Spring for Apache Pulsar

Soby Chacko, Chris Bono, Alexander Preuß

Table of Contents

Chapter 1. Preface

This project provides a basic Spring-friendly API for developing Apache Pulsar applications.

On a very high-level, Spring for Apache Pulsar provides a `PulsarTemplate` for publishing to a Pulsar topic and a `PulsarListener` annotation for consuming from it. In addition, it also provides various convenient APIs for Spring developers to ramp up their development journey into Apache Pulsar.

Chapter 2. Introduction

This first part of the reference documentation is a high-level quick-tour of Spring for Apache Pulsar.

2.1. Quick Tour

In this section, we will take a quick tour of Spring for Apache Pulsar.

2.1.1. Minimum Supported Versions

The minimum supported versions for the underlying libraries required by the framework are as follows:

Library	Version
Java	17
Apache Pulsar	2.10.0
Spring Boot	3.0.0
Spring Framework	6.0.0
Gradle	7.x (7.5 or later)

2.2. Quick Sample

In the following sample Spring Boot application, we show how to write a publisher and consumer using Spring for Apache Pulsar. This is a complete application and does not require any additional configuration as long as you have Pulsar cluster running on the default location - `localhost:6650`.

We recommend the usage of a Spring-Boot-First-Approach for Spring for Apache Pulsar based application as that simplifies things tremendously. To encourage this, a `spring-pulsar-spring-boot-starter` module is published that can easily be consumed by an application as a dependency.

2.2.1. Dependencies

Spring Boot applications only need the `spring-pulsar-spring-boot-starter` dependency. The following shows how to define the dependency for Maven and Gradle, respectively:

Maven

```
<dependencies>
    <dependency>
        <groupId>org.springframework.pulsar</groupId>
        <artifactId>spring-pulsar-spring-boot-starter</artifactId>
        <version>0.1.0-SNAPSHOT</version>
    </dependency>
</dependencies>
```

Gradle

```
dependencies {
    implementation 'org.springframework.pulsar:spring-pulsar-spring-boot-
starter:0.1.0-SNAPSHOT'
}
```

Non-GA Versions

Snapshot or Milestone versions of the dependency can be found on the Spring Artifactory repository. The following shows how to define the repositories for Maven and Gradle, respectively:

Maven

```
<repositories>
    ...
    <repository>
        <id>spring-milestones</id>
        <name>Spring Milestones</name>
        <url>https://repo.spring.io/milestone</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </repository>
    <repository>
        <id>spring-snapshots</id>
        <name>Spring Snapshots</name>
        <url>https://repo.spring.io/snapshot</url>
        <releases>
            <enabled>false</enabled>
        </releases>
    </repository>
</repositories>
```

Gradle

```
repositories {
  ...
  maven { url 'https://repo.spring.io/milestone' }
  maven { url 'https://repo.spring.io/snapshot' }
}
```

2.2.2. Application Code

```
@SpringBootApplication
public class PulsarBootHelloWorld {

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

    @Bean
    ApplicationRunner runner(PulsarTemplate<String> pulsarTemplate) {
        return (args) -> pulsarTemplate.send("hello-pulsar", "Hello Pulsar World!");
    }

    @PulsarListener(subscriptionName = "hello-pulsar-subscription", topics = "hello-
pulsar")
    void listen(String message) {
        System.out.println("Message Received: " + message);
    }
}
```

Let us go through the higher-level details of this application quickly. Later on in this documentation, we will see these components in much more detail.

In the sample above, we are heavily relying on Spring Boot auto-configuration. Spring Boot auto-configures several components for our application. It automatically provides a `PulsarClient` for the application which is used by both the producer and the consumer.

`PulsarTemplate` also is auto-configured by Spring Boot which we inject in the application and start sending records to a Pulsar topic. The application sends messages to a topic named `hello-pulsar`. Note that the application does not specify any schema information. That is because Spring for Apache Pulsar library automatically infers the schema type from the type of the data that you are sending.

We use `PulsarListener` annotation to consume from the `hello-pulsar` topic where we publish the data. `PulsarListener` is a convenient annotation that wraps the message listener container infrastructure in Spring for Apache Pulsar. Behind the scenes, it creates a message listener container which creates and manages the Pulsar consumer. As with a regular Pulsar consumer, the default subscription type when using `PulsarListener` is the `Exclusive` mode. As records are published in to the `hello-pulsar` topic, the `Pulsarlistener` consumes them and prints them on the

console. Here also, the framework infers the schema type used from the data type that the `PulsarListner` method uses as the payload - `String` in this case.

2.3. Building the Project

If you have cloned the project locally, follow these steps in order to build the project from the soure code.

Spring for Apache Pulsar uses Gradle as it's build tool. Run the following command to do a full build of the project:

```
./gradlew clean build
```

You can build without running tests by using the following command:

```
./gradlew clean build -x test
```

Chapter 3. Reference

This part of the reference documentation goes through the details of the various components in Spring for Apache Pulsar.

3.1. Using Spring for Apache Pulsar

This section offers detailed explanations of the various concerns that impact using Spring for Apache Pulsar. For a quick but less detailed introduction, see Introduction.

3.1.1. Pulsar Client

When using the Pulsar Spring Boot Starter, you get the `PulsarClient` auto-configured. This is done through a factory bean called `PulsarClientFactoryBean`, which takes a configuration object `PulsarClientConfiguration`. By default, the application tries to connect to a local Pulsar instance at `pulsar://localhost:6650`. However, there are many application properties available to configure the client.

Click here to view the available **Pulsar Client Properties**.

Name	Description	Default Value
`spring.pulsar.client.auth-params`	Authentication parameter(s) as a JSON encoded string.	
`spring.pulsar.client.auth-plugin-class-name`	Fully qualified class name of the authentication plugin.	
`spring.pulsar.client.authentication.*`	Authentication parameter(s) as a map of parameter names to parameter values.	
`spring.pulsar.client.connection-timeout-ms`	Duration to wait for a connection to a broker to be established in milliseconds.	`10000`
`spring.pulsar.client.dns-lookup-bind-address`	DNS lookup bind address.	
`spring.pulsar.client.dns-lookup-bind-port`	DNS lookup bind port.	`0`
`spring.pulsar.client.enable-busy-wait`	Enables spin-waiting on executors and IO threads in order to reduce latency during context switches.	`false`
`spring.pulsar.client.enable-transaction`	Enables transactions. To use this, start the transactionCoordinatorClient with the pulsar client.	`false`
`spring.pulsar.client.initial-backoff-interval-nanos`	Initial backoff interval in nanoseconds.	`100`
`spring.pulsar.client.keep-alive-interval-seconds`	Keep alive interval for broker-client connection in seconds.	`30`
`spring.pulsar.client.listener-name`	Listener name for lookup. Clients can use listenerName to choose one of the listeners as the service URL to create a connection to the broker. To use this, "advertisedListeners" must be enabled on the broker.	

Name	Description	Default Value
spring.pulsar.client.lookup-timeout-ms	Client lookup timeout in milliseconds.	-1
spring.pulsar.client.max-backoff-interval-nanos	Maximum backoff interval in nanoseconds.	30
spring.pulsar.client.max-concurrent-lookup-request	Number of concurrent lookup-requests allowed to send on each broker-connection to prevent overload on broker.	5000
spring.pulsar.client.max-lookup-redirects	Maximum number of times a lookup-request to a broker will be redirected.	20
spring.pulsar.client.max-lookup-request	Number of max lookup-requests allowed on each broker-connection to prevent overload on broker.	50000
spring.pulsar.client.max-number-of-rejected-request-per-connection	Maximum number of broker-rejected requests in a certain timeframe, after which the current connection is closed and a new connection is created by the client.	50
spring.pulsar.client.memory-limit-bytes	Limit of direct memory that will be allocated by the client.	0
spring.pulsar.client.num-connections-per-broker	Maximum number of connections that the client will open to a single broker.	1
spring.pulsar.client.num-io-threads	Number of threads to be used for handling connections to brokers.	1

Name	Description	Default Value
spring.pulsar.client.num-listener-threads	Number of threads to be used for message listeners. The listener thread pool is shared across all the consumers and readers that are using a "listener" model to get messages. For a given consumer, the listener will always be invoked from the same thread, to ensure ordering.	1
spring.pulsar.client.operation-timeout-ms	Client operation timeout in milliseconds.	30000
spring.pulsar.client.service-url	Pulsar cluster URL to connect to a broker.	
spring.pulsar.client.socks5-proxy-address	SOCKS5 proxy address.	
spring.pulsar.client.socks5-proxy-password	SOCKS5 proxy password.	
spring.pulsar.client.socks5-proxy-username	SOCKS5 proxy username.	
spring.pulsar.client.ssl-provider	Name of the security provider used for SSL connections.	
spring.pulsar.client.stats-interval-seconds	Interval between each stat info in seconds.	60
spring.pulsar.client.tls-allow-insecure-connection	Whether the client accepts untrusted TLS certificates from the broker.	false
spring.pulsar.client.tls-ciphers	Comma-separated list of cipher suites. This is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol. By default, all the available cipher suites are supported.	

