Spring for Apache Pulsar

Table of Contents

Soby Chacko; Chris Bono; Alexander Preuß; Jay Bryant; Christophe Bornet (v1.0.0-M1)

Chapter 1. Introduction

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 a Pulsar topic. In addition, it also provides various convenience APIs for Spring developers to ramp up their development journey into Apache Pulsar.

1.1. Project Status

While the code and functionality of the framework is thoroughly tested and ready for production, the project is in a bit of flux while transitioning from experimental GA (`0.2.x`) to GA (`1.0.0`).

The evolution of the project (by version) is as follows:

Version 0.2.x

All components (`the core + Spring Boot autoconfiguration + Spring Cloud Stream binder`) reside in the `spring-pulsar` Github repository.

This is considered an "experimental" GA, defined as:

- code and functionality is production ready, but decision has not been made to support the project permanently
- available on Maven Central using a major version of `0` to indicate its nature

Version 1.0.x

This is the "full" GA and the components are split as follows (in-progress):

- the core stays in the `spring-pulsar` repo (`main` branch)
- the autoconfiguration moves to the Spring Boot project (targeting Spring Boot `3.2.0`)
- the binder moves to the Spring Cloud Stream project (targeting SCSt `4.1.0`)

Until the GA release, the recommended version of the framework is `0.2.x`.

1.2. Supported Versions

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

Library	Minimum Version	Version Used
Apache Pulsar	3.0.0	3.0.0
Spring Framework	6.0.9	6.1.0-M3
Spring Boot	3.2.0	3.2.0-SNAPSHOT

Library	Minimum Version	Version Used
Java	17	-
Gradle	7.4	-

The **Version Used** is the version of the library targeted by Spring Pulsar 1.0.0-M1 (the version of the framework this reference guide belongs to).

Details for other versions and compatibility can be found in the appendix.

1.3. Building the Project

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

Spring for Apache Pulsar uses Gradle as its 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
```

1.4. Getting Help

If you have trouble with Spring for Apache Pulsar, we would like to help.

- Learn the Spring basics. Spring for Apache Pulsar builds on several other Spring projects. Check the spring.io web-site for a wealth of reference documentation. If you are starting out with Spring, try one of the guides.
- Ask a question. We monitor stackoverflow.com for questions tagged with `spring-pulsar`.
- Report bugs at github.com/spring-projects/spring-pulsar/issues.

All of Spring for Apache Pulsar is open source, including the documentation. If you find problems with the docs or if you want to improve them, please get involved.

Chapter 2. Reference

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

2.1. Using Spring for Apache Pulsar

2.1.1. Preface

We recommend using a Spring-Boot-First approach for Spring for Apache Pulsar-based applications, as that simplifies things tremendously. To do so, you can add the `spring-pulsar-spring-boot-starter` module as a dependency.

The majority of this reference expects the reader to be using the starter and gives most directions for configuration with that in mind. However, an effort is made to call out when instructions are specific to the Spring Boot starter usage.

2.1.2. Quick Tour

We will take a quick tour of Spring for Apache Pulsar by showing a sample Spring Boot application that produces and consumes. This is a complete application and does not require any additional configuration, as long as you have a Pulsar cluster running on the default location - `localhost:6650`.

Dependencies

Spring Boot applications need only the `spring-pulsar-spring-boot-starter` dependency. The following listings show 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>0.2.1-SNAPSHOT</version>
    </dependency>
</dependencies>
```

Gradle

```
dependencies {
    implementation 'org.springframework.pulsar:spring-pulsar-spring-boot-
starter:0.2.1-SNAPSHOT'
}
```

Application Code

The following listing shows the Spring Boot application case for the example:

```
@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-topic", "Hello Pulsar
World!");
    }

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

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

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

Spring Boot also auto-configures `PulsarTemplate`, 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, because Spring for Apache Pulsar library automatically infers the schema type from the type of the data that you send.

We use the `PulsarListener` annotation to consume from the `hello-pulsar` topic where we publish the data. `PulsarListener` is a convenience annotation that wraps the message listener container infrastructure in Spring for Apache Pulsar. Behind the scenes, it creates a message listener container to create and manage 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. The framework also infers the schema type used from the data type that the `PulsarListner` method uses as the payload — `String`, in this case.

2.1.3. Pulsar Client

When you use the Pulsar Spring Boot Starter, you get the `PulsarClient` auto-configured.

By default, the application tries to connect to a local Pulsar instance at `pulsar://localhost:6650`. This can be adjusted by setting the `spring.pulsar.client.service-url` property to a different value.

The value must be a valid Pulsar Protocol URL

You can further configure the client by specifying any of the `spring.pulsar.client.*` application properties.

If you are not using the starter, you will need to configure and register the `PulsarClient` yourself. There is a `DefaultPulsarClientFactory` that accepts a builder customizer that can be used to help with this.

Authentication

To connect to a Pulsar cluster that requires authentication, you need to set the `authPluginClassName` and any parameters required by the authentication plugin. You can set the parameters as a single JSON-encoded string or as map of parameter names to parameter values. The following listings show both approaches:

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 listings show 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
```

You can find more information on each of the schemes and their required properties in the official Pulsar security documentation.

2.1.4. Message Production

Pulsar Template

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 block calls by 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`, which you can use to asynchronously receive the message ID once the messages are published.

For the API variants that do not include a topic parameter, a topic resolution process is used to determine the destination topic.

Simple API

The template provides a handful of methods (prefixed with *'send'*) for simple send requests. For more complicated send requests, a fluent API lets you configure more options.

Fluent API

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

Message customization

You can specify a `TypedMessageBuilderCustomizer` 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

You can specify a `ProducerBuilderCustomizer` 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 methods (such as `create`) may have unintended side effects.

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

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

This other example shows how to use custom routing when publishing records to partitioned topics. Specify your custom `MessageRouter` implementation on the `Producer` builder such as:

```
template.newMessage(msg)
    .withProducerCustomizer((pb) -> pb.messageRouter(messageRouter))
    .send();
```

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

This other example shows how to add a `ProducerInterceptor` that will intercept and mutate messages received by the producer before being published to the brokers:

```
template.newMessage(msg)
    .withProducerCustomizer((pb) -> pb.intercept(interceptor))
    .send();
```

Specifying Schema Information

If you use Java primitive types, the framework auto-detects the schema for you, and you need not specify any schema types for publishing the data. For non-primitive types, if the Schema is not explicitly specified when invoking send operations on the `PulsarTemplate`, the Spring Pulsar framework will try to build a `Schema.JSON` from the type.

Complex Schema types that are currently supported are JSON, AVRO, PROTOBUF, and KEY_VALUE w/ INLINE encoding.

Custom Schema Mapping

As an alternative to specifying the schema when invoking send operations on the `PulsarTemplate` for complex types, the schema resolver can be configured with mappings for the types. This removes the need to specify the schema as the framework consults the resolver using the outgoing message type.

Schema mappings can be configured with the `spring.pulsar.defaults.type-mappings` property. The following example uses `application.yml` to add mappings for the `User` and `Address` complex objects using `AVRO` and `JSON` schemas, respectively:

```
spring:
  pulsar:
    defaults:
      type-mappings:
        - message-type: com.acme.User
          schema-info:
            schema-type: AVRO
        - message-type: com.acme.Address
          schema-info:
            schema-type: JSON
```

The `message-type` is the fully-qualified name of the message class.

The preferred method of adding mappings is via the property mentioned above. However, if more control is needed you can provide a schema resolver customizer to add the mapping(s).

The following example uses a schema resolver customizer to add mappings for the `User` and `Address` complex objects using `AVRO` and `JSON` schemas, respectively:

```
@Bean
public SchemaResolverCustomizer<DefaultSchemaResolver> schemaResolverCustomizer()
{
    return (schemaResolver) -> {
        schemaResolver.addCustomSchemaMapping(User.class,
Schema.AVRO(User.class));
        schemaResolver.addCustomSchemaMapping(Address.class,
Schema.JSON(Address.class));
    }
}
```

With this configuration in place, there is no need to set specify the schema on send operations.

Pulsar Producer Factory

The `PulsarTemplate` relies on a `PulsarProducerFactory` to actually create the underlying producer. Spring Boot auto-configuration also provides this producer factory which you can further configure by specifying any of the `spring.pulsar.producer.*` application properties.

If topic information is not specified when using the producer factory APIs directly, the same topic resolution process used by the `PulsarTemplate` is used with the one exception that the "Message type default" step is **omitted**.

Pulsar Producer Caching

Each underlying Pulsar producer consumes resources. 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.pulsar.producer.cache.*` application properties.

Intercept Messages on the Producer

Adding a `ProducerInterceptor` lets you intercept and mutate messages received by the producer before they are 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 is the order in which they appear in the list.

If you use Spring Boot auto-configuration, you can specify the interceptors as Beans. They are passed automatically to the `PulsarTemplate`. Ordering of the interceptors is achieved by using the `@Order` annotation as follows:

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

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

If you are not using the starter, you will need to configure and register the aforementioned components yourself.

2.1.5. Message Consumption

Pulsar Listener

When it comes to Pulsar consumers, we recommend that end-user applications use the `PulsarListener` annotation. To use `PulsarListener`, you need to use the `@EnablePulsar` annotation. When you use 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` to create and manage the Pulsar consumer.

Spring Boot auto-configuration also provides this consumer factory which you can further configure by specifying any of the `spring.pulsar.consumer.*` application properties.

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);
}
```

You can further simplify this method:

```
@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:
  topics: hello-pulsar
  subscription-name: hello-pulsar-subscription
```

If the topic information is not directly provided, a topic resolution process is used to determine the destination topic.

In the `PulsarListener` method shown earlier, we receive the data as `String`, but we do not 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 expect 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, and others), the framework cannot do this inference and the user needs to provide the schema type on the annotation using the `schemaType` property.

The following example shows another `PulsarListener` method, which takes an `Integer`:

```
@PulsarListener(subscriptionName = "my-subscription-1", topics = "my-topic-1")
public void listen(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 listen(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.

Let us look at a few more ways.

You can consume the Pulsar message directly:

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

The following example consumes the record by using the Spring messaging envelope:

```
@PulsarListener(subscriptionName = "my-subscription", topics = "my-topic")
public void listen(org.springframework.messaging.Message<String> message) {
    System.out.println(message.getPayload());
}
```

Now let us see how we can consume records in batches. The following example uses `PulsarListener` to consume records in batches as POJOs:

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

Note that, in this example, we receive the records as a collection (`List`) of objects. In addition, 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, you still need to provide the `schemaType` on `PulsarListener`.

The following uses the `Message` envelope provided by the Pulsar Java client:

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

The following example consumes batch records with an envelope of the Spring messaging `Message` type:

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

Finally, you can also use the `Messages` holder object from Pulsar for the batch listener:

```
@PulsarListener(subscriptionName = "hello-batch-subscription", topics = "hello-
batch", schemaType = SchemaType.JSON, batch = true)
public void listen(org.apache.pulsar.client.api.Messages<Foo>> messages) {
    System.out.println("records received :" + messages.size());
    messages.forEach((message) -> System.out.println("record : " +
message.getValue()));
}
```

When you use `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 earlier or have multiple `PulsarListener` methods.

