.Collectors;

@Component
public class KafkaTopicProvider {

    private final KafkaClustersConfig kafkaClustersConfig;

    public KafkaTopicProvider(KafkaClustersConfig kafkaClustersConfig) {
        this.kafkaClustersConfig = kafkaClustersConfig;
    }

    public List<String> getAllTopics() {
        return kafkaClustersConfig.getClusters().stream()
                .flatMap(cluster -> cluster.getTopics().stream())
                .collect(Collectors.toList());
    }
}
```

---

8. Error Handling and High Availability

- **Retry Mechanism**: Implemented using Resilience4j's `` `@Retry` `` annotation.
- **Circuit Breaker**: Optionally, you can add a circuit breaker to prevent overwhelming the EventBridge service during outages.
- **Dead Letter Queue (DLQ)**: Messages that fail after retries are sent to an Amazon SQS queue for later processing.

---

9. Setting Up AWS Resources

Create EventBridge Buses

In AWS, create the custom EventBridge buses specified in your configuration.

Create SQS Queue for DLQ

Create an Amazon SQS queue to act as the DLQ.

---

10. Security and Permissions

Ensure that the IAM role or user associated with your application has the necessary permissions:

- **EventBridge Permissions**:
    - ``` `events:PutEvents` ``` for the EventBridge buses.
- **SQS Permissions**:
    - ``` `sqs:SendMessage` ``` for the DLQ queue.

---

11. Health Checks and Monitoring

Health Indicator for EventBridge and SQS

Implement health indicators to monitor the status of EventBridge and SQS.

EventBridgeHealthIndicator.java

```java
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;
import software.amazon.awssdk.services.eventbridge.EventBridgeClient;
import software.amazon.awssdk.services.eventbridge.model.DescribeEventBusRequest;

@Component
public class EventBridgeHealthIndicator implements HealthIndicator {

    private final AWSConfig awsConfig;
    private final AWSClientFactory awsClientFactory;

    public EventBridgeHealthIndicator(AWSConfig awsConfig, AWSClientFactory
awsClientFactory) {
        this.awsConfig = awsConfig;
        this.awsClientFactory = awsClientFactory;
    }

    @Override
    public Health health() {
        for (AWSConfig.EventBridgeConfig.EventBusConfig busConfig :
awsConfig.getEventbridge().getBuses()) {
            try {
                EventBridgeClient client =
awsClientFactory.getEventBridgeClient(busConfig.getRegion());
                client.describeEventBus(DescribeEventBusRequest.builder()
                        .name(busConfig.getEventBusName())
                        .build());
            } catch (Exception e) {
                return Health.down()
                        .withDetail("EventBridge Bus", busConfig.getEventBusName())
```

```
                        .withException(e)
                        .build();
            }
        }
        return Health.up().build();
    }
}
```

SQSHealthIndicator.java

```java
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;
import software.amazon.awssdk.services.sqs.SqsClient;
import software.amazon.awssdk.services.sqs.model.GetQueueAttributesRequest;

@Component
public class SQSHealthIndicator implements HealthIndicator {

    private final AWSConfig awsConfig;
    private final AWSClientFactory awsClientFactory;

    public SQSHealthIndicator(AWSConfig awsConfig, AWSClientFactory awsClientFactory) {
        this.awsConfig = awsConfig;
        this.awsClientFactory = awsClientFactory;
    }

    @Override
    public Health health() {
        try {
            AWSConfig.SQSConfig.DLQConfig dlqConfig = awsConfig.getSqs().getDlq();
            SqsClient sqsClient = awsClientFactory.getSqsClient(dlqConfig.getRegion());

            sqsClient.getQueueAttributes(GetQueueAttributesRequest.builder()
                    .queueUrl(dlqConfig.getQueueUrl())
                    .attributeNames("QueueArn")
                    .build());