Name	Description	Default Value
spring.pulsar.client.tls-hostname-verification-enable	Whether the hostname is validated when the proxy creates a TLS connection with brokers.	false
spring.pulsar.client.tls-protocols	Comma-separated list of SSL protocols used to generate the SSLContext. Allowed values in recent JVMs are TLS, TLSv1.3, TLSv1.2 and TLSv1.1.	
spring.pulsar.client.tls-trust-certs-file-path	Path to the trusted TLS certificate file.	
spring.pulsar.client.tls-trust-store-password	Store password for the key store file.	
spring.pulsar.client.tls-trust-store-path	Location of the trust store file.	
spring.pulsar.client.tls-trust-store-type	File format of the trust store file.	
spring.pulsar.client.use-key-store-tls	Enable KeyStore instead of PEM type configuration if TLS is enabled.	false
spring.pulsar.client.use-tcp-no-delay	Whether to use TCP no-delay flag on the connection, to disable Nagle algorithm.	true
spring.pulsar.client.use-tls	Whether to use TLS encryption on the connection.	false

Authentication

To connect to a Pulsar cluster requiring authentication, you need to set the `authPluginClassName` and any parameters required by the authentication plugin. The parameters can be set as a single JSON encoded string or as map of parameter names to parameter values.

Map

```
spring:
  pulsar:
    client:
      auth-plugin-class-name:
org.apache.pulsar.client.impl.auth.oauth2.AuthenticationOAuth2
      authentication:
        issuer-url: https://auth.server.cloud/
        private-key: file:///Users/some-key.json
        audience: urn:sn:acme:dev:my-instance
```

JSON encoded string

```
spring:
  pulsar:
    client:
      auth-plugin-class-name:
org.apache.pulsar.client.impl.auth.oauth2.AuthenticationOAuth2
      auth-params: "{\"privateKey\":\"file:///Users/some-
key.json\",\"issuerUrl\":\"https://auth.server.cloud/",
\"audience\":\"urn:sn:acme:dev:my-instance"}"
```

Using a map is the recommended approach as it is less error-prone and easier to read

The following shows how to configure each of the supported authentication mechanisms.

Click here for **Athenz**

```
spring:
  pulsar:
    client:
      auth-plugin-class-name:
org.apache.pulsar.client.impl.auth.AuthenticationAthenz
      authentication:
        tenant-domain: ...
        tenant-service: ...
        provider-domain: ...
        private-key: ...
        key-id: ...
      enable-tls: true
      tls-trust-certs-file: /path/to/cacert.pem
```

Click here for **Basic**

```
spring:
  pulsar:
    client:
      auth-plugin-class-name:
org.apache.pulsar.client.impl.auth.AuthenticationBasic
      authentication:
        user-id: ...
        password: ...
```

Click here for **OAuth2**

```
spring:
  pulsar:
    client:
      auth-plugin-class-name:
org.apache.pulsar.client.impl.auth.oauth2.AuthenticationFactoryOAuth2
      authentication:
        issuer-url: ...
        private-key: ...
        audience: ...
        scope: ...
```

Click here for **Sasl**

```
spring:
  pulsar:
    client:
      auth-plugin-class-name:
org.apache.pulsar.client.impl.auth.AuthenticationSasl
      authentication:
        sasl-jaas-client-section-name: ...
        server-type: ...
```

Click here for **Tls**

```
spring:
  pulsar:
    client:
      auth-plugin-class-name: org.apache.pulsar.client.impl.auth.AuthenticationTls
      authentication:
        tls-cert-file: /path/to/my-role.cert.pem
        tls-key-file: /path/to/my-role.key-pk8.pem
      enable-tls: true
      tls-trust-certs-file: /path/to/cacert.pem
```

Click here for **Token**

```
spring:
  pulsar:
    client:
      auth-plugin-class-name:
org.apache.pulsar.client.impl.auth.AuthenticationToken
      authentication:
        token: some-token-goes-here
```

More information on each of the schemes and their required properties can be found in the official Pulsar security documentation.

3.1.2. Pulsar Producer

On the Pulsar producer side, Spring Boot auto-configuration provides a `PulsarTemplate` for publishing records. The template implements an interface called `PulsarOperations` and provides methods to publish records through its contract.

There are two categories of these send API methods - `send` and `sendAsync`. The `send` methods are blocking calls using the synchronous sending capabilities on the Pulsar producer. They return the `MessageId` of the message that was published once the message is persisted on the broker. The `sendAsync` method calls are asynchronous calls that are non-blocking. They return a `CompletableFuture` using which you can asynchronously receive the message id once the messages are published.

Simple API

The template provides a handful of methods (prefixed with *'send'*) for simple send requests that contain only a message and/or destination topic. For more complicated send requests there is a fluent API that allows the user to configure more options (see below).

Both `send` and `sendAsync` methods have a variant that allows to publish simply with the message. When you do that, the application must provide the topic name using the property `spring.pulsar.producer.topic-name`.

Fluent API

The template provides a fluent builder to handle more complicated send requests.

Message customization

A `TypedMessageBuilderCustomizer` can be specified in order to configure the outgoing message. For example, the following code shows how to send a keyed message:

```
template.newMessage(msg)
    .withMessageCustomizer((mb) -> mb.key("foo-msg-key"))
    .send();
```

Producer customization

A `ProducerBuilderCustomizer` can be specified in order to configure the underlying Pulsar producer builder that ultimately constructs the producer used to send the outgoing message.

Use with caution as this gives full access to the producer builder and invoking some of its method's may have unintended side effects (eg. `create`).

For example, the following code shows how to disable batching and enable chunking:

```
template.newMessage(msg)
        .withProducerCustomizer((pb) ->
pb.enableChunking(true).enableBatching(false))
        .send();
```

Custom routing

You can use custom routing when publishing records to partitioned topics. Simple specify your custom `MessageRouter` implementation on the fluent builder such as:

```
template.newMessage(msg)
    .withCustomRouter(myCustomRouter)
    .send();
```

Note that, when using a `MessageRouter`, the only valid setting for `spring.pulsar.producer.message-routing-mode` is `custom`.

Schema

If you are using simple Java primitive types, then the framework auto-detects the schema for you, and you do not need to specify any schema types for publishing the data. However, if you are using any complex types such as `JSON`, `AVRO`, `PROTOBUF`, etc. then you need to set the proper schema type on the `PulsarTemplate` before invoking any send operations as shown below.

```
pulsarTemplate.setSchema(Schema.JSON(Foo.class));
```

Click here to view the available **Pulsar Producer Properties**

Name	Description	Default Value
spring.pulsar.producer.batching-enabled	Whether to automatically batch messages.	true
spring.pulsar.producer.batching-max-messages	Maximum number of messages to be batched.	1000
spring.pulsar.producer.batching-max-publish-delay-micros	Time period within which the messages sent will be batched in milliseconds.	1
spring.pulsar.producer.block-if-queue-full	Whether the "send" and "sendAsync" methods should block if the outgoing message queue is full.	false
spring.pulsar.producer.cache.expire-after-access	Time period to expire unused entries in the cache.	1m
spring.pulsar.producer.cache.initial-capacity	Initial size of cache.	50
spring.pulsar.producer.cache.maximum-size	Maximum size of cache (entries).	1000
spring.pulsar.producer.chunking-enabled	Whether to split large-size messages into multiple chunks.	false
spring.pulsar.producer.compression-type	Message compression type.	
spring.pulsar.producer.crypto-failure-action	Action the producer will take in case of encryption failure.	
spring.pulsar.producer.hashing-scheme	Message hashing scheme to choose the partition to which the message is published.	
spring.pulsar.producer.initial-subscription-name	Name of the initial subscription of the topic.	
spring.pulsar.producer.max-pending-messages	Maximum number of pending messages for the producer.	1000
spring.pulsar.producer.max-pending-messages-across-partitions	Maximum number of pending messages across all the partitions.	50000

Name	Description	Default Value
`spring.pulsar.producer.message-routing-mode`	Message routing mode for a partitioned producer.	
`spring.pulsar.producer.producer-access-mode`	Type of access to the topic the producer requires.	
`spring.pulsar.producer.producer-name`	Name for the producer. If not assigned, a unique name is generated.	
`spring.pulsar.producer.send-timeout-ms`	Time before a message has to be acknowledged by the broker in milliseconds.	`30000`
`spring.pulsar.producer.topic-name`	Topic the producer will publish to.	