The following example uses Pulsar consumer properties directly on `PulsarListener`:

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

The properties used are direct Pulsar consumer properties, not the `spring.pulsar.consumer` application configuration properties

Specifying Schema Information

As indicated earlier, for Java primitives, the Spring Pulsar framework can infer the proper Schema to use on the `PulsarListener`. For non-primitive types, if the Schema is not explicitly specified on the annotation, the Spring Pulsar framework will try to build a `Schema.JSON` from the type.

Complex Schema types that are currently supported are JSON, AVRO, PROTOBUF, and KEY_VALUE w/ INLINE encoding.

Custom Schema Mapping

As an alternative to specifying the schema on the `PulsarListener` for complex types, the schema resolver can be configured with mappings for the types. This removes the need to set the schema on the listener as the framework consults the resolver using the incoming message type.

Schema mappings can be configured with the `spring.pulsar.defaults.type-mappings` property. The following example uses `application.yml` to add mappings for the `User` and `Address` complex objects using `AVRO` and `JSON` schemas, respectively:

```
spring:
  pulsar:
    defaults:
      type-mappings:
        - message-type: com.acme.User
          schema-info:
            schema-type: AVRO
        - message-type: com.acme.Address
          schema-info:
            schema-type: JSON
```

The `message-type` is the fully-qualified name of the message class.

The preferred method of adding mappings is via the property mentioned above. However, if more control is needed you can provide a schema resolver customizer to add the mapping(s).

The following example uses a schema resolver customizer to add mappings for the `User` and `Address` complex objects using `AVRO` and `JSON` schemas, respectively:

```
@Bean
public SchemaResolverCustomizer<DefaultSchemaResolver> schemaResolverCustomizer()
{
    return (schemaResolver) -> {
        schemaResolver.addCustomSchemaMapping(User.class,
Schema.AVRO(User.class));
        schemaResolver.addCustomSchemaMapping(Address.class,
Schema.JSON(Address.class));
    }
}
```

With this configuration in place, there is no need to set the schema on the listener, for example:

```
@PulsarListener(subscriptionName = "user-sub", topics = "user-topic")
public void listen(User user) {
    System.out.println(user);
}
```

Accessing the Pulsar Consumer Object

Sometimes, you need direct access to the Pulsar Consumer object. The following example shows how to get it:

```
@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, do NOT 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.

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 the

`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. This section goes through those details.

As briefly mentioned earlier, the message listener container is at the heart of message consumption when you use 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 by 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 you use Spring for Apache Pulsar.

- PulsarRecordMessageListener
- PulsarAcknowledgingMessageListener
- PulsarBatchMessageListener
- PulsarBatchAcknowledgingMessageListener

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

Before doing so, however, let us take a closer look at the container itself.

DefaultPulsarMessageListenerContainer

This is a single consumer-based message listener container. The following listing shows its constructor:

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

It receives a `PulsarConsumerFactory` (which it uses to create the consumer) and a `PulsarContainerProperties` object (which 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. The following example uses 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;
```

If topic information is not specified when using the listener containers directly, the same topic resolution process used by the `PulsarListener` is used with the one exception that the "Message type default" step is **omitted**.

`DefaultPulsarMessageListenerContainer` creates only 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` lets you specify a `concurrency` property through a setter. Concurrency of more than `1` is allowed only 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.

The following example enables `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 preceding listener, it is assumed that the topic `my-topic` has three partitions. If it is a non-partitioned topic, having concurrency set to `3` does nothing. You get two idle consumers in addition to the main active one. If the topic has more than three partitions, messages are load-balanced across the consumers that the container creates. If you run this `PulsarListener`, you see that messages from different partitions are consumed through different consumers, as implied by the thread name and consumer names printouts in the preceding example.

When you use the `Failover` subscription this way on partitioned topics, Pulsar guarantees message ordering.

The following listing shows 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 preceding example, the `PulsarListener` creates five different consumers (this time, we assume that the topic has five partitions).

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, you need to use the `Key_Shared` subscription type.

Message Consumption

Let us take a look at how the message listener container enables both single-record and batch-based message consumption.

Single Record Consumption

Let us revisit 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, we essential ask 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. This 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 invokes the listener method through an adapter for `PulsarRecordMessageListener`. As you can see in the previous section, `PulsarRecordMessageListener` extends from the `MessageListener` provided by the Pulsar Java client, and it supports the basic `received` method.

Batch Consumption

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

Pulsar Headers

The Pulsar message metadata can be consumed as Spring message headers. The list of available headers can be found in PulsarHeaders.java.

Accessing in Single Record based Consumer

The following example shows how you can access the various Pulsar Headers in an application that uses the single record mode of consuming:

```
@PulsarListener(topics = "simpleListenerWithHeaders")
void simpleListenerWithHeaders(String data, @Header(PulsarHeaders.MESSAGE_ID)
MessageId messageId,
                @Header(PulsarHeaders.RAW_DATA) byte[] rawData,
                @Header("foo") String foo) {

}
```

In the preceding example, we access the values for the `messageId` and `rawData` message metadata as well as a custom message property named `foo`. The Spring `@Header` annotation is used for each header field.

You can also use Pulsar's `Message` as the envelope to carry the payload. When doing so, the user can directly call the corresponding methods on the Pulsar message for retrieving the metadata. However, as a convenience, you can also retrieve it by using the `Header` annotation. Note that you can also use the Spring messaging `Message` envelope to carry the payload and then retrieve the Pulsar headers by using `@Header`.

Accessing in Batch Record based Consumer

In this section, we see how to access the various Pulsar Headers in an application that uses a batch consumer:

```
@PulsarListener(topics = "simpleBatchListenerWithHeaders", batch = true)
void simpleBatchListenerWithHeaders(List<String> data,
                    @Header(PulsarHeaders.MESSAGE_ID) List<MessageId> messageIds,
                    @Header(PulsarHeaders.TOPIC_NAME) List<String> topicNames,
@Header("foo") List<String> fooValues) {

}
```

In the preceding example, we consume the data as a `List<String>`. When extracting the various headers, we do so as a `List<>` as well. Spring Pulsar ensures that the headers list corresponds to the data list.

You can also extract headers in the same manner when you use the batch listener and receive payloads as `List<org.apache.pulsar.client.api.Message<?>`, `org.apache.pulsar.client.api.Messages<?>`, or `org.springframework.messaging.Messsge<?>`.

Message Acknowledgment

When you use 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.

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 see how acknowledgment works when you use both single and batch versions of `PulsarListener` and how they translate to the backing message listener container (and, ultimately, to the Pulsar consumer).

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 you use `PulsarListener`, especially if you are familiar with using Pulsar consumer directly. The answer comes down to the message listener container, as that is the central place in Spring for Apache Pulsar that coordinates all the consumer related activities.

Assuming you are not overriding the default behavior, this is what happens behind the scenes when you use the preceding `PulsarListener`:

1. First, the listener container receives messages as batches 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 acknowledges all the messages from the original batch.

This is the normal flow. If any records from the original batch throw an exception, Spring for Apache Pulsar track those records separately. When all the records from the batch are processed, Spring for Apache Pulsar acknowledges all the successful messages and negatively acknowledges (nack) all the failed messages. In other words, when consuming single records by using

PulsarRecordMessageListener and the default ack mode of BATCH is used, the framework waits for all the records received from the batchReceive call to process successfully and then calls 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 calls negativeAcknowledge on those records after the entire batch is processed.

If the application wants the acknowledgment or negative acknowledgment to occur per record, 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. The following example enables RECORD ack mode on the 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 need not set this on the PulsarListener annotation. In that case, all the PulsarListener methods in the application acquire that property.

Manual Message Ack in Single Record Mode

You might not always want the framework to send acknowledgments but, rather, do that directly from the application itself. Spring for Apache Pulsar provides a couple of ways to enable manual message acknowledgments. The following example shows one of them:

```
@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();
}
```

A few things merit explanation here. First, we enablE manual ack mode by setting ackMode on PulsarListener. When enabling manual ack mode, Spring for Apache Pulsar lets the application inject an Acknowledgment object. 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 into your `PulsarListener` while using `MANUAL` ack mode and then call one of the corresponding methods.

In the preceding `PulsarListener` example, we call a parameter-less `acknowledge` method. This is because the framework knows which `Message` it is currently operating under. When calling `acknowledge()`, you need not receive the payload with the `Message` enveloper` but, rather, use 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 you use `acknowledge(messageId)`, you must receive the payload by using the `Message<?>` envelope.

Similar to what is possible for acknowledging, the `Acknowledgment` API also provides options for negatively acknowledging. See the nack methods shown earlier.

You can also call `acknowledge` directly on the Pulsar consumer:

```
@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) {
        ....
    }
}
```

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

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

Automatic Message Ack in Batch Consumption

When you consume records in batches (see "Message ACK modes") and you use the default ack mode of `BATCH` is used, when the entire batch is processed successfully, the entire batch is acknowledged. If any records throw an exception, 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

Consider 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 acknowledges all of them.

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

Manual Message Ack in Batch Consumption

As seen in the previous section, when `MANUAL` ack mode is set on the message listener container, the framework does 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. The following are the methods available 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 into your `PulsarListener` while using `MANUAL` ack mode. The following listing shows 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 you use a batch listener, the message listener container cannot know which record it is currently operating upon. Therefore, 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.

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. We take a look at them and see how we can use them through Spring for Apache Pulsar.

Specifying Acknowledgment Timeout for Message Redelivery

By default, Pulsar consumers does not redeliver messages unless the consumer crashes, but you can change this behavior by setting an ack timeout on the Pulsar consumer. When you use Spring for Apache Pulsar, you can enable this property by setting the `spring.pulsar.consumer.ack-timeout` Boot

property. If this property has a value above zero and if the Pulsar consumer does not acknowledge a message within that timeout period, the message is redelivered.

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

```
@PulsarListener(subscriptionName = "subscription-1", topics = "topic-1"
                properties = {"ackTimeout=60s"})
public void listen(String s) {
    ...
}
```

When you specify `ackTimeout` (as seen in the preceding `PulsarListener` method), if the consumer does not send an acknowledgement within 60 seconds, the message is redelivered by Pulsar to the consumer.

