using `@EnableElasticsearchRepositories` and `@EntityScan` respectively.

> **TIP** For complete details of Spring Data Elasticsearch, see the reference documentation.

Spring Boot supports both classic and reactive Elasticsearch repositories, using the `ElasticsearchRestTemplate` or `ReactiveElasticsearchTemplate` beans. Most likely those beans are auto-configured by Spring Boot given the required dependencies are present.

If you wish to use your own template for backing the Elasticsearch repositories, you can add your own `ElasticsearchRestTemplate` or `ElasticsearchOperations` `@Bean`, as long as it is named `"elasticsearchTemplate"`. Same applies to `ReactiveElasticsearchTemplate` and `ReactiveElasticsearchOperations`, with the bean name `"reactiveElasticsearchTemplate"`.

You can choose to disable the repositories support with the following property:

Properties

```
spring.data.elasticsearch.repositories.enabled=false
```

Yaml

```
spring:
  data:
    elasticsearch:
      repositories:
        enabled: false
```

9.2.5. Cassandra

Cassandra is an open source, distributed database management system designed to handle large amounts of data across many commodity servers. Spring Boot offers auto-configuration for Cassandra and the abstractions on top of it provided by Spring Data Cassandra. There is a `spring-boot-starter-data-cassandra` “Starter” for collecting the dependencies in a convenient way.

Connecting to Cassandra

You can inject an auto-configured `CassandraTemplate` or a Cassandra `CqlSession` instance as you would with any other Spring Bean. The `spring.cassandra.*` properties can be used to customize the connection. Generally, you provide `keyspace-name` and `contact-points` as well the local datacenter name, as shown in the following example:

Properties

```
spring.cassandra.keyspace-name=mykeyspace
spring.cassandra.contact-points=cassandrahost1:9042,cassandrahost2:9042
spring.cassandra.local-datacenter=datacenter1
```

Yaml

```
spring:
  cassandra:
    keyspace-name: "mykeyspace"
    contact-points: "cassandrahost1:9042,cassandrahost2:9042"
    local-datacenter: "datacenter1"
```

If the port is the same for all your contact points you can use a shortcut and only specify the host names, as shown in the following example:

Properties

```
spring.cassandra.keyspace-name=mykeyspace
spring.cassandra.contact-points=cassandrahost1,cassandrahost2
spring.cassandra.local-datacenter=datacenter1
```

Yaml

```
spring:
  cassandra:
    keyspace-name: "mykeyspace"
    contact-points: "cassandrahost1,cassandrahost2"
    local-datacenter: "datacenter1"
```

TIP Those two examples are identical as the port default to `9042`. If you need to configure the port, use `spring.cassandra.port`.

The auto-configured `CqlSession` can be configured to use SSL for communication with the server by setting the properties as shown in this example:

Properties

```
spring.cassandra.keyspace-name=mykeyspace
spring.cassandra.contact-points=cassandrahost1,cassandrahost2
spring.cassandra.local-datacenter=datacenter1
spring.cassandra.ssl.enabled=true
```

Yaml

```
spring:
  cassandra:
    keyspace-name: "mykeyspace"
    contact-points: "cassandrahost1,cassandrahost2"
    local-datacenter: "datacenter1"
    ssl:
      enabled: true
```

Custom SSL trust material can be configured in an SSL bundle and applied to the `CqlSession` as shown in this example:

Properties

```
spring.cassandra.keyspace-name=mykeyspace
spring.cassandra.contact-points=cassandrahost1,cassandrahost2
spring.cassandra.local-datacenter=datacenter1
spring.cassandra.ssl.bundle=example
```

Yaml

```
spring:
  cassandra:
    keyspace-name: "mykeyspace"
    contact-points: "cassandrahost1,cassandrahost2"
    local-datacenter: "datacenter1"
    ssl:
      bundle: "example"
```

NOTE

The Cassandra driver has its own configuration infrastructure that loads an `application.conf` at the root of the classpath.

Spring Boot does not look for such a file by default but can load one using `spring.cassandra.config`. If a property is both present in `spring.cassandra.*` and the configuration file, the value in `spring.cassandra.*` takes precedence.

For more advanced driver customizations, you can register an arbitrary number of beans that implement `DriverConfigLoaderBuilderCustomizer`. The `CqlSession` can be customized with a bean of type `CqlSessionBuilderCustomizer`.