3.1.3. Pulsar Producer Factory

The `PulsarTemplate` relies on a `PulsarProducerFactory` for actually creating the underlying producer. Spring Boot auto-configuration also provides this producer factory. Additionally, you can configure the factory by specifying any of the available producer-centric application properties listed above.

3.1.4. Pulsar Producer Caching

Each underlying Pulsar producer consumes resources. In order to improve performance and avoid continual creation of producers, the producer factory caches the producers that it creates. They are cached in an LRU fashion and evicted when they have not been used within a configured time period. The cache key is composed of just enough information to ensure that callers are returned the same producer on subsequent creation requests.

Additionally, you can configure the cache settings by specifying any of the `spring.producer.cache` prefixed application properties listed above.

3.1.5. Pulsar Listener

When it comes to Pulsar consumer, we recommend the end user applications to make use of the `PulsarListener` annotation. In order to use `PulsarListener`, you need to use the `@EnablePulsar` annotation. When using the Spring Boot support, it automatically enables this annotation and configures all the components necessary for `PulsarListener` such as the message listener infrastructure which is responsible for creating the Pulsar consumer. `PulsarMessageListenerContainer` uses a `PulsarConsumerFactory` in order to create and manage the Pulsar consumer. This consumer factory is also auto-configured through Spring Boot.

Click here to view the available **Pulsar Consumer Properties**

Name	Description	Default Value
`spring.pulsar.consumer.ack-timeout-millis`	Timeout for unacked messages to be redelivered.	`0`
`spring.pulsar.consumer.acknowledgements-group-time-micros`	Time to group acknowledgements before sending them to the broker in microseconds.	`100`
`spring.pulsar.consumer.auto-ack-oldest-chunked-message-on-queue-full`	Whether to automatically drop outstanding un-acked messages if the queue is full.	`true`
`spring.pulsar.consumer.auto-update-partitions`	Whether the consumer auto-subscribes for partition increase. This is only for partitioned consumers.	`true`
`spring.pulsar.consumer.consumer-name`	Consumer name to identify a particular consumer from the topic stats.	
`spring.pulsar.consumer.crypto-failure-action`	Action the consumer will take in case of decryption failure.	
`spring.pulsar.consumer.expire-time-of-incomplete-chunked-message-millis`	Time to expire incomplete chunks if the consumer won't be able to receive all chunks before in milliseconds.	`60000`
`spring.pulsar.consumer.max-pending-chunked-message`	Maximum number of chunked messages to be kept in memory.	`10`
`spring.pulsar.consumer.max-total-receiver-queue-size-across-partitions`	Maximum number of messages that a consumer can be pushed at once from a broker across all partitions.	`50000`
`spring.pulsar.consumer.negative-ack-redelivery-delay-micros`	Delay before re-delivering messages that have failed to be processed in microseconds.	`1`

Name	Description	Default Value
spring.pulsar.consumer.pattern-auto-discovery-period	Auto-discovery period for topics when topic pattern is used in minutes.	1
spring.pulsar.consumer.priority-level	Priority level for shared subscription consumers.	0
spring.pulsar.consumer.properties	Map of properties to add to the consumer.	
spring.pulsar.consumer.read-compacted	Whether to read messages from the compacted topic rather than the full message backlog.	false
spring.pulsar.consumer.receiver-queue-size	Number of messages that can be accumulated before the consumer calls "receive".	1000
spring.pulsar.consumer.regex-subscription-mode	Determines which topics the consumer should be subscribed to when using pattern subscriptions.	
spring.pulsar.consumer.replicate-subscription-state	Whether to replicate subscription state.	false
spring.pulsar.consumer.subscription-initial-position	Position where to initialize a newly created subscription.	
spring.pulsar.consumer.subscription-name	Subscription name for the consumer.	
spring.pulsar.consumer.subscription-type	Subscription type to be used when subscribing to a topic.	
spring.pulsar.consumer.tick-duration-millis	Precision for the ack timeout messages tracker in milliseconds.	1000
spring.pulsar.consumer.topics	Comma-separated list of topics the consumer subscribes to.	
spring.pulsar.consumer.topics-pattern	Pattern for topics the consumer subscribes to.	
spring.pulsar.listener.ack-mode	AckMode for acknowledgements. Allowed values are RECORD, BATCH, MANUAL.	

Name	Description	Default Value
spring.pulsar.listener.batch-timeout-millis	Number of milliseconds to wait for enough message to fill a batch request before timing out.	100
spring.pulsar.listener.max-num-bytes	Max number of bytes in a single batch request.	0
spring.pulsar.listener.max-num-messages	Max number of messages in a single batch request.	-1
spring.pulsar.listener.schema-type	SchemaType of the consumed messages.	

Let us revisit the `PulsarListener` code snippet we saw in the quick-tour section.

```
@PulsarListener(subscriptionName = "hello-pulsar-subscription", topics = "hello-
pulsar")
public void listen(String message) {
    System.out.println("Message Received: " + message);
}
```

This can even be further simplified as below.

```
@PulsarListener
public void listen(String message) {
    System.out.println("Message Received: " + message);
}
```

In this most basic form, you must provide the following two properties with their corresponding values.

```
spring.pulsar.consumer:
  topic-names: hello-pulsar
  subscription-name: hello-pulsar-subscription
```

In the `PulsarListener` method above, we receive the data as `String`, but we don't specify any schema types. Internally, the framework relies on Pulsar's schema mechanism to convert the data to the required type. The framework detects that you are expecting the `String` type and then infers the schema type based on that information. Then it provides that schema to the consumer. For all the primitive types in Java, the framework does this inference. For any complex types, such as JSON, AVRO etc. the framework cannot do this inference and the user needs to provide the schema type

on the annotation using the `schemaType` property.

Here is another `PulsarListener` method, that takes an `Integer`.

```
@PulsarListener(subscriptionName = "my-subscription-1", topics = "my-topic-1")
public void listen2(Integer message) {
   System.out.println(message);
}
```

The following `PulsarListener` method shows how we can consume complex types from a topic

```
@PulsarListener(subscriptionName = "my-subscription-2", topics = "my-topic-2",
schemaType = SchemaType.JSON)
public void listen3(Foo message) {
    System.out.println(message);
}
```

Note the addition of a `schemaType` property on `PulsarListener`. That is because the library is not capable of inferring the schema type from the provided type `Foo`, we must tell the framework what schema to use.

Here is an example of using `PulsarListener` to consume records in batches.

```
@PulsarListener(subscriptionName = "hello-batch-subscription", topics = "hello-
batch", schemaType = SchemaType.JSON, batch = true)
public void listen4(List<Foo> messages) {
    System.out.println("records received :" + messages.size());
    messages.forEach((message) -> System.out.println("record : " + message));
}
```

Note that in this example, we are receiving the records as a collection (`List`) of objects. In addition, in order to enable batch consumption at the `PulsarListener` level, you need to set the `batch` property on the annotation to `true`.

Based on the actual type that the `List` holds, the framework tries to infer the schema to use. If the `List` contains a complex type, then the `schemaType` still needs to be provided on `PulsarListener`.

The following also should work in which we use the `Message` envelope provided by the Pulsar Java client.

```
@PulsarListener(subscriptionName = "hello-batch-subscription", topics = "hello-
batch", schemaType = SchemaType.JSON, batch = true)
public void listen4(List<Message<Foo>> messages) {
    System.out.println("records received :" + messages.size());
    messages.forEach((message) -> System.out.println("record : " + message));
}
```

When using `PulsarListener`, you can provide Pulsar consumer properties directly on the annotation itself. This is convenient, if you do not want to use the Boot configuration properties mentioned above or have multiple `PulsarListener` methods.