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

```
@EnablePulsar
@Configuration
class AckTimeoutRedeliveryConfig {

    @PulsarListener(subscriptionName =
"withAckTimeoutRedeliveryBackoffSubscription",
            topics = "withAckTimeoutRedeliveryBackoff-test-topic",
            ackTimeoutRedeliveryBackoff = "ackTimeoutRedeliveryBackoff",
            properties = { "ackTimeout=60s" })
    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 preceding example, we specify a bean for Pulsar's `RedeliveryBackoff` with a minimum delay of 1 second, a maximum delay of 10 seconds, and a backoff multiplier of 2. After the initial ack timeout occurs, the message redeliveries are 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 lets you specify how the application wants the message to be re-delivered. The default is to redeliver the message in one minute, but you can change it by setting `spring.pulsar.consumer.negative-ack-redelivery-delay`. You can also set it as a consumer property directly on `PulsarListener`, as follows:

```
@PulsarListener(subscriptionName = "subscription-1", topics = "topic-1"
                properties = {"negativeAckRedeliveryDelay=10ms"})
public void listen(String s) {
    ...
}
```

You can also specify different delays and backoff mechanisms with a multiplier by providing a `RedeliveryBackoff` bean and providing the bean name as the `negativeAckRedeliveryBackoff` property on the PulsarProducer, as follows:

```
@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 lets applications use a dead letter topic on consumers with a `Shared` subscription

type. For the `Exclusive` and `Failover` subscription types, this feature is not available. The basic idea is that, if a message is retried a certain number of times (maybe due to an ack timeout or nack redelivery), once the number of retries are exhausted, the message can be sent to a special topic called the dead letter queue (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 = {
"ackTimeout=1s" })
    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();
    }

}
```

First, we have a special bean for `DeadLetterPolicy`, and it is named as `deadLetterPolicy` (it can 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`, in this case. If you do not specify a DLQ topic name, it defaults to `<topicname>-<subscriptionname>-DLQ` in Pulsar. Next, we provide this bean name to `PulsarListener` by setting the `deadLetterPolicy` property. 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 `ackTimeout` value of 1 second. The idea is that the code throws the exception and, if Pulsar does not receive an ack within 1 second, it does a retry. If that cycle continues ten times (as that is our max redelivery count in the `DeadLetterPolicy`), the Pulsar consumer publishes the messages to the DLQ topic. We have another `PulsarListener` that listens 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, 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), Pulsar publishes to a default topic name that has this `` `partition-<n> `` info in it — for example: `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 noted earlier, the DLQ feature in Apache Pulsar works only for shared subscriptions. What does an application do if it needs to use some similar feature for non-shared subscriptions? The main reason Pulsar does not support DLQ on exclusive and failover subscriptions is because those subscription types are order-guaranteed. Allowing redeliveries, DLQ, and so on effectively receives messages out of order. However, what if an application 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 you can use across any subscription types in Pulsar: `Exclusive`, `Failover`, `Shared`, or `Key_Shared`.

When you use `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);
    }

}
```

Consider the `pulsarConsumerErrorHandler` bean. 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 can recover the message by sending it to a Dead Letter Topic (DLT). We provide this implementation to the constructor for the preceding `DefaultPulsarConsumerErrorHandler`. As the second argument, we provide 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` provided implementation values. In our example, we do 10 retries (11 total tries — 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 uses a `PulsarTemplate` that is used 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 auto-configured `PulsarProducerFactory` that is populated with a value of `custompartition` for `message-routing-mode`. 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 provide a destination resolver to the `PulsarDeadLetterPublishingRecoverer` as the second constructor argument. If not provided, `PulsarDeadLetterPublishingRecoverer` uses `<subscription-name>-<topic-name>-DLT>` as the DLT topic name. When using this feature, you should use a proper destination name by setting the destination resolver rather than using the default.

When using a single record message listener, as we did with `PulsarConsumerErrorHnadler`, and if you use manual acknowledgement, make sure to not negatively acknowledge the message when an exception is thrown. Rather, re-throw the exception back to the container. Otherwise, the container thinks the message is handled separately, and the error handling is not triggered.

Finally, we have a second `PulsarListener` that receives 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 look at how you 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 provide the `pulsarConsumerErrorHandler` property with the `PulsarConsumerErrorHandler` bean name. When you use a batch listener (as shown in the preceding example) and want to use the `PulsarConsumerErrorHandler` from Spring for Apache Pulsar, 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 on which it fails. Without this exception, the framework does not know what to do with the failure. On retry, the container sends 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 is sent to the DLT. At that point, the message is acknowledged by the container, and the listener is handed over with the subsequent messages in the original batch.

Consumer Customization on PulsarListener

Spring for Apache Pulsar provides a convenient way to customize the consumer created by the container used by the `PulsarListener`. Applications can provide a bean for `ConsumerBuilderCustomizer`. Here is an example.

```
@Bean
public ConsumerBuilderCustomizer<String> myCustomizer() {
    return cb -> {
        cb.subscriptionName("modified-subscription-name");
    };
}
```

Then this customizer bean name can be provided as an attribute on the `PuslarListener` annotation as shown below.

```
@PulsarListener(subscriptionName = "my-subscription",
        topics = "my-topic", consumerCustomizer = "myCustomizer")
void listen(String message) {

}
```

The framework detects the provided bean through the `PulsarListener` and applies this customizer on the Consumer builder before creating the Pulsar Consumer.

If you have multiple `PulsarListener` methods, and each of them have different customization rules, you should create multiple customizer beans and attach the proper customizers on each `PulsarListener`.

Pausing and Resuming Message Listener Containers

There are situations in which an application might want to pause message consumption temporarily and then resume later. Spring for Apache Pulsar provides the ability to pause and resume the underlying message listener containers. When the Pulsar message listener container is paused, any polling done by the container to receive data from the Pulsar consumer will be paused. Similarly, when the container is resumed, the next poll starts returning data if the topic has any new records added while paused.

To pause or resume a listener container, first obtain the container instance via the `PulsarListenerEndpointRegistry` bean and then invoke the pause/resume API on the container instance - as shown in the snippet below:

```
@Autowired
private PulsarListenerEndpointRegistry registry;

void someMethod() {
  PulsarMessageListenerContainer container = registry.getListenerContainer("my-
listener-id");
  container.pause();
}
```

The id parameter passed to `getListenerContainer` is the container id - which will be the value of the `@PulsarListener` id attribute when pausing/resuming a `@PulsarListener`.

Pulsar Reader Support

The framework provides support for using Pulsar Reader via the `PulsarReaderFactory`.

Spring Boot provides this reader factory which you can further configure by specifying any of the `spring.pulsar.reader.*` application properties.

PulsarReader Annotation

While it is possible to use `PulsarReaderFactory` directly, Spring for Apache Pulsar provides a convenient annotation called `PulsarReader` that you can use to quickly read from a topic without setting up any reader factories yourselves. This is similar to the same ideas behind `PulsarListener`. Here is a quick example.

```
@PulsarReader(id = "pulsar-reader-demo-id", subscriptionName = "pulsar-reader-
demo-subscription",
topics = "pulsar-reader-demo-topic", startMessageId = "earliest")
void read(String message) {
    //...
}
```

As you can see from the above example, `PulsarReader` is a quick way to read from a Pulsar topic. The `id` and `subscriptionName` attributes are optional, but always a best practice to provide them. `PulsarReader` requires `topics` and the `startMessageId` as mandatory attributes. The `topics` attribute can be a single topic or a comma-separated list of topics. The `startMessageId` attribute instructs the reader to start from a particular message in the topic. The valid values for `startMessageId` are `earliest` or `latest`. Suppose you want the reader to start reading messages arbitrarily from a topic other than the earliest or latest available messages. In that case, you need to use a `ReaderBuilderCustomizer` to customize the `ReaderBuilder` so it knows the right `MessageId` to start from.

Customizing the ReaderBuilder

You can customize any fields available through `ReaderBuilder` using a `ReaderBuilderCustomizer` in Spring for Apache Pulsar. You can provide a `@Bean` from `ReaderBuilderCustomizer` and then make it available to the `PulsarReader` as below.

```
@PulsarReader(id = "with-customizer-reader", subscriptionName = "with-customizer-
reader-subscription",
                    topics = "with-customizer-reader-topic", readerCustomizer =
"myCustomizer")
void read(String message) {
    //...
}

@Bean
public ReaderBuilderCustomizer<String> myCustomizer() {
    return readerBuilder -> {
        readerBuilder.startMessageId(messageId); // the first message read is
after this message id.
        // Any other customizations on the readerBuilder
    };
}
```

2.1.6. Topic Resolution

A destination topic is needed when producing or consuming messages. The framework looks in the following ordered locations to determine a topic (stopping at the first find):

- User specified
- Message type default
- Global default

When a topic is found via one of the default mechanisms, there is no need to specify the topic on the produce or consume API.

When a topic is not found, the API will throw an exception accordingly.

User specified

A topic passed into the API being used has the highest precedence (eg. `PulsarTemplate.send("my-topic", myMessage)` or `@PulsarListener(topics = "my-topic")`).

Message type default

When no topic is passed into the API, the system looks for a message type to topic mapping configured for the type of the message being produced or consumed.

Mappings can be configured with the `spring.pulsar.defaults.type-mappings` property. The following

example uses `application.yml` to configure default topics to use when consuming or producing `Foo` or `Bar` messages:

```
spring:
  pulsar:
    defaults:
      type-mappings:
        - message-type: com.acme.Foo
          topic-name: foo-topic
        - message-type: com.acme.Bar
          topic-name: bar-topic
```

The `message-type` is the fully-qualified name of the message class.

If the message (or the first message of a `Publisher` input) is `null`, the framework won't be able to determine the topic from it. Another method shall be used to specify the topic if your application is likely to send `null` messages.

Custom topic resolver

The preferred method of adding mappings is via the property mentioned above. However, if more control is needed you can replace the default resolver by proving your own implementation, for example:

```
@Bean
public MyTopicResolver topicResolver() {
    return new MyTopicResolver();
}
```

Producer global default

The final location consulted (when producing) is the system-wide producer default topic. It is configured via the `spring.pulsar.producer.topic-name` property when using the imperative API and the `spring.pulsar.reactive.sender.topic-name` property when using the reactive API.

Consumer global default

The final location consulted (when consuming) is the system-wide consumer default topic. It is configured via the `spring.pulsar.consumer.topics` or `spring.pulsar.consumer.topics-pattern` property when using the imperative API and one of the `spring.pulsar.reactive.consumer.topics` or `spring.pulsar.reactive.consumer.topics-pattern` property when using the reactive API.

2.1.7. Publishing and Consuming Partitioned Topics

In the following example, we publish 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;
        }
    }
```

```
}
```

In the preceding example, we publish 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. To do so, we provide a message router object with the `send` method. Consider the three message routers implemented. `FooRouter` always sends data to partition `0`, `BarRouter` sends to partition `1`, and `BuzzRouter` sends to partition `2`. Also note that we now use 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 use a `PulsarListener` with the exclusive subscription type. This means that data from all the partitions ends 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:

```
@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 you follow this approach, a single partition always gets consumed by a dedicated consumer.

In a similar vein, if you want to use Pulsar's shared consumer type, you can use the `shared` subscription type. However, when you use 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.