NOTE

If you use `CqlSessionBuilder` to create multiple `CqlSession` beans, keep in mind the builder is mutable so make sure to inject a fresh copy for each session.

The following code listing shows how to inject a Cassandra bean:

Java

```
import org.springframework.data.cassandra.core.CassandraTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final CassandraTemplate template;

    public MyBean(CassandraTemplate template) {
        this.template = template;
    }

    public long someMethod() {
        return this.template.count(User.class);
    }

}
```

Kotlin

```
import org.springframework.data.cassandra.core.CassandraTemplate
import org.springframework.stereotype.Component

@Component
class MyBean(private val template: CassandraTemplate) {

    fun someMethod(): Long {
        return template.count(User::class.java)
    }

}
```

If you add your own `@Bean` of type `CassandraTemplate`, it replaces the default.

Spring Data Cassandra Repositories

Spring Data includes basic repository support for Cassandra. Currently, this is more limited than the JPA repositories discussed earlier and needs `@Query` annotated finder methods.

Repositories and entities are found through scanning. By default, the auto-configuration packages are scanned. You can customize the locations to look for repositories and entities by using `@EnableCassandraRepositories` and `@EntityScan` respectively.

TIP For complete details of Spring Data Cassandra, see the reference documentation.

9.2.6. Couchbase

Couchbase is an open-source, distributed, multi-model NoSQL document-oriented database that is optimized for interactive applications. Spring Boot offers auto-configuration for Couchbase and the abstractions on top of it provided by Spring Data Couchbase. There are `spring-boot-starter-data-couchbase` and `spring-boot-starter-data-couchbase-reactive` "Starters" for collecting the dependencies in a convenient way.

Connecting to Couchbase

You can get a `Cluster` by adding the Couchbase SDK and some configuration. The `spring.couchbase.*` properties can be used to customize the connection. Generally, you provide the connection string, username, and password, as shown in the following example:

Properties

```
spring.couchbase.connection-string=couchbase://192.168.1.123
spring.couchbase.username=user
spring.couchbase.password=secret
```

Yaml

```
spring:
  couchbase:
    connection-string: "couchbase://192.168.1.123"
    username: "user"
    password: "secret"
```

It is also possible to customize some of the `ClusterEnvironment` settings. For instance, the following configuration changes the timeout to open a new `Bucket` and enables SSL support with a reference to a configured SSL bundle:

Properties

```
spring.couchbase.env.timeouts.connect=3s
spring.couchbase.env.ssl.bundle=example
```

Yaml

```
spring:
  couchbase:
    env:
      timeouts:
        connect: "3s"
      ssl:
        bundle: "example"
```

TIP	Check the `spring.couchbase.env.*` properties for more details. To take more control, one or more `ClusterEnvironmentBuilderCustomizer` beans can be used.

Spring Data Couchbase Repositories

Spring Data includes repository support for Couchbase.

Repositories and documents are found through scanning. By default, the auto-configuration packages are scanned. You can customize the locations to look for repositories and documents by using `@EnableCouchbaseRepositories` and `@EntityScan` respectively.

For complete details of Spring Data Couchbase, see the reference documentation.

You can inject an auto-configured `CouchbaseTemplate` instance as you would with any other Spring Bean, provided a `CouchbaseClientFactory` bean is available. This happens when a `Cluster` is available, as described above, and a bucket name has been specified:

Properties

```
spring.data.couchbase.bucket-name=my-bucket
```

Yaml

```
spring:
  data:
    couchbase:
      bucket-name: "my-bucket"
```

The following examples shows how to inject a `CouchbaseTemplate` bean:

Java

```
import org.springframework.data.couchbase.core.CouchbaseTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final CouchbaseTemplate template;

    public MyBean(CouchbaseTemplate template) {
        this.template = template;
    }

    public String someMethod() {
        return this.template.getBucketName();
    }

}
```

Kotlin

```
import org.springframework.data.couchbase.core.CouchbaseTemplate
import org.springframework.stereotype.Component

@Component
class MyBean(private val template: CouchbaseTemplate) {

    fun someMethod(): String {
        return template.bucketName
    }

}
```

There are a few beans that you can define in your own configuration to override those provided by the auto-configuration:

- A `CouchbaseMappingContext` `@Bean` with a name of `couchbaseMappingContext`.
- A `CustomConversions` `@Bean` with a name of `couchbaseCustomConversions`.
- A `CouchbaseTemplate` `@Bean` with a name of `couchbaseTemplate`.