Here is an example of using Pulsar consumer properties directly on `PulsarListener`.

```
@PulsarListener(properties = { "subscriptionName=subscription-1", "topicNames=foo-
1", "receiverQueueSize=5000" })
void listen(String message) {
}
```

Note that the properties used are direct Pulsar consumer properties.

3.1.6. Pulsar Message Listener Container

Now that we saw the basic interactions on the consumer side through `PulsarListener`, let us now dive into the inner workings of how `PulsarListener` interacts with the underlying Pulsar consumer. Keep in mind that, for end-user applications, in most scenarios, we recommend using `PulsarListener` annotation directly for consuming from a Pulsar topic when using Spring for Apache Pulsar, as that model covers a broad set of application use cases. However, it is important to understand how `PulsarListener` works internally and this section will go through those details.

As briefly mentioned above, the message listener container is at the heart of message consumption when using Spring for Apache Pulsar. `PulsarListener` uses the message listener container infrastructure behind the scenes to create and manage the Pulsar consumer. Spring for Apache Pulsar provides the contract for this message listener container through `PulsarMessageListenerContainer`. The default implementation for this message listener container is provided through `DefaultPulsarMessageListenerContainer`. As its name indicates, `PulsarMessageListenerContainer` contains the message listener. The container creates the Pulsar consumer and then runs a separate thread to receive and handle the data. The data is handled by the provided message listener implementation.

The message listener container consumes the data in batch using the consumer's `batchReceive` method. Once data is received, it is handed over to the selected message listener implementation.

The following message listener types are available when using Spring for Apache Pulsar.

- PulsarRecordMessageListener
- PulsarAcknowledgingMessageListener
- PulsarBatchMessageListener
- PulsarBatchAcknowledgingMessageListener

We will see the details about these various message listeners in the sections below.

Before doing so however, lets take a closer look at the container itself

DefaultPulsarMessageListenerContainer

This is a single consumer based message listener container. Here is it's constructor.

```
public DefaultPulsarMessageListenerContainer(PulsarConsumerFactory<? super T>
pulsarConsumerFactory,
            PulsarContainerProperties pulsarContainerProperties)
}
```

It receives a `PulsarConsumerFactory` that it uses to create the consumer and a `PulsarContainerProperties` object that contains information about the container properties. `PulsarContainerProperties` has the following constructors.

```
public PulsarContainerProperties(String... topics)

public PulsarContainerProperties(Pattern topicPattern)
```

You can provide the topic information through `PulsarContainerProperties` or as a consumer property that is provided to the consumer factory. Here is an example of using the `DefaultPulsarMessageListenerContainer`.

```
Map<String, Object> config = new HashMap<>();
config.put("topics", "my-topic");
PulsarConsumerFactory<String> pulsarConsumerFactorY =
DefaultPulsarConsumerFactory<>(pulsarClient, config);

PulsarContainerProperties pulsarContainerProperties = new
PulsarContainerProperties();

pulsarContainerProperties.setMessageListener((PulsarRecordMessageListener<?>)
(consumer, msg) -> {
        });

DefaultPulsarMessageListenerContainer<String> pulsarListenerContainer = new
DefaultPulsarMessageListenerContainer(pulsarConsumerFacotyr,
        pulsarContainerProperties);

return pulsarListenerContainer;
```

`DefaultPulsarMessageListenerContainer` only creates a single consumer. If you want to have multiple consumers managed through multiple threads, you need to use `ConcurrentPulsarMessageListenerContainer`.

ConcurrentPulsarMessageListenerContainer

`ConcurrentPulsarMessageListenerContainer` has the following constructor.

```
public ConcurrentPulsarMessageListenerContainer(PulsarConsumerFactory<? super T>
pulsarConsumerFactory,
    PulsarContainerProperties pulsarContainerProperties)
```

`ConcurrentPulsarMessageListenerContainer` allows to specify a `concurrency` property through a setter. Concurrency of more than `1` is only allowed on non-exclusive subscriptions (`failover`, `shared` and `key-shared`). You can only have the default `1` for concurrency when you have an exclusive subscription mode.

Here is an example of enabling `concurrency` through the `PulsarListener` annotation for a `failover` subscription.

```
@PulsarListener(topics = "my-topic", subscriptionName = "subscription-1",
                subscriptionType = SubscriptionType.Failover, concurrency = "3")
void listen(String message, Consumer<String> consumer) {
    ...
    System.out.println("Current Thread: " + Thread.currentThread().getName());
    System.out.println("Current Consumer: " + consumer.getConsumerName());
}
```

In the above listener, it is assumed that the topic `my-topic` has 3 partitions. If it is a non-partitioned topic, then having concurrency set to `3`, will not do anything, you will simply get two idle consumers in addition to the main active one. If the topic has more than 3 partitions, then messages will be load-balanced across the consumers that the container creates. If you run this `PulsarListener`, you will see that messages from different partitions will be consumed through different consumers as implied by the thread name and consumer names printouts in the example code above.

Note: When using the `Failover` subscription this way on partitioned topics, Pulsar guarantees message ordering.

Here is another example of `PulsarListener`, but with `Shared` subscription and `concurrency` enabled.

```
@PulsarListener(topics = "my-topic", subscriptionName = "subscription-1",
                subscriptionType = SubscriptionType.Shared, concurrency = "5")
void listen(String message) {
    ...
}
```

In the example above, the `PulsarListener` creates 5 different consumers (once again, we are assuming that the topic has 5 partitions).

Keep in mind that, in this version, there is no message ordering as `Shared` subscriptions do not guarantee any message ordering in Pulsar

If you need message ordering and still want a shared subscription types, then you need to use the `Key_Shared` subscription type.

3.1.7. Consuming the Records

In this section, we are going to see how the message listener container enables both single record and batch based message consumption.

3.1.8. Single Record Consumption

Let us re-visit our basic `PulsarListener` for the sake of this discussion.

```
@PulsarListener(subscriptionName = "hello-pulsar-subscription", topics = "hello-
pulsar")
public void listen(String message) {
    System.out.println("Message Received: " + message);
}
```

With this `PulsarListener` method, what we are essentially doing is that asking Spring for Apache Pulsar to invoke the listener method with a single record each time. We mentioned that the message listener container consumes the data in batches using the `batchReceive` method on the consumer. The framework detects that the `PulsarListener` in this case receives a single record which means that on each invocation of the method it needs a singe record. Although the records are consumed by the message listener container in batches, it iterates through the received batch and then invoke the listener method through an adapter for `PulsarRecordMessageListener`. As you can see in the previous section, `PulsarRecordMessageListener` simply extends from the `MessageListener` provided by the Pulsar Java client and it supports the basic `received` method.

3.1.9. Batch Consumption

Here is the `PulsarListener` example of consuming records in batches.

```
@PulsarListener(subscriptionName = "hello-batch-subscription", topics = "hello-
batch", schemaType = SchemaType.JSON, batch = true)
public void listen4(List<Foo> messages) {
    System.out.println("records received :" + messages.size());
    messages.forEach((message) -> System.out.println("record : " + message));
}
```

When using this type of `PulsarListener`, the framework detects that you are in batch mode. Since it is already received the data in batches using the Consumer's `batchReceive` method, it simply hands off the entire batch to the listener method through an adapter for `PulsarBatchMessageListener`.

3.1.10. Message Acknowledgment

When using Spring for Apache Pulsar, the message acknowledgment is handled by the framework unless opted out by the application. In this section, we go through the details of how the framework takes care of message acknowledgment.

3.1.11. Message ACK modes

Spring for Apache Pulsar provides the following modes for acknowledging messages

```
BATCH,

RECORD,

MANUAL;
```

`BATCH` acknowledgment mode is the default, but you can change it on the message listener container. In the following sections, we will see how acknowledgment works when using both single and batch versions of `PulsarListener` and how they translate to the backing message listener container (and of course ultimately to the Pulsar consumer).

3.1.12. Automatic Message Ack in Single Record Mode

Let us revisit our basic single message based `PulsarListener`.

```
@PulsarListener(subscriptionName = "hello-pulsar-subscription", topics = "hello-
pulsar")
public void listen(String message) {
    System.out.println("Message Received: " + message);
}
```

It is natural to wonder, how acknowledgment works when using `PulsarListener`, espcially if you are familiar with Pulsar consumer directly. The answer comes down to the message listener container as that is the central place in Spring for Apache Pulsar which coordinates all the consumer related activities.