Consider the following 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);
}
```

2.2. Reactive Support

The framework provides a Reactive counterpart for almost all supported features.

If you put the word `Reactive` in front of a provided imperative component, you will likely find its Reactive counterpart.

- `PulsarTemplate` → `ReactivePulsarTemplate`
- `PulsarListener` → `ReactivePulsarListener`
- `PulsarConsumerFactory` → `ReactivePulsarConsumerFactory`
- etc..

However, the following is not yet supported:

- Error Handling in non-shared subscriptions
- Accessing Pulsar headers via `@Header` in streaming mode
- Observations

2.2.1. Preface

We recommend using a Spring-Boot-First approach for Spring for Apache Pulsar-based applications, as that simplifies things tremendously. To do so, you can add the `spring-pulsar-reactive-spring-boot-starter` module as a dependency.

The majority of this reference expects the reader to be using the starter and gives most directions for configuration with that in mind. However, an effort is made to call out when instructions are specific to the Spring Boot starter usage.

2.2.2. Quick Tour

We will take a quick tour of the Reactive support in Spring for Apache Pulsar by showing a sample

Spring Boot application that produces and consumes in a Reactive fashion. This is a complete application and does not require any additional configuration, as long as you have a Pulsar cluster running on the default location - `localhost:6650`.

Dependencies

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

Maven

```
<dependencies>
    <dependency>
        <groupId>org.springframework.pulsar</groupId>
        <artifactId>spring-pulsar-reactive-spring-boot-starter</artifactId>
        <version>0.2.1-SNAPSHOT</version>
    </dependency>
</dependencies>
```

Gradle

```
dependencies {
    implementation 'org.springframework.pulsar:spring-pulsar-reactive-spring-boot-
starter:0.2.1-SNAPSHOT'
}
```

Application Code

Here is the application source code:

```
@SpringBootApplication
public class ReactiveSpringPulsarHelloWorld {

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

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

    @ReactivePulsarListener(subscriptionName = "hello-pulsar-sub", topics = "hello-
pulsar-topic")
    Mono<Void> listen(String message) {
        System.out.println("Reactive listener received: " + message);
        return Mono.empty();
    }
}
```

That is it, with just a few lines of code we have a working Spring Boot app that is producing and consuming messages from a Pulsar topic in a Reactive fashion.

Once started, the application uses a `ReactivePulsarTemplate` to send messages to the `hello-pulsar-topic`. It then consumes from the `hello-pulsar-topic` using a `@ReactivePulsarListener`.

One of the key ingredients to the simplicity is the Spring Boot starter which auto-configures and provides the required components to the application

2.2.3. Design

Here are a few key design points to keep in mind.

Apache Pulsar Reactive

The reactive support is ultimately provided by the Apache Pulsar Reactive client whose current implementation is a fully non-blocking adapter around the regular Pulsar client's asynchronous API. This implies that the Reactive client requires the regular client.

Additive Auto-Configuration

Due to the dependence on the regular (imperative) client, the Reactive auto-configuration provided by the framework is additive to the imperative auto-configuration. In other words, The imperative starter only includes the imperative components but the reactive starter includes both imperative and reactive components.

2.2.4. Reactive Pulsar Client

When you use the Reactive Pulsar Spring Boot Starter, you get the `ReactivePulsarClient` auto-configured.

By default, the application tries to connect to a local Pulsar instance at `pulsar://localhost:6650`. This can be adjusted by setting the `spring.pulsar.client.service-url` property to a different value.

The value must be a valid Pulsar Protocol URL

There are many other application properties (inherited from the adapted imperative client) available to configure. See the `spring.pulsar.client.*` application properties.

Authentication

To connect to a Pulsar cluster that requires authentication, follow the same steps as the imperative client. Again, this is because the reactive client adapts the imperative client which handles all security configuration.

2.2.5. Message Production

ReactivePulsarTemplate

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

The template provides send methods that accept a single message and return a `Mono<MessageId>`. It also provides send methods that accept multiple messages (in the form of the ReactiveStreams `Publisher` type) and return a `Flux<MessageId>`.

For the API variants that do not include a topic parameter, a topic resolution process is used to determine the destination topic.

Fluent API

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

Message customization

You can specify a `MessageSpecBuilderCustomizer` to configure the outgoing message. For example, the following code shows how to send a keyed message:

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

Sender customization

You can specify a `ReactiveMessageSenderBuilderCustomizer` to configure the underlying Pulsar sender builder that ultimately constructs the sender used to send the outgoing message.

Use with caution as this gives full access to the sender builder and invoking some of its methods (such as `create`) may have unintended side effects.

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

```
template.newMessage(msg)
    .withSenderCustomizer((sc) -> sc.enableChunking(true).enableBatching(false))
    .send();
```

This other example shows how to use custom routing when publishing records to partitioned topics. Specify your custom `MessageRouter` implementation on the sender builder such as:

```
template.newMessage(msg)
    .withSenderCustomizer((sc) -> sc.messageRouter(messageRouter))
    .send();
```

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

Specifying Schema Information

If you use Java primitive types, the framework auto-detects the schema for you, and you need not specify any schema types for publishing the data. For non-primitive types, if the Schema is not explicitly specified when invoking send operations on the `ReactivePulsarTemplate`, the Spring Pulsar framework will try to build a `Schema.JSON` from the type.

Complex Schema types that are currently supported are JSON, AVRO, PROTOBUF, and KEY_VALUE w/ INLINE encoding.

Custom Schema Mapping

As an alternative to specifying the schema when invoking send operations on the `ReactivePulsarTemplate` for complex types, the schema resolver can be configured with mappings for the types. This removes the need to specify the schema as the framework consults the resolver using the outgoing message type.

Schema mappings can be configured with the `spring.pulsar.defaults.type-mappings` property. The following example uses `application.yml` to add mappings for the `User` and `Address` complex objects using `AVRO` and `JSON` schemas, respectively:

```
spring:
  pulsar:
    defaults:
      type-mappings:
        - message-type: com.acme.User
          schema-info:
            schema-type: AVRO
        - message-type: com.acme.Address
          schema-info:
            schema-type: JSON
```

The `message-type` is the fully-qualified name of the message class.

The preferred method of adding mappings is via the property mentioned above. However, if more control is needed you can provide a schema resolver customizer to add the mapping(s).

The following example uses a schema resolver customizer to add mappings for the `User` and `Address` complex objects using `AVRO` and `JSON` schemas, respectively:

```
@Bean
public SchemaResolverCustomizer<DefaultSchemaResolver> schemaResolverCustomizer()
{
    return (schemaResolver) -> {
        schemaResolver.addCustomSchemaMapping(User.class,
Schema.AVRO(User.class));
        schemaResolver.addCustomSchemaMapping(Address.class,
Schema.JSON(Address.class));
    }
}
```

With this configuration in place, there is no need to set specify the schema on send operations.

ReactivePulsarSenderFactory

The `ReactivePulsarTemplate` relies on a `ReactivePulsarSenderFactory` to actually create the underlying sender.

Spring Boot provides this sender factory which can be configured with any of the `spring.pulsar.reactive.sender.*` application properties.

If topic information is not specified when using the sender factory APIs directly, the same topic resolution process used by the `ReactivePulsarTemplate` is used with the one exception that the "Message type default" step is **omitted**.

Producer Caching

Each underlying Pulsar producer consumes resources. To improve performance and avoid continual creation of producers, the `ReactiveMessageSenderCache` in the underlying Apache Pulsar Reactive client 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.

You can configure the cache settings by specifying any of the `spring.pulsar.reactive.sender.cache.*` application properties.

2.2.6. Message Consumption

@ReactivePulsarListener

When it comes to Pulsar consumers, we recommend that end-user applications use the `ReactivePulsarListener` annotation. To use `ReactivePulsarListener`, you need to use the `@EnableReactivePulsar` annotation. When you use Spring Boot support, it automatically enables this annotation and configures all necessary components, such as the message listener infrastructure (which is responsible for creating the underlying Pulsar consumer).

Let us revisit the `ReactivePulsarListener` code snippet we saw in the quick-tour section:

```
@ReactivePulsarListener(subscriptionName = "hello-pulsar-sub", topics = "hello-
pulsar-topic")
Mono<Void> listen(String message) {
    System.out.println(message);
    return Mono.empty();
}
```

The listener method returns a `Mono<Void>` to signal whether the message was successfully processed. `Mono.empty()` indicates success (acknowledgment) and `Mono.error()` indicates failure (negative acknowledgment).

You can also further simplify this method:

```
@ReactivePulsarListener
Mono<Void> listen(String message) {
    System.out.println(message);
    return Mono.empty();
}
```

In this most basic form, you must still provide the topic name by setting the following property:

```
spring.pulsar.reactive.consumer:
  topic-names: hello-pulsar-topic
```

When `subscription-name` is not provided an auto-generated subscription name will be used.

If the topic information is not directly provided, a topic resolution process is used to determine the destination topic.

In the `ReactivePulsarListener` method shown earlier, we receive the data as `String`, but we do not 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 expect 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, and others), the framework cannot do this inference and the user needs to provide the schema type on the annotation using the `schemaType` property.

This example shows how we can consume complex types from a topic:

```
@ReactivePulsarListener(topics = "my-topic-2", schemaType = SchemaType.JSON)
Mono<Void> listen(Foo message) {
    System.out.println(message);
    return Mono.empty();
}
```

Note the addition of a `schemaType` property on `ReactivePulsarListener`. 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.

Let us look at a few more ways we can consume.

This example consumes the Pulsar message directly:

```
@ReactivePulsarListener(topics = "my-topic")
Mono<Void> listen(org.apache.pulsar.client.api.Message<String> message) {
    System.out.println(message.getValue());
    return Mono.empty();
}
```

This example consumes the record wrapped in a Spring messaging envelope:

```
@ReactivePulsarListener(topics = "my-topic")
Mono<Void> listen(org.springframework.messaging.Message<String> message) {
    System.out.println(message.getPayload());
    return Mono.empty();
}
```

Streaming

All of the above are examples of consuming a single record one-by-one. However, one of the compelling reasons to use Reactive is for the streaming capability with backpressure support.

The following example uses `ReactivePulsarListener` to consume a stream of POJOs:

```
@ReactivePulsarListener(topics = "streaming-1", stream = true)
Flux<MessageResult<Void>> listen(Flux<Message<String>> messages) {
    return messages
        .doOnNext((msg) -> System.out.println("Received: " + msg.getValue()))
        .map(MessageResult::acknowledge);
```

Here we receive the records as a `Flux` of messages. In addition, to enable stream consumption at the `ReactivePulsarListener` level, you need to set the `stream` property on the annotation to `true`.

The listener method returns a `Flux<MessageResult<Void>>` where each element represents a processed message and holds the message id, value and whether it was acknowledged. The `MessageResult` has a set of static factory methods that can be used to create the appropriate `MessageResult` instance.

Based on the actual type of the messages in the `Flux`, the framework tries to infer the schema to use. If it contains a complex type, you still need to provide the `schemaType` on `ReactivePulsarListener`.

The following listener uses the Spring messaging `Message` envelope with a complex type :

```
@ReactivePulsarListener(topics = "streaming-2", stream = true, schemaType =
SchemaType.JSON)
Flux<MessageResult<Void>> listen2(Flux<org.springframework.messaging.Message<Foo>>
messages) {
    return messages
        .doOnNext((msg) -> System.out.println("Received: " + msg.getPayload()))
        .map(MessageResult::acknowledge);
}
```

Configuration - Application Properties

The listener ultimately relies on `ReactivePulsarConsumerFactory` to create and manage the underlying Pulsar consumer.

Spring Boot provides this consumer factory which can be configured with any of the `spring.pulsar.reactive.consumer.*` application-properties.

Consumer Customization

You can specify a `ReactiveMessageConsumerBuilderCustomizer` to configure the underlying Pulsar consumer builder that ultimately constructs the consumer used by the listener to receive the messages.

Use with caution as this gives full access to the consumer builder and invoking some of its methods (such as `create`) may have unintended side effects.

For example, the following code shows how to set the initial position of the subscription to the earliest messaage on the topic.