            return Health.up().build();
        } catch (Exception e) {
            return Health.down()
                    .withDetail("SQS DLQ", awsConfig.getSqs().getDlq().getQueueUrl())
                    .withException(e)
                    .build();
        }
    }
}
```

---

12. Logging and Monitoring

- **Structured Logging**: Use a logging framework like SLF4J with Logback or Log4j2.
- **Log Important Events**: Log successes and failures when sending messages to EventBridge and DLQ.
- **Monitoring Tools**: Use AWS CloudWatch for monitoring metrics and logs.

---

13. Testing

- **Unit Tests**: Mock AWS SDK clients to test your application logic.
- **Integration Tests**: Use AWS SDK's `LocalStack` or similar tools to simulate AWS services for testing.

---

14. Implementing Resiliency Patterns

- **Circuit Breaker**: Prevent continuous attempts to send messages when EventBridge is down.

Applying Circuit Breaker

Add `@CircuitBreaker` to your `publishEvent` method.

```java
import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;

@Service
public class EventBridgePublisher {

    // Existing code...

    @CircuitBreaker(name = "eventBridgePublisherCircuitBreaker", fallbackMethod =
"publishEventFallback")
    @Retry(name = "eventBridgePublisherRetry", fallbackMethod = "publishEventFallback")
    public void publishEvent(Object eventDetail) {
        // Existing code...
    }

    // Existing methods...
}
```

Resilience4j Configuration for Circuit Breaker

```yaml
resilience4j:
  circuitbreaker:
    instances:
      eventBridgePublisherCircuitBreaker:
        registerHealthIndicator: true
        slidingWindowSize: 5
        failureRateThreshold: 50
        waitDurationInOpenState: 30s
```

---

15. Security Considerations

- **Credentials Management**: Use AWS IAM roles, AWS Secrets Manager, or AWS Systems Manager Parameter Store.
- **Encryption**: Ensure data is encrypted in transit (HTTPS) and at rest (SQS encryption).

---

16. Complete Project Structure

- **src/main/java**
    - **com.example.yourapp**
        - **Application.java**: Main application class.
        - **config**
            - **AWSConfig.java**
            - **KafkaClustersConfig.java**
        - **factory**
            - **AWSClientFactory.java**
        - **service**
            - **EventBridgePublisher.java**
            - **GenericKafkaConsumerService.java**
        - **health**
            - **EventBridgeHealthIndicator.java**
            - **SQSHealthIndicator.java**
        - **model**
            - **ClusterConfig.java**
            - **EventBusConfig.java**
            - **DLQConfig.java**
        - **provider**
            - **KafkaTopicProvider.java**
- **src/main/resources**
    - **application.yml**
    - **logback-spring.xml** (for logging configuration)

---

17. Next Steps

1. **Implement the Code**: Integrate the provided code snippets into your application.
2. **Set Up AWS Resources**: Ensure all EventBridge buses and SQS queues are created and configured correctly.
3. **Configure IAM Permissions**: Update IAM policies to grant necessary permissions.
4. **Test the Application**: Perform end-to-end testing to verify messages are sent to EventBridge and the DLQ works as expected.
5. **Deploy and Monitor**: Deploy the application and monitor its performance, health, and logs.

Additional Considerations

- **Idempotency**: Implement idempotency tokens if required to prevent duplicate processing.
- **Message Attributes**: Include additional attributes or metadata if needed for downstream processing.
- **Event Schema**: Define and validate event schemas to ensure data consistency.
- **Scaling**: Ensure your application can scale horizontally to handle increased load.

Summary

By following this guide, you've extended your Spring Boot application to:

- **Transmit messages to multiple AWS EventBridge instances**.

- Implement a **Dead Letter Queue (DLQ)** using Amazon SQS for messages that fail to be sent after retries.
- Incorporate **resiliency patterns** like retries and circuit breakers using Resilience4j.
- Ensure **high availability** through proper error handling and scalable design.
- Maintain **observability** with health checks and logging.

Feel free to reach out if you have any questions or need further assistance with any part of the implementation!

ChatGPT can make mistakes. Check important info.