Assuming you are not overriding the default behavior, this is what happens behind the scenes when using the above `PulsarListener`.

1. First, the Listener container receives messages as batch from the Pulsar consumer.
2. The received messages are handed down to `PulsarListener` one message at a time
3. When all the records are handed down to the listener method and successfully processed, the container will acknowledge all the messages from the original bach receive.

This is the normal flow. If any record from the original batch received, throws an exception, Spring for Apache Pulsar will track them separately. When all the records from the batch are processed, then Spring for Apache Pulsar will acknowledge all the succesful messages and negatively acknowledge (nack) all the failed messages. In other words, when consuming single records using `PulsarRecordMessageListener` and the default ack mode of `BATCH` is used, the framework waits for all the record received from the `batchReceive` call to process successfully and then call the `acknowledge` method on the Pulsar Consumer. If any particular record throws an exception when invoking the handler method, Spring for Apache Pulsar tracks those records and separately call `negativeAcknowledge` on those records after the entire batch is processed.

If the application wants the acknowledgment or negative acknowledgment to occur per record,

then the RECORD ack mode can be enabled. In that case, after handling each record, the message is acknowledged if no error and negatively acknowledged if there was an error. Here is an example of enabling RECORD ack mode on Pulsar Listener.

```
@PulsarListener(subscriptionName = "hello-pulsar-subscription", topics = "hello-
pulsar", ackMode = AckMode.RECORD)
public void listen(String message) {
    System.out.println("Message Received: " + message);
}
```

You can also set the listener property, spring.pulsar.listner.ack-mode to set the ack mode application wide. When doing this, you do not need to set this on the PulsarListener annotation. In that case, all the PulsarListener methods in the application acquires that property.

3.1.13. Manual Message Ack in Single Record Mode

There are situations in which you might not want the framework to do any acknowledgments, but rather do that directly from the application itself. Spring for Apache Pulsar provides a couple of ways to enable manual message acknowledgments. Let us look at a few examples.

```
@PulsarListener(subscriptionName = "hello-pulsar-subscription", topics = "hello-
pulsar", ackMode = AckMode.MANUAL)
public void listen(Message<String> message, Acknowledgment acknowledgment) {
    System.out.println("Message Received: " + message.getValue());
    acknowledgment.acknowledge();
}
```

Few things merit explanation here - First we are enabling manual ack mode by setting ackMode on PulsarListener. When enabling manual ack mode, Spring for Apache Pulsar allows the application to inject an Acknowledgment object as you can see in the above PulsarListener method. The framework achieves this by selecting a compatible message listener container - PulsarAcknowledgingMessageListener for single record based consumption which gives you access to an Acknowledgment object.

The Acknowledgment object provides the following API methods.

```
void acknowledge();

void acknowledge(MessageId messageId);

void acknowledge(List<MessageId> messageIds);

void nack();

void nack(MessageId messageId);
```

You can inject this `Acknowledgment` object to your `PulsarListener` while using `MANUAL` ack mode and then call one of the corresponding methods above.

In the above `PulsarListener` example, we are calling a parameter-less `acknowledge` method. This is because the framework knows which `Message` it is operating under currently. When calling `acknowledge()`, you do not need to receive the payload with the `Message` enveloper`, but rather simply using the target type - `String` in this example. You can also call a different variant of `acknowledge` by providing the message id - `acknowledge.acknowledge(message.getMessageId());` When using `acknowledge(messageId)`, you must receive the payload using the `Message<?>` envelope.

Similar to what is possible for acknowledging, the `Acknowledgment` API also provides options for negatively acknowledging - see the nack methods above.

You can also call `acknowledge` directly on the Pulsar consumer as below.

```
@PulsarListener(subscriptionName = "hello-pulsar-subscription", topics = "hello-
pulsar", ackMode = AckMode.MANUAL)
public void listen(Message<String> message, Consumer<String> consumer) {
    System.out.println("Message Received: " + message.getValue());
    try {
        consumer.acknowledge(message);
    }
    catch (Exception e) {
        ....
    }
}
```

As you can see, when calling `acknowledge` directly on the underlying consumer, then you need to do error handling by yourself. Using the `Acknowledgment` does not require that as the framework can do that for you. Therefore, it is recommended to use the `Acknowledgment` object approach when using manual acknowledgment.

When using manual acknowledgment, it is important to understand that the framework completely stay from any acknowledgment at all. Hence, it is extremely important for the end-users to think through the right acknowledgment strategies when designing applications.

3.1.14. Message Ack in Batch Consumption

When records are consumed in batches (See the section above), then if the default ack mode of `BATCH` is used, then when the entire batch is processed successfully, it will be acknowledged. If any records throw an exception, then the entire batch is negatively acknowledged. Note that this may not be the same batch that was batched on the producer side, rather this is the batch that returned from calling `batchReceive` on the consumer

Let us look at the following batch listener:

```
@PulsarListener(subscriptionName = "hello-pulsar-subscription", topics = "hello-
pulsar", batch = true)
public void batchListen(List<Foo> messages) {
    for (Foo foo : messages) {
        ...
    }
}
```

When all the messages in the incoming collection (`messages` in this example) are processed, the framework will acknowledge all of them.

When consuming in batch mode, `RECORD` is not an allowed ack mode. This might cause an issue as application may not want the entire batch to be re-delivered again. For such situations, you need to use the `MANUAL` acknowledgement mode.

3.1.15. Manual Messge Acknowledgment in Batch Consumption

As seen in the previous section, when `MANUAL` ack mode is set on the message listener container, then the framework will not do any acknowledgment - positive or negative. It is entirely up to the application to take care of such concerns. When `MANUAL` ack mode is set, Spring for Apache Pulsar selects a compatible message listener container - `PulsarBatchAcknowledgingMessageListener` for batch consumption which gives you access to an `Acknowledgment` object. Once again, the following are the methods availble in the `Acknowledgment` API.

```
void acknowledge();

void acknowledge(MessageId messageId);

void acknowledge(List<MessageId> messageIds);

void nack();

void nack(MessageId messageId);
```

You can inject this `Acknowledgment` object to your `PulsarListener` while using `MANUAL` ack mode. Here

is a basic example for a batch based listener.

```
@PulsarListener(subscriptionName = "hello-pulsar-subscription", topics = "hello-
pulsar")
public void listen(List<Message<String>> messgaes, Acknowlegement acknowledgment)
{
    for (Message<String> message : messages) {
        try {
            ...
            acknowledgment.acknowledge(message.getMessageId());
        }
        catch (Exception e) {
            acknowledgment.nack(message.getMessageId());
        }
    }
}
```

When using a batch listener, the message listener container cannot know which record it is currently operating upon. Therefore, in order to manually acknowledge, you need to use one of the overloaded `acknowledge` method that takes a `MessageId` or a `List<MessageId>`. You can also negatively acknowledge with the `MessageId` for the batch listener.

3.1.16. Partitioned topics - Publishing and Consuming.

In the sample below, we are publishing to a topic called `hello-pulsar-partitioned`. It is a topic that is partitioned and for this sample we assume that the topic is already created with three partitions.

```
@SpringBootApplication
public class PulsarBootPartitioned {

    public static void main(String[] args) {
        SpringApplication.run(PulsarBootPartitioned.class, "--
spring.pulsar.producer.message-routing-mode=CustomPartition");
    }

    @Bean
    public ApplicationRunner runner(PulsarTemplate<String> pulsarTemplate) {
        pulsarTemplate.setDefaultTopicName("hello-pulsar-partitioned");
        return args -> {
            for (int i = 0; i < 10; i++) {
                pulsarTemplate.sendAsync("hello john doe 0 ", new FooRouter());
                pulsarTemplate.sendAsync("hello alice doe 1", new BarRouter());
                pulsarTemplate.sendAsync("hello buzz doe 2", new BuzzRouter());
            }
        };
    }

    @PulsarListener(subscriptionName = "hello-pulsar-partitioned-subscription",
topics = "hello-pulsar-partitioned")
    public void listen(String message) {
        System.out.println("Message Received: " + message);
    }

    static class FooRouter implements MessageRouter {

        @Override
        public int choosePartition(Message<?> msg, TopicMetadata metadata) {
            return 0;
        }
    }

    static class BarRouter implements MessageRouter {

        @Override
        public int choosePartition(Message<?> msg, TopicMetadata metadata) {
            return 1;
        }
    }

    static class BuzzRouter implements MessageRouter {

        @Override
        public int choosePartition(Message<?> msg, TopicMetadata metadata) {
            return 2;
        }
    }
```

```
}
```

A few things require explanation in the application above. We are publishing to a partitioned topic and we would like to publish some data segment to a specific partition. If you leave it to Pulsar's default, it follows a round-robin mode of partition assignments, and we would like to override that. In order to do that, we are providing a message router object with the send method. Look at the three message routers implemented. `FooRouter` always sends data to partition `0`, `BarRouter` to partition `1` and `BuzzRouter` to partition `2`. Also note that, we are now using the `sendAsync` method of `PulsarTemplate` that returns a `CompletableFuture`. When running the application, we also need to set the `messageRoutingMode` on the producer to `CustomPartition` (`spring.pulsar.producer.message-routing-mode`).