```
@ReactivePulsarListener(topics = "hello-pulsar-topic", consumerCustomizer =
"myConsumerCustomizer")
Mono<Void> listen(String message) {
    System.out.println(message);
    return Mono.empty();
}

@Bean
ReactiveMessageConsumerBuilderCustomizer<String> myConsumerCustomizer() {
    return b ->
b.subscriptionInitialPosition(SubscriptionInitialPosition.Earliest);
}
```

You can also use the customizer to provide direct Pulsar consumer properties to the consumer builder. This is convenient if you do not want to use the Boot configuration properties mentioned earlier or have multiple `ReactivePulsarListener` methods whose configuration varies.

The following customizer example uses direct Pulsar consumer properties:

```
@Bean
ReactiveMessageConsumerBuilderCustomizer<String> directConsumerPropsCustomizer() {
    return b -> b.property("subscriptionName", "subscription-
1").property("topicNames", "foo-1");
}
```

The properties used are direct Pulsar consumer properties, not the `spring.pulsar.reactive.consumer` Spring Boot configuration properties

Specifying Schema Information

As indicated earlier, for Java primitives, the Spring Pulsar framework can infer the proper Schema to use on the `ReactivePulsarListener`. For non-primitive types, if the Schema is not explicitly specified on the annotation, the Spring Pulsar framework will try to build a `Schema.JSON` from the type.

Complex Schema types that are currently supported are JSON, AVRO, PROTOBUF, and KEY_VALUE w/ INLINE encoding.

Custom Schema Mapping

As an alternative to specifying the schema on the `ReactivePulsarListener` for complex types, the schema resolver can be configured with mappings for the types. This removes the need to set the schema on the listener as the framework consults the resolver using the incoming message type.

Schema mappings can be configured with the `spring.pulsar.defaults.type-mappings` property. The following example uses `application.yml` to add mappings for the `User` and `Address` complex objects using `AVRO` and `JSON` schemas, respectively:

```
spring:
  pulsar:
    defaults:
      type-mappings:
        - message-type: com.acme.User
          schema-info:
            schema-type: AVRO
        - message-type: com.acme.Address
          schema-info:
            schema-type: JSON
```

The `message-type` is the fully-qualified name of the message class.

The preferred method of adding mappings is via the property mentioned above. However, if more control is needed you can provide a schema resolver customizer to add the mapping(s).

The following example uses a schema resolver customizer to add mappings for the `User` and `Address` complex objects using `AVRO` and `JSON` schemas, respectively:

```
@Bean
public SchemaResolverCustomizer<DefaultSchemaResolver> schemaResolverCustomizer()
{
    return (schemaResolver) -> {
        schemaResolver.addCustomSchemaMapping(User.class,
Schema.AVRO(User.class));
        schemaResolver.addCustomSchemaMapping(Address.class,
Schema.JSON(Address.class));
    }
}
```

With this configuration in place, there is no need to set the schema on the listener, for example:

```
@ReactivePulsarListener(topics = "user-topic")
Mono<Void> listen(User user) {
    System.out.println(user);
    return Mono.empty();
}
```

Message Listener Container Infrastructure

In most scenarios, we recommend using the `ReactivePulsarListener` annotation directly for consuming from a Pulsar topic as that model covers a broad set of application use cases. However, it is important to understand how `ReactivePulsarListener` works internally.

The message listener container is at the heart of message consumption when you use Spring for Apache Pulsar. The `ReactivePulsarListener` uses the message listener container infrastructure behind the scenes to create and manage the underlying Pulsar consumer.

ReactivePulsarMessageListenerContainer

The contract for this message listener container is provided through `ReactivePulsarMessageListenerContainer` whose default implementation creates a reactive Pulsar consumer and wires up a reactive message pipeline that uses the created consumer.

ReactiveMessagePipeline

The pipeline is a feature of the underlying Apache Pulsar Reactive client which does the heavy lifting of receiving the data in a reactive manner and then handing it over to the provided message handler. The reactive message listener container implementation is much simpler because the pipeline handles the majority of the work.

ReactivePulsarMessageHandler

The "listener" aspect is provided by the `ReactivePulsarMessageHandler` of which there are two

provided implementations:

- `ReactivePulsarOneByOneMessageHandler` - handles a single message one-by-one
- `ReactivePulsarStreamingHandler` - handles multiple messages via a `Flux`

If topic information is not specified when using the listener containers directly, the same topic resolution process used by the `ReactivePulsarListener` is used with the one exception that the "Message type default" step is **omitted**.

Concurrency

When consuming records in streaming mode (`stream = true`) concurrency comes naturally via the underlying Reactive support in the client implementation.

However, when handling messages one-by-one, concurrency can be specified to increase processing throughput. Simply set the `concurrency` property on `@ReactivePulsarListener`. Additionally, when `concurrency > 1` you can ensure messages are ordered by key and therefore sent to the same handler by setting `useKeyOrderedProcessing = "true"` on the annotation.

Again, the `ReactiveMessagePipeline` does the heavy lifting, we simply set the properties on it.

Reactive vs Imperative

Concurrency in the reactive container is different from its imperative counterpart. The latter creates multiple threads (each with a Pulsar consumer) whereas the former dispatches the messages to multiple handler instances concurrently on the Reactive parallel scheduler.

One advantage of reactive concurrency is that it can be used with `Exclusive` and `Failover` subscriptions to increase processing throughput if strict ordering is not required. In contrast to imperative concurrency that can not currently be used with `Exclusive` and does not provide more processing power with `Failover`.

Pulsar Headers

The Pulsar message metadata can be consumed as Spring message headers. The list of available headers can be found in PulsarHeaders.java.

Accessing In OneByOne Listener

The following example shows how you can access Pulsar Headers when using a one-by-one message listener:

```
@ReactivePulsarListener(topics = "some-topic")
Mono<Void> listen(String data,
        @Header(PulsarHeaders.MESSAGE_ID) MessageId messageId,
        @Header("foo") String foo) {
    System.out.println("Received " + data + " w/ id=" + messageId + " w/ foo=" +
foo);
    return Mono.empty();
}
```

In the preceding example, we access the values for the `messageId` message metadata as well as a custom message property named `foo`. The Spring `@Header` annotation is used for each header field.

You can also use Pulsar's `Message` as the envelope to carry the payload. When doing so, the user can directly call the corresponding methods on the Pulsar message for retrieving the metadata. However, as a convenience, you can also retrieve it by using the `Header` annotation. Note that you can also use the Spring messaging `Message` envelope to carry the payload and then retrieve the Pulsar headers by using `@Header`.

Accessing In Streaming Listener

When using a streaming message listener the header support is limited. Only when the `Flux` contains Spring `org.springframework.messaging.Message` elements will the headers be populated. Additionally, the Spring `@Header` annotation can not be used to retrieve the data. You must directly call the corresponding methods on the Spring message to retrieve the data.

Message Acknowledgment

The framework automatically handles message acknowledgement. However, the listener method must send a signal indicating whether the message was successfully processed. The container implementation then uses that signal to perform the ack or nack operation. This is a slightly different from its imperative counterpart where the signal is implied as positive unless the method throws an exception.

OneByOne Listener

The single message (aka OneByOne) message listener method returns a `Mono<Void>` to signal whether the message was successfully processed. `Mono.empty()` indicates success (acknowledgment) and `Mono.error()` indicates failure (negative acknowledgment).

Streaming Listener

The streaming listener method returns a `Flux<MessageResult<Void>>` where each `MessageResult` element represents a processed message and holds the message id, value and whether it was acknowledged. The `MessageResult` has a set of `acknowledge` and `negativeAcknowledge` static factory methods that can be used to create the appropriate `MessageResult` instance.

Message Redelivery and Error Handling

Apache Pulsar provides various native strategies for message redelivery and error handling. We will take a look at them and see how to use them through Spring for Apache Pulsar.

Acknowledgment Timeout

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

You can also specify this property directly as a Pulsar consumer property via a consumer customizer such as:

```
@Bean
ReactiveMessageConsumerBuilderCustomizer<String> consumerCustomizer() {
    return b -> b.property("ackTimeout", "60s");
}
```

Negative Acknowledgment Redelivery Delay

When acknowledging negatively, Pulsar consumer lets you specify how the application wants the message to be re-delivered. The default is to redeliver the message in one minute, but you can change it by setting `spring.pulsar.reactive.consumer.negative-ack-redelivery-delay`.

You can also set it directly as a Pulsar consumer property via a consumer customizer such as:

```
@Bean
ReactiveMessageConsumerBuilderCustomizer<String> consumerCustomizer() {
    return b -> b.property("negativeAckRedeliveryDelay", "10ms");
}
```

Dead Letter Topic

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

```
@Configuration(proxyBeanMethods = false)
class DeadLetterPolicyConfig {

    @ReactivePulsarListener(
            topics = "topic-with-dlp",
            subscriptionType = SubscriptionType.Shared,
            deadLetterPolicy = "myDeadLetterPolicy",
            consumerCustomizer = "ackTimeoutCustomizer" )
    void listen(String msg) {
        throw new RuntimeException("fail " + msg);
    }

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

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

    @Bean
    ReactiveMessageConsumerBuilderCustomizer<String> ackTimeoutCustomizer() {
        return b -> b.property("ackTimeout", "1s");
    }
}
```

First, we have a special bean for `DeadLetterPolicy`, and it is named as `deadLetterPolicy` (it can 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`, in this case. If you do not specify a DLQ topic name, it defaults to `<topicname>-<subscriptionname>-DLQ` in Pulsar. Next, we provide this bean name to `ReactivePulsarListener` by setting the `deadLetterPolicy` property. Note that the `ReactivePulsarListener` 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 `ackTimeout` value of 1 second. The idea is that the code throws the exception and, if Pulsar does not receive an ack within 1 second, it does a retry. If that cycle continues ten times (as that is our max redelivery count in the `DeadLetterPolicy`), the Pulsar consumer publishes the messages to the DLQ topic. We have another `ReactivePulsarListener` that listens 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, 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), Pulsar publishes to a default topic name that has this `partition-<n>` info in it — for example: `topic-with-dlp-partition-0-deadLetterPolicySubscription-DLQ`. The easy way to solve this is to provide a DLQ topic name always.

Pulsar Reader Support

The framework provides support for using Pulsar Reader in a Reactive fashion via the `ReactivePulsarReaderFactory`.

Spring Boot provides this reader factory which can be configured with any of the `spring.pulsar.reactive.reader.*` application properties.

2.2.7. Topic Resolution

A destination topic is needed when producing or consuming messages. The framework looks in the following ordered locations to determine a topic (stopping at the first find):

- User specified
- Message type default
- Global default

When a topic is found via one of the default mechanisms, there is no need to specify the topic on the produce or consume API.

When a topic is not found, the API will throw an exception accordingly.

User specified

A topic passed into the API being used has the highest precedence (eg. `PulsarTemplate.send("my-topic", myMessage)` or `@PulsarListener(topics = "my-topic")`).

Message type default

When no topic is passed into the API, the system looks for a message type to topic mapping configured for the type of the message being produced or consumed.

Mappings can be configured with the `spring.pulsar.defaults.type-mappings` property. The following example uses `application.yml` to configure default topics to use when consuming or producing `Foo` or `Bar` messages:

```
spring:
  pulsar:
    defaults:
      type-mappings:
        - message-type: com.acme.Foo
          topic-name: foo-topic
        - message-type: com.acme.Bar
          topic-name: bar-topic
```

The `message-type` is the fully-qualified name of the message class.

If the message (or the first message of a `Publisher` input) is `null`, the framework won't be able to determine the topic from it. Another method shall be used to specify the topic if your application is likely to send `null` messages.

Custom topic resolver

The preferred method of adding mappings is via the property mentioned above. However, if more control is needed you can replace the default resolver by proving your own implementation, for example:

```
@Bean
public MyTopicResolver topicResolver() {
    return new MyTopicResolver();
}
```

Producer global default

The final location consulted (when producing) is the system-wide producer default topic. It is configured via the `spring.pulsar.producer.topic-name` property when using the imperative API and the `spring.pulsar.reactive.sender.topic-name` property when using the reactive API.

Consumer global default

The final location consulted (when consuming) is the system-wide consumer default topic. It is configured via the `spring.pulsar.consumer.topics` or `spring.pulsar.consumer.topics-pattern` property when using the imperative API and one of the `spring.pulsar.reactive.consumer.topics` or `spring.pulsar.reactive.consumer.topics-pattern` property when using the reactive API.

2.3. Pulsar Administration

2.3.1. Pulsar Admin Client

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 you use the Pulsar Spring Boot starter, you get the `PulsarAdministration` auto-configured.

By default, the application tries to connect to a local Pulsar instance at `http://localhost:8080`. This can be adjusted by setting the `spring.pulsar.administration.service-url` property to a different value in the form `(http|https)://<host>:<port>`.

There are many application properties available to configure the client. See the `spring.pulsar.administration.*` application properties.

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 replacing `spring.pulsar.client` with `spring.pulsar.administration`.

2.3.2. Automatic Topic Creation

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

The following example shows 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();
}
```

2.4. Pulsar Functions

Spring for Apache Pulsar provides basic suppport for Pulsar IO (connectors) and Pulsar Functions which allow users to define stream processing pipelines made up of `sources`, `processors`, and `sinks`. The `sources` and `sinks` are modeled by *Pulsar IO (connectors)* and the `processors` are represented by *Pulsar Functions*.

Because connectors are just special functions, and for simplicity, we refer to sources, sinks and functions collectively as "Pulsar Functions".

Pre-requisites

Familiarity - the audience is expected to be somewhat familiar w/ *Pulsar IO* and *Pulsar Functions*. If that is not the case it may be helpful to see their getting started guides.

Feature enabled - to use these features the functions support in Apache Pulsar must be enabled and configured (it is disabled by default). The built-in connectors may also need to be installed on the Pulsar cluster.

See the Pulsar IO and Pulsar Functions docs for more details.

2.4.1. Pulsar Function Administration

The framework provides the `PulsarFunctionAdministration` component to manage Pulsar functions. When you use the Pulsar Spring Boot starter, you get the `PulsarFunctionAdministration` auto-configured.

By default, the application tries to connect to a local Pulsar instance at `localhost:8080`. However, because it leverages the already configured `PulsarAdministration`, see Pulsar Admin Client for available client options (including authentication). Additional configuration options are available with the `spring.pulsar.function.*` application properties.

2.4.2. Automatic Function Management

On application startup, the framework finds all `PulsarFunction`, `PulsarSink`, and `PulsarSource` beans in the application context. For each bean, the corresponding Pulsar function is either created or updated. The proper API is called based on function type, function config, and whether the function already exists.

The `PulsarFunction`, `PulsarSink`, and `PulsarSource` beans are simple wrappers around the Apache Pulsar config objects `FunctionConfig`, `SinkConfig`, and `SourceConfig`, respectively. Due to the large number of supported connectors (and their varied configurations) the framework does not attempt to create a configuration properties hierarchy to mirror the varied Apache Pulsar connectors. Instead, the burden is on the user to supply the full config object and then the framework handles the management (create/update) using the supplied config.

On application shutdown, all functions that were processed during application startup have their stop policy enforced and are either left alone, stopped, or deleted from the Pulsar server.

2.4.3. Limitations

No Magic Pulsar Functions

Pulsar functions and custom connectors are represented by custom application code (eg. a `java.util.Function`). There is no magic support to automatically register the custom code. While this would be amazing, it has some technical challenges and not yet been implemented. As such, it is up to the user to ensure the function (or custom connector) is available at the location specified in the function config. For example, if the function config has a `jar` value of `./some/path/MyFunction.jar` then the function jar file must exist at the specified path.

Name Identifier

The `name` property from the function config is used as the identifier to determine if a function already exists in order to decide if an update or create operation is performed. As such, the name should not be modified if function updates are desired.

2.4.4. Configuration

Pulsar Function Archive

Each Pulsar function is represented by an actual archive (eg. jar file). The path to the archive is specified via the `archive` property for sources and sinks, and the `jar` property for functions.

The following rules determine the "type" of path:

- The path is a **URL** when it starts w/ `(file|http|https|function|sink|source)://`
- The path is **built-in** when it starts w/ `builtin://` (points to one of the provided out-of-the-box connectors)
- The path is **local** otherwise.

The action that occurs during the create/update operation is dependent on path "type" as follows:

- When the path is a **URL** the content is downloaded by the server
- When the path is **built-in** the content is already available on the server
- When the path is **local** the content is uploaded to the server

Built-in Source and Sinks

Apache Pulsar provides many source and sink connectors out-of-the-box, aka built-in connectors. To use a built-in connector simply set the `archive` to `builtin://<connector-type>` (eg `builtin://rabbit`).

2.4.5. Custom functions

The details on how to develop and package custom functions can be found in the Pulsar docs. However, at a high-level, the requirements are as follows:

- Code uses Java8
- Code implements either `java.util.Function` or `org.apache.pulsar.functions.api.Function`

- Packaged as uber jar

Once the function is built and packaged, there are several ways to make it available for function registration.

file://

The jar file can be uploaded to the server and then referenced via `file://` in the `jar` property of the function config

local

The jar file can remain local and then referenced via the local path in the `jar` property of the function config.

http://

The jar file can be made available via HTTP server and then referenced via `http(s)://` in the `jar` property of the function config

function://

The jar file can be uploaded to the Pulsar package manager and then referenced via `function://` in the `jar` property of the function config

2.4.6. Examples

Here are some examples that show how to configure a `PulsarSource` bean which results in the `PulsarFunctionAdministration` auto-creating the backing Pulsar source connector.

Example 1. PulsarSource using built-in Rabbit connector

```
@Bean
PulsarSource rabbitSource() {
    Map<String, Object> configs = new HashMap<>();
    configs.put("host", "my.rabbit.host");
    configs.put("port", 5672);
    configs.put("virtualHost", "/");
    configs.put("username", "guest");
    configs.put("password", "guest");
    configs.put("queueName", "test_rabbit");
    configs.put("connectionName", "test-connection");
    SourceConfig sourceConfig = SourceConfig.builder()
            .tenant("public")
            .namespace("default")
            .name("rabbit-test-source")
            .archive("builtin://rabbitmq")
            .topicName("incoming_rabbit")
            .configs(configs).build();
    return new PulsarSource(sourceConfig, null);
}
```

This next example is the same as the previous, except that it uses the Spring Boot auto-configured `RabbitProperties` to ease the configuration burden. This of course requires the application to be using Spring Boot with Rabbit auto-configuration enabled.

Example 2. PulsarSource using built-in Rabbit connector and Spring Boot RabbitProperties

```
@Bean
PulsarSource rabbitSourceWithBootProps(RabbitProperties props) {
    Map<String, Object> configs = new HashMap<>();
    configs.put("host", props.determineHost());
    configs.put("port", props.determinePort());
    configs.put("virtualHost", props.determineVirtualHost());
    configs.put("username", props.determineUsername());
    configs.put("password", props.determinePassword());
    configs.put("queueName", "test_rabbit");
    configs.put("connectionName", "test-connection");
    SourceConfig sourceConfig = SourceConfig.builder()
            .tenant("public")
            .namespace("default")
            .name("rabbit-test-source")
            .archive("builtin://rabbitmq")
            .topicName("incoming_rabbit")
            .configs(configs).build();
    return new PulsarSource(sourceConfig, null);
}
```

For a more elaborate example see the Sample Stream Pipeline with Pulsar Functions sample app

2.5. Observability

Spring for Apache Pulsar includes a way to manage observability through Micrometer.

Observability has not been added to the Reactive components yet

2.5.1. Micrometer Observations

The `PulsarTemplate` and `PulsarListener` are instrumented with the Micrometer observations API. When a Micrometer `ObservationRegistry` bean is provided, send and receive operations are traced and timed.

Custom tags

The default implementation adds the `bean.name` tag for template observations and `listener.id` tag for listener observations. To add other tags to timers and traces, configure a custom `PulsarTemplateObservationConvention` or `PulsarListenerObservationConvention` to the template or listener container, respectively.

You can subclass either `DefaultPulsarTemplateObservationConvention` or `DefaultPulsarListenerObservationConvention` or provide completely new implementations.

Observability - Metrics

Below you can find a list of all metrics declared by this project.

Listener Observation

> Observation created when a Pulsar listener receives a message.

Metric name `spring.pulsar.listener` (defined by convention class `org.springframework.pulsar.observation.DefaultPulsarListenerObservationConvention`). **Type** `timer`.

Metric name `spring.pulsar.listener.active` (defined by convention class `org.springframework.pulsar.observation.DefaultPulsarListenerObservationConvention`). **Type** `long task timer`.

KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Micrometer internally uses `nanoseconds` for the baseunit. However, each backend determines the actual baseunit. (i.e. Prometheus uses seconds)

Fully qualified name of the enclosing class `org.springframework.pulsar.observation.PulsarListenerObservation`.

All tags must be prefixed with `spring.pulsar.listener` prefix!

Table 1. Low cardinality Keys

Name	Description
`spring.pulsar.listener.id` *(required)*	Id of the listener container that received the message.

Template Observation

> Observation created when a Pulsar template sends a message.

Metric name `spring.pulsar.template` (defined by convention class `org.springframework.pulsar.observation.DefaultPulsarTemplateObservationConvention`). **Type** `timer`.

Metric name `spring.pulsar.template.active` (defined by convention class `org.springframework.pulsar.observation.DefaultPulsarTemplateObservationConvention`). **Type** `long task timer`.

KeyValues that are added after starting the Observation might be missing from the *.active metrics.