On the consumer side, we are using a `PulsarListener` with the exclusive subscription type. This means that data from all the partitions will end up in the same consumer and there is no ordering guarantee.

What can we do if we want each partition to be consumed by a single distinct consumer? We can switch to the `failover` subscription mode and add three separate consumers.

Here is an example.

```
@PulsarListener(subscriptionName = "hello-pulsar-partitioned-subscription", topics
= "hello-pulsar-partitioned", subscriptionType = SubscriptionType.Failover)
public void listen1(String foo) {
    System.out.println("Message Received 1: " + foo);
}

@PulsarListener(subscriptionName = "hello-pulsar-partitioned-subscription", topics
= "hello-pulsar-partitioned", subscriptionType = SubscriptionType.Failover)
public void listen2(String foo) {
    System.out.println("Message Received 2: " + foo);
}

@PulsarListener(subscriptionName = "hello-pulsar-partitioned-subscription",
topics = "hello-pulsar-partitioned", subscriptionType = SubscriptionType.Failover)
public void listen3(String foo) {
    System.out.println("Message Received 3: " + foo);
}
```

When following this approach, you can see that a single partition always gets consumed by a dedicated consumer.

In the similar vein, if you want to use Pulsar's shared consumer type, you can use the subscription type `shared`. Keep in mind though, that when using the `shared` mode, you lose any ordering guarantees as a single consumer may receive messages from all the partitions before another

consumer gets a chance.

Here is an example.

```
@PulsarListener(subscriptionName = "hello-pulsar-shared-subscription", topics =
"hello-pulsar-partitioned", subscriptionType = SubscriptionType.Shared)
public void listen1(String foo) {
    System.out.println("Message Received 1: " + foo);
}

@PulsarListener(subscriptionName = "hello-pulsar-shared-subscription", topics =
"hello-pulsar-partitioned", subscriptionType = SubscriptionType.Shared)
public void listen2(String foo) {
    System.out.println("Message Received 2: " + foo);
}
```

3.1.17. Accessing the Pulsar Message Object

In your `PulsarListener` method, you can receive the record directly as a Pulsar Message instead of the actual payload type. Here is an example.

```
@PulsarListener(subscriptionName = "hello-pulsar-subscription", topics = "hello-
pulsar")
public void listen(org.apache.pulsar.client.api.Message<String> message) {
    System.out.println("Data Received: " + message.getValue());
}
```

or in batch receiver:

```
@PulsarListener(subscriptionName = "batch-subscription", topics = "hello-pulsar",
batch = "true")
public void listen(List<org.apache.pulsar.client.api.Message<String>> messages) {
    // Iterate on the messages
}
```

3.1.18. Accessing the Pulsar Consumer Object

Sometimes, it is necessary to gain direct access to the Pulsar Consumer object. Here is how you may do so.

```
@PulsarListener(subscriptionName = "hello-pulsar-subscription", topics = "hello-
pulsar")
public void listen(String message, org.apache.pulsar.client.api.Consumer<String>
consumer) {
    System.out.println("Message Received: " + message);
    ConsumerStats stats = consumer.getStats();
    ...
}
```

When accessing the `Consumer` object this way, make sure NOT to invoke any operations that would change the Consumer's cursor position by invoking any receive methods. All such operations must be done by the container.

3.1.19. Specify schema information

As indicated above, for normal Java types (the primitive ones), Spring Pulsar framework can infer the proper Schema to use on the `PulsarListener`. However, for more complex types such as JSON or AVRO, you need to specify the schema type on the annotation. Here is how you provide that.

```
@PulsarListener(subscriptionName = "json-subscription", topics = "hello-pulsar-
json", schemaType = SchemaType.JSON)
public void listen(Foo foo) {
    System.out.println("Message received: " + foo);
}
```

On the producer side also, for the Java primitive types, the framework can infer the Schema, but for any other types, you need to set them on the `PulsarTemplate` as shown below.

```
template.setSchema(JSONSchema.of(Foo.class));
```

Complex Schema types that are currently supported are JSON, AVRO, PROTOBUF, and KEY_VALUE. For KEY_VALUE schemata, only INLINE encoding is supported.

3.1.20. Message Redelivery and Error Handling

Now that we have seen both `PulsarListener` and the message listener container infrastructure, and its various functions, let us now try to understand message redelivery and error handling. Apache Pulsar provides various native strategies for message redelivery and error handling, and we are going to take a look at them first and see how we can leverage them through Spring for Apache Pulsar.

Specifying Acknowledgment Timeout for Message Redelivery

By default, Pulsar consumers will not redeliver messages unless the consumer crashes, but you can change this behavior by setting an ack timeout on the Pulsar consumer. When using Spring for Apache Pulsar, we can enable this property by setting the Boot property `spring.pulsar.consumer.ack-timeout-millis`. If this property has a value above zero, then if Pulsar consumer does not acknowledge a message within that timeout period, then the message will be redelivered.

You can also specify this property directly as a Pulsar consumer property on the `PulsarListener` itself as shown below:

```
@PulsarListener(subscriptionName = "subscription-1", topics = "topic-1"
                properties = {"ackTimeoutMillis=60000"})
public void listen(String s) {
    ...
}
```

When specifying `ackTimeoutMillis` as seen in the above `PulsarListener` method, then if the consumer does not send an acknowledgement within 60 seconds, the message will be redelivered by Pulsar to the consumer.

If you want to specify some advanced backoff options for ack timeout with different delays, then you can do the following:

```
@EnablePulsar
@Configuration
class AckTimeoutRedeliveryConfig {

    @PulsarListener(subscriptionName =
"withAckTimeoutRedeliveryBackoffSubscription",
            topics = "withAckTimeoutRedeliveryBackoff-test-topic",
            ackTimeoutRedeliveryBackoff = "ackTimeoutRedeliveryBackoff",
            properties = { "ackTimeoutMillis=60000" })
    void listen(String msg) {
        // some long-running process that may cause an ack timeout
    }

    @Bean
    RedeliveryBackoff ackTimeoutRedeliveryBackoff() {
        return
MultiplierRedeliveryBackoff.builder().minDelayMs(1000).maxDelayMs(10 *
1000).multiplier(2)
                .build();
    }

}
```

In the example above, we are specifying a bean for Pulsar's `RedeliveryBackoff` with a minimum delay of 1 second and a maximum delay of 10 seconds with a backoff multiplier of 2. After the initial ack timeout occurs, then the message redeliveries will be controlled through this backoff bean. We provide the backoff bean to the `PulsarListener` annotation by setting the `ackTimeoutRedeliveryBackoff` property to the actual bean name - `ackTimeoutRedeliveryBackoff` in this case.

Specifying Negative Acknowledgment Redelivery

When acknowledging negatively, Pulsar consumer allows you to specify how the application want the message to be re-delivered. The default is to redeliver the message in 1 minute, but you can change it by providing `spring.pulsar.consumer.negative-ack-redelivery-delay-micros`. You can also set it as a consumer property directly on `PulsarListener` as shown below:

```
@PulsarListener(subscriptionName = "subscription-1", topics = "topic-1"
                properties = {"negativeAckRedeliveryDelayMicros=10000"})
public void listen(String s) {
    ...
}
```

Here also, you can specify different delays and backoff mechanisms with a multiplier by providing a `RedeliveryBackoff` bean and provide the bean name as the `negativeAckRedeliveryBackoff` property

on the PulsarProducer. Here is an example:

```
@EnablePulsar
@Configuration
class NegativeAckRedeliveryConfig {

    @PulsarListener(subscriptionName = "withNegRedeliveryBackoffSubscription",
            topics = "withNegRedeliveryBackoff-test-topic",
negativeAckRedeliveryBackoff = "redeliveryBackoff",
            subscriptionType = SubscriptionType.Shared)
    void listen(String msg) {
        throw new RuntimeException("fail " + msg);
    }

    @Bean
    RedeliveryBackoff redeliveryBackoff() {
        return
MultiplierRedeliveryBackoff.builder().minDelayMs(1000).maxDelayMs(10 *
1000).multiplier(2)
                .build();
    }

}
```

Using Dead Letter Topic from Apache Pulsar for Message Redelivery and Error Handling

Apache Pulsar allows applications to use a dead letter topic on consumers with a `Shared` subscription type. For subscription types `Exclusive` and `Failover`, this feature is not available. The basic idea is that if a message is retried for a certain number of times, maybe due to an ack timeout or nack redelivery, and once the number of retries are exhausted, then the message can be sent to a special topic called DLQ. Let us see some details around this feature in action by inspecting some code snippets.

```
@EnablePulsar
@Configuration
class DeadLetterPolicyConfig {

    @PulsarListener(id = "deadLetterPolicyListener", subscriptionName =
"deadLetterPolicySubscription",
            topics = "topic-with-dlp", deadLetterPolicy = "deadLetterPolicy",
            subscriptionType = SubscriptionType.Shared, properties = {
"ackTimeoutMillis=1" })
    void listen(String msg) {
        throw new RuntimeException("fail " + msg);
    }

    @PulsarListener(id = "dlqListener", topics = "my-dlq-topic")
    void listenDlq(String msg) {
        System.out.println("From DLQ: " + msg);
    }

    @Bean
    DeadLetterPolicy deadLetterPolicy() {
        return
DeadLetterPolicy.builder().maxRedeliverCount(10).deadLetterTopic("my-dlq-
topic").build();
    }

}
```

Let us go through some details. First, we have a special bean for `DeadLetterPolicy` and it's named as `deadLetterPolicy` (it acn be any name as you wish). This bean specifies a number of things, such as the max delivery - 10 in this case, and the name of the dead letter topic - `my-dlq-topic`. If you don't specify a DLQ topic name, then it defaults to `<topicname>-<subscriptionname>-DLQ` in Pulsar. Next, we provide this bean name to `PulsarListener` using the property `deadLetterPolicy`. Note that the `PulsarListener` has a subscription type of `Shared`, as the DLQ feature only works with shared subscriptions. This code is primarily for demonstration purposes, so we provide an `ackTimeoutMillis` value of 1 millisecond. The idea is that the code throws the exception and if Pulsar does not receive an ack within 1 millisecond, it does a retry. If that cycle continues for 10 times, (as that is our max redelivery count in the `DeadLetterPolicy`), then Pulsar consumer publishes the messages to the DQL topic. We have another `PulsarListener` that is listening on the DLQ topic to receive data as it is published to the DLQ topic.

Special note on DLQ topics when using partitioned topics: If the main topic is partitioned, then behind the scenes, each partition is treated as a separate topic by Pulsar. Pulsar appends `partition-<n>` where `n` stands for the partition number to the main topic name. The problem is that, if you do not specify a DLQ topic (as opposed to what we did above), then Pulsar will publish to a default topic name that has this `partition-<n>` info in it - for ex: `topic-with-dlp-partition-0-deadLetterPolicySubscription-DLQ`. The easy way to solve this is to provide a DLQ topic name always.

Native Error Handling in Spring for Apache Pulsar

As we have noted above, the DLQ feature in Apache Pulsar only works for shared subscriptions. What does an application do if they need to use some similar feature for non-shared subscriptions? The main reason why Pulsar does not support DLQ on exclusive and failover subscriptions, is because those subscription types are order-guaranteed. By allowing redeliveries, DLQ etc. it effectively receives messages in out-of-order. But, what if some applications are okay with that, but more importantly needs this DLQ feature for non-shared subscriptions? For that, Spring for Apache Pulsar provides a `PulsarConsumerErrorHandler` which can be used across any subscription types in Pulsar - `Exclusive`, `Failover`, `Shared`, `Key_Shared`.

When using `PulsarConsumerErrorHandler` from Spring for Apache Pulsar, make sure not to set the ack timeout properties on the listener.

Let us see some details by examining a few code snippets.

```
@EnablePulsar
@Configuration
class PulsarConsumerErrorHandlerConfig {

    @Bean
    PulsarConsumerErrorHandler<String> pulsarConsumerErrorHandler(
            PulsarTemplate<String> pulsarTemplate) {
        return new DefaultPulsarConsumerErrorHandler<>(
                new PulsarDeadLetterPublishingRecoverer<>(pulsarTemplate, (c, m)
-> "my-foo-dlt"), new FixedBackOff(100, 10));
    }

    @PulsarListener(id = "pulsarConsumerErrorHandler-id", subscriptionName =
"pulsatConsumerErrorHandler-subscription",
                    topics = "pulsarConsumerErrorHandler-topic",
                    pulsarConsumerErrorHandler = "pulsarConsumerErrorHandler")
    void listen(String msg) {
        throw new RuntimeException("fail " + msg);
    }

    @PulsarListener(id = "pceh-dltListener", topics = "my-foo-dlt")
    void listenDlt(String msg) {
        System.out.println("From DLT: " + msg);
    }

}
```

Let us take a look at the `pulsarConsumerErrorHandler` bean provided. This creates a bean of type `PulsarConsumerErrorHandler` and uses the default implementation provided out of the box by Spring for Apache Pulsar - `DefaultPulsarConsumerErrorHandler`. `DefaultPulsarConsumerErrorHandler` has a constructor that takes a `PulsarMessageRecovererFactory` and a `org.springframework.util.backoff.Backoff`. `PulsarMessageRecovererFactory` is a functional interface

with the following API:

```
@FunctionalInterface
public interface PulsarMessageRecovererFactory<T> {

    /**
     * Provides a message recoverer {@link PulsarMessageRecoverer}.
     * @param consumer Pulsar consumer
     * @return {@link PulsarMessageRecoverer}.
     */
    PulsarMessageRecoverer<T> recovererForConsumer(Consumer<T> consumer);

}
```

The `recovererForConsumer` method takes a Pulsar consumer and returns a `PulsarMessageRecoverer` which is another functional interface. Here is the API of `PulsarMessageRecoverer`:

```
public interface PulsarMessageRecoverer<T> {

    /**
     * Recover a failed message, for e.g. send the message to a DLT.
     * @param message Pulsar message
     * @param exception exception from failed message
     */
    void recoverMessage(Message<T> message, Exception exception);

}
```

Spring for Apache Pulsar provides an implementation for `PulsarMessageRecovererFactory` called `PulsarDeadLetterPublishingRecoverer` that provides a default implementation that is capable of recovering the message by sending it to a DLT - (Dead Letter Topic). This is the implementation that we are providing to the constructor for `DefaultPulsarConsumerErrorHandler` above. As the second argument, we are providing a `FixedBackOff`. You can also provide the `ExponentialBackoff` from Spring for advanced backoff features. Then we provide this bean name for the `PulsarConsumerErrorHandler` as a property to the `PulsarListener`. The property is called `pulsarConsumerErrorHandler`. Each time the `PulsarListener` method fails for a message, it gets retried. The number of retries are controlled by the `Backoff` implementation values provided - in our example, we do 10 retries - 11 total tries all in all - the first one and then the 10 retries. Once all the retries are exhausted, the message is sent to the DLT topic.