Micrometer internally uses `nanoseconds` for the baseunit. However, each backend determines the actual baseunit. (i.e. Prometheus uses seconds)

Fully qualified name of the enclosing class `org.springframework.pulsar.observation.PulsarTemplateObservation`.

All tags must be prefixed with `spring.pulsar.template` prefix!

Table 2. Low cardinality Keys

Name	Description
`spring.pulsar.template.name` *(required)*	Bean name of the template that sent the message.

Observability - Spans

Below you can find a list of all spans declared by this project.

Listener Observation Span

> Observation created when a Pulsar listener receives a message.

Span name `spring.pulsar.listener` (defined by convention class `org.springframework.pulsar.observation.DefaultPulsarListenerObservationConvention`).

Fully qualified name of the enclosing class `org.springframework.pulsar.observation.PulsarListenerObservation`.

All tags must be prefixed with `spring.pulsar.listener` prefix!

Table 3. Tag Keys

Name	Description
`spring.pulsar.listener.id` *(required)*	Id of the listener container that received the message.

Template Observation Span

> Observation created when a Pulsar template sends a message.

Span name `spring.pulsar.template` (defined by convention class `org.springframework.pulsar.observation.DefaultPulsarTemplateObservationConvention`).

Fully qualified name of the enclosing class

`org.springframework.pulsar.observation.PulsarTemplateObservation`.

All tags must be prefixed with `spring.pulsar.template` prefix!

Table 4. Tag Keys

Name	Description
`spring.pulsar.template.name` *(required)*	Bean name of the template that sent the message.

See Micrometer Tracing for more information.

Manual Configuration without Spring Boot

If you do not use Spring Boot, you need to configure and provide an `ObservationRegistry` as well as Micrometer Tracing. See Micrometer Tracing for more information.

Auto-Configuration with Spring Boot

If you use Spring Boot, the Spring Boot Actuator auto-configures an instance of `ObservationRegistry` for you. If `micrometer-core` is on the classpath, every stopped observation leads to a timer.

Spring Boot also auto-configures Micrometer Tracing for you. This includes support for Brave OpenTelemetry, Zipkin, and Wavefront. When using the Micrometer Observation API, finishing observations leads to spans reported to Zipkin or Wavefront. You can control tracing by setting properties under `management.tracing`. You can use Zipkin with `management.zipkin.tracing`, while Wavefront uses `management.wavefront`.

Example Configuration

The following example shows the steps to configure your Spring Boot application to use Zipkin with Brave.

1. Add the required dependencies to your application (in Maven or Gradle, respectively):

Maven

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <dependency>
        <groupId>io.micrometer</groupId>
        <artifactId>micrometer-tracing-bridge-brave</artifactId>
    </dependency>
    <dependency>
        <groupId>io.zipkin.reporter2</groupId>
        <artifactId>zipkin-reporter-brave</artifactId>
    </dependency>
    <dependency>
        <groupId>io.zipkin.reporter2</groupId>
        <artifactId>zipkin-sender-urlconnection</artifactId>
    </dependency>
</dependencies>
```

Gradle

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-actuator'
    implementation 'io.micrometer:micrometer-tracing-bridge-brave'
    implementation 'io.zipkin.reporter2:zipkin-reporter-brave'
    implementation 'io.zipkin.reporter2:zipkin-sender-urlconnection'
}
```

NOTE

You need the `'io.zipkin.reporter2:zipkin-sender-urlconnection'` dependency only if your application does not have a configured WebClient or RestTemplate.

1. Add the required properties to your application:

```
management:
  tracing.enabled: true
  zipkin:
    tracing.endpoint: "http://localhost:9411/api/v2/spans"
```

The `tracing.endpoint` above expects Zipkin is running locally as described here.

At this point, your application should record traces when you send and receive Pulsar messages. You should be able to view them in the Zipkin UI (at localhost:9411, when running locally).

You can also see the preceding configuration on the Spring Pulsar Sample Apps.

The steps are very similar to configuring any of the other supported Tracing environments.

2.6. Spring Cloud Stream Binder for Apache Pulsar

Spring for Apache Pulsar provides a binder for Spring Cloud Stream that we can use to build event-driven microservices using pub-sub paradigms. In this section, we will go through the basic details of this binder.

For those unfamiliar with the concepts in Spring Cloud Stream, go through the primary reference docs for Spring Cloud Stream to understand the core concepts.

2.6.1. Usage

We need to include the following dependency on your application to use Apache Pulsar binder for Spring Cloud Stream.

Maven

```
<dependencies>
    <dependency>
        <groupId>org.springframework.pulsar</groupId>
        <artifactId>spring-pulsar-spring-cloud-stream-binder</artifactId>
        <version>0.2.1-SNAPSHOT</version>
    </dependency>
</dependencies>
```

Gradle

```
dependencies {
    implementation 'org.springframework.pulsar:spring-pulsar-spring-cloud-stream-
binder:0.2.1-SNAPSHOT'
}
```

2.6.2. Overview

The Spring Cloud Stream binder for Apache Pulsar allows the applications to focus on business logic rather than dealing with the lower-level details of managing and maintaining Pulsar. The binder takes care of all those details for the application developer. Spring Cloud Stream brings a powerful programming model based on Spring Cloud Function that allows the app developer to write complex event-driven applications using a functional style. Applications can start from a middleware-neutral manner and then map Pulsar topics as destinations in Spring Cloud Stream through Spring Boot configuration properties. Spring Cloud Stream is built on top of Spring Boot, and when writing an event-driven microservice using Spring Cloud Stream, you are essentially writing a Boot application. Here is a straightforward Spring Cloud Stream application.

```
@SpringBootApplication
public class SpringPulsarBinderSampleApp {

    private final Logger logger = LoggerFactory.getLogger(this.getClass());

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

    @Bean
    public Supplier<Time> timeSupplier() {
        return () -> new Time(String.valueOf(System.currentTimeMillis()));
    }

    @Bean
    public Function<Time, EnhancedTime> timeProcessor() {
        return (time) -> {
            EnhancedTime enhancedTime = new EnhancedTime(time, "5150");
            this.logger.info("PROCESSOR: {} --> {}", time, enhancedTime);
            return enhancedTime;
        };
    }

    @Bean
    public Consumer<EnhancedTime> timeLogger() {
        return (time) -> this.logger.info("SINK:      {}", time);
    }

    record Time(String time) {
    }

    record EnhancedTime(Time time, String extra) {
    }

}
```

The above sample application, a full-blown Spring Boot application, deserves a few explanations. However, on a first pass, you can see that this is just plain Java and a few Spring and Spring Boot annotations. We have three `Bean` methods here - a `java.util.function.Supplier`, a `java.util.function.Function`, and finally, a `java.util.function.Consumer`. The supplier produces the current time in milliseconds, the function takes this time and then enhances it by adding some random data, and then the consumer logs the enhanced time.

We omitted all the imports for brevity, but nothing Spring Cloud Stream specific in the entire application. How does it become a Spring Cloud Stream application that interacts with Apache Pulsar? You must include the above dependency for the binder in the application. Once that dependency is added, you must provide the following configuration properties.

```
spring:
  cloud:
    function:
      definition: timeSupplier;timeProcessor;timeLogger;
    stream:
      bindings:
        timeProcessor-in-0:
          destination: timeSupplier-out-0
        timeProcessor-out-0:
          destination: timeProcessor-out-0
        timeLogger-in-0:
          destination: timeProcessor-out-0
```

With this, the above Spring Boot application has become an end-to-end event-driven application based on Spring Cloud Stream. Because we have the Pulsar binder on the classpath, the application interacts with Apache Pulsar. If there is only one function in the application, then we don't need to tell Spring Cloud Stream to activate the function for execution since it does that by default. If there is more than one such function in the application, as in our example, we need to instruct Spring Cloud Stream which functions we would like to activate. In our case, we need all of them to be activated, and we do that through the `spring.cloud.function.definition` property. The bean name becomes part of the Spring Cloud Stream binding name by default. A binding is a fundamentally abstract concept in Spring Cloud Stream, using which the framework communicates with the middleware destination. Almost everything that Spring Cloud Stream does occurs over a concrete binding. A supplier has only an output binding; functions have input and output bindings, and consumers have only input binding. Let's take as an example our supplier bean - `timeSupplier.` The default binding name for this supplier will be `timeSupplier-out-0`. Similarly, the default binding names for the `timeProcessor` function will be `timeProcessor-in-0` on the inbound and `timeProcessor-out-0` on the outbound. Please refer to the Spring Cloud Stream reference docs for details on how you can change the default binding names. In most situations, using the default binding names is enough. We set the destination on the binding names, as shown above. If a destination is not provided, the binding name becomes the value for the destination as in the case of `timeSupplier-out-0`.

When running the above app, you should see that the supplier executes every second, which is then consumed by the function and enhances the time consumed by the logger consumer.

2.6.3. Message Conversion in Binder-based Applications

In the above sample application, we provided no schema information for message conversion. That is because, by default, Spring Cloud Stream uses its message conversion mechanism using the messaging support established in Spring Framework through the Spring Messaging project. Unless specified, Spring Cloud Stream uses `application/json` as the `content-type` for message conversion on both inbound and outbound bindings. On the outbound, the data is serialized as `byte[],` and the Pulsar binder then uses `Schema.BYTES` to send it over the wire to the Pulsar topic. Similarly, on the inbound, the data is consumed as `byte[]` from the Pulsar topic and then converted into the target type using the proper message converter.

Using Native Conversion in Pulsar using Pulsar Schema

Although the default is to use the framework-provided message conversion, Spring Cloud Stream allows each binder to determine how the message should be converted. Suppose the application chooses to go this route. In that case, Spring Cloud Stream steers clear of using any Spring-provided message conversion facility and passes around the data it receives or produces. This feature in Spring Cloud Stream is known as native encoding on the producer side and native decoding on the consumer side. This means that the encoding and decoding natively occur on the target middleware, in our case, on Apache Pulsar. For the above application, we can use the following configuration to bypass the framework conversion and uses native encoding and decoding.

```
spring:
  cloud:
    stream:
      bindings:
        timeSupplier-out-0:
          producer:
            use-native-encoding: true
        timeProcessor-in-0:
          destination: timeSupplier-out-0
          consumer:
            use-native-decoding: true
        timeProcessor-out-0:
          destination: timeProcessor-out-0
          producer:
            use-native-encoding: true
        timeLogger-in-0:
          destination: timeProcessor-out-0
          consumer:
            use-native-decoding: true
      pulsar:
        bindings:
          timeSupplier-out-0:
            producer:
              schema-type: JSON
              message-type:
org.springframework.pulsar.sample.binder.SpringPulsarBinderSampleApp.Time
          timeProcessor-in-0:
            consumer:
              schema-type: JSON
              message-type:
org.springframework.pulsar.sample.binder.SpringPulsarBinderSampleApp.Time
          timeProcessor-out-0:
            producer:
              schema-type: AVRO
              message-type:
org.springframework.pulsar.sample.binder.SpringPulsarBinderSampleApp.EnhancedTime
          timeLogger-in-0:
            consumer:
              schema-type: AVRO
              message-type:
org.springframework.pulsar.sample.binder.SpringPulsarBinderSampleApp.EnhancedTime
```

The property to enable native encoding on the producer side is a binding level property from the core Spring Cloud Stream. You set it on the producer binding - `spring.cloud.stream.bindings.<binding-name>.producer.use-native-encoding` and set this to `true`. Similarly, use - `spring.cloud.stream.bindings.<binding-name>.consumer.user-native-decoding` for consumer bindings and set it to `true`. If we decide to use native encoding and decoding, in the case of Pulsar, we need to set the corresponding schema and the underlying message type information. This information is provided as extended binding properties. As you can see above in the

configuration, the properties are - `spring.cloud.stream.pulsar.bindings.<binding-name>.producer|consumer.schema-type` for schema information and `spring.cloud.stream.pulsar.bindings.<binding-name>.producer|consumer.message-type` for the actual target type. If you have both keys and values on the message, you can use `message-key-type` and `message-value-type` to specify their target types.

Any configured custom schema mappings will be consulted when the `schema-type` property is omitted.

Message Header Conversion

Each message typically has header information that needs to be carried along as the message traverses between Pulsar and Spring Messaging via Spring Cloud Stream input and output bindings. To support this traversal, the framework handles the necessary message header conversion.

Pulsar Headers

Pulsar does not have a first-class "header" concept but instead provides a map for custom user properties as well as methods to access the message metadata typically stored in a message header (eg. `id` and `event-time`). As such, the terms "Pulsar message header" and "Pulsar message metadata" are used interchangeably. The list of available message metadata (headers) can be found in PulsarHeaders.java.

Spring Headers

Spring Messaging provides first-class "header" support via its `MessageHeaders` abstraction.

Message Header Mapping

The `PulsarHeaderMapper` strategy is provided to map headers to and from Pulsar user properties and Spring `MessageHeaders`.

Its interface definition is as follows:

```
public interface PulsarHeaderMapper {

    Map<String, String> toPulsarHeaders(MessageHeaders springHeaders);

    MessageHeaders toSpringHeaders(Message<?> pulsarMessage);
}
```

The framework provides a couple of mapper implementations.

- The `JsonPulsarHeaderMapper` maps headers as JSON in order to support rich header types and is the default when the Jackson JSON library is on the classpath.
- The `ToStringPulsarHeaderMapper` maps headers as strings using the `toString()` method on the header values and is the fallback mapper.

JSON Header Mapper

The `JsonPulsarHeaderMapper` uses a "special" header (with a key of `spring_json_header_types`) that contains a JSON map of `<key>:<type>`. This header is used on the inbound side (Pulsar → Spring) to provide appropriate conversion of each header value to the original type.

Trusted Packages

By default, the JSON mapper deserializes classes in all packages. However, if you receive messages from untrusted sources, you may wish to add only those packages you trust via the `trustedPackages` property on a custom configured `JsonPulsarHeaderMapper` bean you provide.

ToString Classes

Certain types are not suitable for JSON serialization, and a simple `toString()` serialization might be preferred for these types. The `JsonPulsarHeaderMapper` has a property called `addToStringClasses()` that lets you supply the names of classes that should be treated this way for outbound mapping. During inbound mapping, they are mapped as `String`. By default, only `org.springframework.util.MimeType` and `org.springframework.http.MediaType` are mapped this way.

Inbound/Outbound Patterns

On the inbound side, by default, all Pulsar headers (message metadata plus user properties) are mapped to `MessageHeaders`. On the outbound side, by default, all `MessageHeaders` are mapped, except `id`, `timestamp`, and the headers that represent the Pulsar message metadata. You can specify which headers are mapped for inbound and outbound messages by configuring the `inboundPatterns` and `outboundPatterns` on a mapper bean you provide.

Patterns are rather simple and can contain a leading wildcard (`*`), a trailing wildcard, or both (for example, `*.cat.*`). You can negate patterns with a leading `!`. The first pattern that matches a header name (whether positive or negative) wins.

When you provide your own patterns, we recommend including `!id` and `!timestamp`, since these headers are read-only on the inbound side.

Custom Header Mapper

The Pulsar binder is configured with a default header mapper that can be overridden by providing your own `PulsarHeaderMapper` bean.

In the following example, a JSON header mapper is configured that:

- maps all inbound headers (except those with keys "top" or "secret")
- maps outbound headers (except those with keys "id", "timestamp", or "userId")
- only trusts objects in the "com.acme" package for outbound deserialization
- de/serializes any "com.acme.Money" header values w/ simple `toString()` encoding

```
@Bean
public PulsarHeaderMapper customPulsarHeaderMapper() {
    return JsonPulsarHeaderMapper.builder()
            .inboundPatterns("!top", "!secret", "*")
            .outboundPatterns("!id", "!timestamp", "!userId", "*")
            .trustedPackages("com.acme")
            .toStringClasses("com.acme.Money")
            .build();
}
```

2.6.4. Using Pulsar Properties in the Binder

The binder uses basic components from Spring for Apache Pulsar framework to build its producer and consumer bindings. Since binder-based applications are Spring Boot applications, binder, by default, uses the Spring Boot autoconfiguration for Spring for Apache Pulsar. Therefore, all Pulsar Spring Boot properties available at the core framework level are also available through the binder. For example, you can use properties with the prefix `spring.pulsar.producer⋯`, `spring.pulsar.consumer⋯` etc. In addition, you can also set these Pulsar properties at the binder level. For instance, this will also work - `spring.cloud.stream.pulsar.binder.producer⋯` or `spring.cloud.stream.pulsar.binder.consumer⋯`.

Either of the above approaches is fine, but when using properties like these, it is applied to the whole application. If you have multiple functions in the application, they all get the same properties. You can also set these Pulsar properties at the extended binding properties level to address this. Extended binding properties are applied at the binding itself. For instance, if you have an input and output binding, and both require a separate set of Pulsar properties, you must set them on the extended binding. The pattern for producer binding is `spring.cloud.stream.pulsar.bindings.<output-binding-name>.producer⋯`. Similarly, for consumer binding, the pattern is `spring.cloud.stream.pulsar.bindings.<input-binding-name>.consumer⋯`. This way, you can have a separate set of Pulsar properties applied for different bindings in the same application.

The highest precedence is for extended binding properties. The precedence order of applying the properties in the binder is `extended binding properties → binder properties → Spring Boot properties.` (going from highest to lowest).

2.6.5. Pulsar Topic Provisioner

Spring Cloud Stream binder for Apache Pulsar comes with an out-of-the-box provisioner for Pulsar topics. When running an application, if the necessary topics are absent, Pulsar will create the topics for you. However, this is a basic non-partitioned topic, and if you want advanced features like creating a partitioned topic, you can rely on the topic provisioner in the binder. Pulsar topic provisioner uses `PulsarAdministration` from the framework, which uses the `PulsarAdminBuilder.` For this reason, you need to set the `spring.pulsar.administration.service-url` property unless you are running Pulsar on the default server and port.

Specifying partition count when creating the topic

When creating the topic, you can set the partition count in two ways. First, you can set it at the binder level using the property `spring.cloud.stream.pulsar.binder.partition-count`. As we saw above, doing this way will make all the topics created by the application inherit this property. Suppose you want granular control at the binding level for setting partitions. In that case, you can set the `partition-count` property per binding using the format `spring.cloud.stream.pulsar.bindings.<binding-name>.producer|consumer.partition-count`. This way, various topics created by different functions in the same application will have different partitions based on the application requirements.

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.

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

Appendices

Appendix A: Pulsar Clients and Spring Boot Compatibility

The following is the compatibility matrix:

Spring for Apache Pulsar	Pulsar Client	Pulsar Reactive Client	Spring Boot	Java
1.0.x	3.0.x	0.3.x	3.2.x	17+
0.2.x	2.11.x	0.2.x	3.0.x / 3.1.x(*)	17+

In version `1.0.0` the auto-configuration moved into Spring Boot `3.2.x` and therefore `3.2.x` is the minimum Spring Boot version supported when using version `1.0.x` of the framework.

However, prior to version `1.0.0`, the auto-configuration support exists in the framework itself. (*)This makes it theoretically possible to use later versions of Spring Boot besides `3.0.x` which it is tested against and guaranteed to work with. In other words, it may work with `3.1.x` but it has not been tested against it.

Appendix B: Override Spring Boot Dependencies

When using Spring for Apache Pulsar in a Spring Boot application, the Apache Pulsar dependency versions are determined by Spring Boot's dependency management. If you wish to use a different version of `pulsar-client-all` or `pulsar-client-reactive-adapter`, you need to override their version used by Spring Boot dependency management; set the `pulsar.version` or `pulsar-reactive.version` property, respectively.

Or, to use a different Spring for Apache Pulsar version with a supported Spring Boot version, set the `spring-pulsar.version` property.

In the following example, snapshot version of the Pulsar clients and Spring Pulsar are being used.

Gradle

```
ext['pulsar.version'] = '3.0.1-SNAPSHOT'
ext['pulsar-reactive.version'] = '0.3.1-SNAPSHOT'
ext['spring-pulsar.version'] = '1.0.1-SNAPSHOT'

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

Maven

```
<properties>
    <pulsar.version>3.0.1-SNAPSHOT</pulsar.version>
    <pulsar-reactive.version>0.3.1-SNAPSHOT</pulsar-reactive.version>
    <spring-pulsar.version>1.0.1-SNAPSHOT</spring-pulsar.version>
</properties>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-pulsar-reactive</artifactId>
</dependency>
```

Appendix C: Non-GA Versions

You can find snapshot or milestone versions of the dependencies in the following repositories:

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>
    <repository>
        <id>apache-snapshots</id>
        <name>Apache Snapshots</name>
        <url>https://repository.apache.org/content/repositories/snapshots</url>
        <releases>
            <enabled>false</enabled>
        </releases>
    </repository>
</repositories>
```

Gradle

```
repositories {
    maven {
        name = 'spring-milestones'
        url = 'https://repo.spring.io/milestone'
    }
    maven {
        name = 'spring-snapshots'
        url = 'https://repo.spring.io/snapshot'
    }
    maven {
        name = 'apache-snapshot'
        url = 'https://repository.apache.org/content/repositories/snapshots'
    }
}
```

Appendix D: GraalVM Native Image Support

GraalVM Native Images are standalone executables that can be generated by processing compiled

Java applications ahead-of-time. Native Images generally have a smaller memory footprint and start faster than their JVM counterparts.

Support

The required AOT Runtime Hints are built-in to Spring for Apache Pulsar so that it can seamlessly be used in native image based Spring applications.

The native image support in Spring for Apache Pulsar has been tested in basic scenarios, and we expect it to "just work". However, it is possible that more advanced use cases could surface the need to add additional runtime hints to your own application. If this occurs please file a Github issue with some details.

Next Steps

If you are interested in adding native image support to your own application then an excellent place to start is the Spring Boot GraalVM Support section of the Spring Boot reference docs.

Although there is no reference to Spring for Apache Pulsar in the aforementioned guide, you can find specific examples at the following coordinates:

- Spring for Apache Pulsar (imperative)
- Spring for Apache Pulsar (reactive)