The `PulsarDeadLetterPublishingRecoverer` implementation we provide use a `PulsarTemplate` that is uses for publishing the message to the DLT. In most cases, the same auto-configured `PulsarTemplate` from Spring Boot is sufficient with the caveat for partitioned topics. When using partitioned topics and using custom message routing for the main topic, you must use a different `PulsarTemplate` that

does not take the autoconfigured `PulsarProducerFactory` that is populated with a value of `custompartition` for `message-routing-mode`. Towards this extent, you can use a `PulsarConsumerErrorHandler` with the following blueprint.

```
@Bean
PulsarConsumerErrorHandler<Integer> pulsarConsumerErrorHandler(PulsarClient
pulsarClient) {
    PulsarProducerFactory<Integer> pulsarProducerFactory = new
DefaultPulsarProducerFactory<>(pulsarClient, Map.of());
        PulsarTemplate<Integer> pulsarTemplate = new
PulsarTemplate<>(pulsarProducerFactory);

        BiFunction<Consumer<?>, Message<?>, String> destinationResolver =
                (c, m) -> "my-foo-dlt";

        PulsarDeadLetterPublishingRecoverer<Integer>
pulsarDeadLetterPublishingRecoverer =
                new PulsarDeadLetterPublishingRecoverer<>(pulsarTemplate,
destinationResolver);

        return new
DefaultPulsarConsumerErrorHandler<>(pulsarDeadLetterPublishingRecoverer,
                new FixedBackOff(100, 5));
}
```

Note that, we are providing a destination resolver to the `PulsarDeadLetterPublishingRecoverer` as the second constructor argument. If not provided, `PulsarDeadLetterPublishingRecoverer` will use `<subscription-name>-<topic-name>-DLT>` as the DLT topic name. When using this feature, it is recommended to use a properr destination name by setting the destination resolver rather than using the default.

When using a single record message listener as we did above with `PulsarConsumerErrorHnadler` and if you are using manual acknowledgement, make sure not to negatively acknowledge the message when an exception is thrown. Rather, just simply rethrow the exception back to the container; otherwise, the container thinks that the message is handled separately and the error handling will not be triggered.

Finally, we have a second `PulsarListener` above that is receiving messages from the DLT topic.

In the examples provided in this section so far, we only saw how to use `PulsarConsumerErrorHandler` with a single record message listener. Next, we will look how can use this on batch listeners.

Batch listener with PulsarConsumerErrorHandler

First, let us look at a batch `PulsarListener` method.

```
@PulsarListener(subscriptionName = "batch-demo-5-sub", topics = "batch-demo-4",
batch = true, concurrency = "3",
            subscriptionType = SubscriptionType.Failover,
            pulsarConsumerErrorHandler = "pulsarConsumerErrorHandler", ackMode =
AckMode.MANUAL)
void listen(List<Message<Integer>> data, Consumer<Integer> consumer,
Acknowledgment acknowledgment) {
    for (Message<Integer> datum : data) {
        if (datum.getValue() == 5) {
            throw new PulsarBatchListenerFailedException("failed", datum);
        }
        acknowledgement.acknowledge(datum.getMessageId());
    }
}

@Bean
PulsarConsumerErrorHandler<String> pulsarConsumerErrorHandler(
        PulsarTemplate<String> pulsarTemplate) {
    return new DefaultPulsarConsumerErrorHandler<>(
            new PulsarDeadLetterPublishingRecoverer<>(pulsarTemplate, (c, m) ->
"my-foo-dlt"), new FixedBackOff(100, 10));
}

@PulsarListener(subscriptionName = "my-dlt-subscription", topics = "my-foo-dlt")
void dltReceiver(Message<Integer> message) {
    System.out.println("DLT - RECEIVED: " + message.getValue());
}
```

Once again, we re providing the property `pulsarConsumerErrorHandler` with the `PulsarConsumerErrorHandler` bean name. When you are using a batch listener as above and want to use the `PulsarConsumerErrorHandler` from Spring for Apache Pulsar, then you need to use manual acknowledgment This way you can acknowledge all the successful individual messages. For the ones that fail, you must throw a `PulsarBatchListenerFailedException` with the message that it fails on. Without this exception, the framework will not know what to do with the failure. On retry, the container will send a new batch of messages, starting with the failed message to the listener. If it fails again, it is retried, until the retries are exhausted, at which point the message will be sent to the DLT. At that point, the message is acknowledged by the container and the listener will be handed over with the subsequent messages in the original batch.

3.1.21. Intercepting messages

Intercept messages on the Producer

Adding a `ProducerInterceptor` allows you to intercept and mutate messages received by the producer before being published to the brokers. To do so, you can pass a list of interceptors into the `PulsarTemplate` constructor. When using multiple interceptors, the order they are applied in will be the order they appear in the list.

If you are using Spring Boot auto-configuration, you can simply specify the interceptors as Beans. They will be passed automatically to the `PulsarTemplate`. Ordering of the interceptors is achieved by using the `@Order` annotation as seen below.

```
@Bean
@Order(100)
ProducerInterceptor firstInterceptor() {
  ...
}

@Bean
@Order(200)
ProducerInterceptor secondInterceptor() {
  ...
}
```

3.1.22. Pulsar Admin

On the Pulsar administration side, Spring Boot auto-configuration provides a `PulsarAdministration` to manage Pulsar clusters. The administration implements an interface called `PulsarAdminOperations` and provides a 'createOrModify' method to handle topic administration through its contract.

When using the Pulsar Spring Boot Starter, you get the `PulsarAdministration` auto-configured. By default, the application tries to connect to a local Pulsar instance at `localhost:8080`. However, there are many application properties available to configure the client.

Click here to view the available **Pulsar Administration Properties**.

Name	Description	Default Value
`spring.pulsar.administration.auth-params`	Authentication parameter(s) as a JSON encoded string.	
`spring.pulsar.administration.auth-plugin-class-name`	Fully qualified class name of the authentication plugin.	
`spring.pulsar.administration.authentication.*`	Authentication parameter(s) as a map of parameter names to parameter values.	
`spring.pulsar.administration.service-url`	Pulsar service URL for the admin endpoint.	
`spring.pulsar.administration.ssl-provider`	Name of the security provider used for SSL connections.	
`spring.pulsar.administration.tls-allow-insecure-connection`	Whether the client accepts untrusted TLS certificates from the broker.	`false`
`spring.pulsar.administration.tls-ciphers`	Comma-separated list of cipher suites. This is a named combination of authentication, encryption, MAC and key exchange algorithm used to negotiate the security settings for a network connection using TLS or SSL network protocol. By default, all the available cipher suites are supported.	
`spring.pulsar.administration.tls-hostname-verification-enable`	Whether the hostname is validated when the proxy creates a TLS connection with brokers.	`false`
`spring.pulsar.administration.tls-protocols`	Comma-separated list of SSL protocols used to generate the SSLContext. Allowed values in recent JVMs are TLS, TLSv1.3, TLSv1.2 and TLSv1.1.	

Name	Description	Default Value
spring.pulsar.administration.tls-trust-certs-file-path	Path to the trusted TLS certificate file.	
spring.pulsar.administration.tls-trust-store-password	Store password for the key store file.	
spring.pulsar.administration.tls-trust-store-path	Location of the trust store file.	
spring.pulsar.administration.tls-trust-store-type	File format of the trust store file.	
spring.pulsar.administration.use-key-store-tls	Enable KeyStore instead of PEM type configuration if TLS is enabled.	false

Authentication

When accessing a Pulsar cluster that requires authentication, the admin client requires the same security configuration as the regular Pulsar Client. You can use the aforementioned security configuration by simply replacing `spring.pulsar.client` with `spring.pulsar.administration`.

3.1.23. Auto-topic creation

On initialization, the `PulsarAdministration` checks if there are any `PulsarTopic` beans in the application context. For all such beans, the `PulsarAdministration` will either create the corresponding topic, or if necessary modify the number of partitions.

Below is an example how to add `PulsarTopic` beans to let the `PulsarAdministration` auto-create topics for you.

```
@Bean
PulsarTopic simpleTopic {
    // This will create a non-partitioned topic in the public/default namespace
    return PulsarTopic.builder("simple-topic").build();
}

@Bean
PulsarTopic partitionedTopic {
    // This will create a partitioned topic with 3 partitions in the provided
tenant and namespace
    return PulsarTopic.builder("persistent://my-tenant/my-namespace/partitioned-
topic", 3).build();
}
```

3.1.24. Appendix

The reference documentation has the following appendices:

Application Properties	Application properties that you can use to configure your Pulsar application.

Chapter 4. Other Resources

In addition to this reference documentation, we recommend a number of other resources that may help you learn about Spring and Apache Pulsar.

- Apache Pulsar Project Home Page
- Apache Pulsar Java Client
- Spring for Apache Pulsar GitHub Repository
- Apache Pulsar GitHub Repository