To avoid hard-coding those names in your own config, you can reuse `BeanNames` provided by Spring Data Couchbase. For instance, you can customize the converters to use, as follows:

Java

```
import org.assertj.core.util.Arrays;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.couchbase.config.BeanNames;
import org.springframework.data.couchbase.core.convert.CouchbaseCustomConversions;

@Configuration(proxyBeanMethods = false)
public class MyCouchbaseConfiguration {

    @Bean(BeanNames.COUCHBASE_CUSTOM_CONVERSIONS)
    public CouchbaseCustomConversions myCustomConversions() {
        return new CouchbaseCustomConversions(Arrays.asList(new MyConverter()));
    }

}
```

Kotlin

```
import org.assertj.core.util.Arrays
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.data.couchbase.config.BeanNames
import org.springframework.data.couchbase.core.convert.CouchbaseCustomConversions

@Configuration(proxyBeanMethods = false)
class MyCouchbaseConfiguration {

    @Bean(BeanNames.COUCHBASE_CUSTOM_CONVERSIONS)
    fun myCustomConversions(): CouchbaseCustomConversions {
        return CouchbaseCustomConversions(Arrays.asList(MyConverter()))
    }

}
```

9.2.7. LDAP

LDAP (Lightweight Directory Access Protocol) is an open, vendor-neutral, industry standard application protocol for accessing and maintaining distributed directory information services over an IP network. Spring Boot offers auto-configuration for any compliant LDAP server as well as support for the embedded in-memory LDAP server from UnboundID.

LDAP abstractions are provided by Spring Data LDAP. There is a `spring-boot-starter-data-ldap` "Starter" for collecting the dependencies in a convenient way.

Connecting to an LDAP Server

To connect to an LDAP server, make sure you declare a dependency on the `spring-boot-starter-data-ldap` "Starter" or `spring-ldap-core` and then declare the URLs of your server in your application.properties, as shown in the following example:

Properties

```
spring.ldap.urls=ldap://myserver:1235
spring.ldap.username=admin
spring.ldap.password=secret
```

Yaml

```
spring:
  ldap:
    urls: "ldap://myserver:1235"
    username: "admin"
    password: "secret"
```

If you need to customize connection settings, you can use the `spring.ldap.base` and

`spring.ldap.base-environment` properties.

An `LdapContextSource` is auto-configured based on these settings. If a `DirContextAuthenticationStrategy` bean is available, it is associated to the auto-configured `LdapContextSource`. If you need to customize it, for instance to use a `PooledContextSource`, you can still inject the auto-configured `LdapContextSource`. Make sure to flag your customized `ContextSource` as `@Primary` so that the auto-configured `LdapTemplate` uses it.

Spring Data LDAP Repositories

Spring Data includes repository support for LDAP.

Repositories and documents are found through scanning. By default, the auto-configuration packages are scanned. You can customize the locations to look for repositories and documents by using `@EnableLdapRepositories` and `@EntityScan` respectively.

For complete details of Spring Data LDAP, see the reference documentation.

You can also inject an auto-configured `LdapTemplate` instance as you would with any other Spring Bean, as shown in the following example:

Java

```
import java.util.List;

import org.springframework.ldap.core.LdapTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final LdapTemplate template;

    public MyBean(LdapTemplate template) {
        this.template = template;
    }

    public List<User> someMethod() {
        return this.template.findAll(User.class);
    }

}
```

Kotlin

```
import org.springframework.ldap.core.LdapTemplate
import org.springframework.stereotype.Component

@Component
class MyBean(private val template: LdapTemplate) {

    fun someMethod(): List<User> {
        return template.findAll(User::class.java)
    }

}
```

Embedded In-memory LDAP Server

For testing purposes, Spring Boot supports auto-configuration of an in-memory LDAP server from UnboundID. To configure the server, add a dependency to `com.unboundid:unboundid-ldapsdk` and declare a `spring.ldap.embedded.base-dn` property, as follows:

Properties

```
spring.ldap.embedded.base-dn=dc=spring,dc=io
```

Yaml

```
spring:
  ldap:
    embedded:
      base-dn: "dc=spring,dc=io"
```

NOTE

It is possible to define multiple base-dn values, however, since distinguished names usually contain commas, they must be defined using the correct notation.

In yaml files, you can use the yaml list notation. In properties files, you must include the index as part of the property name:

Properties

```
spring.ldap.embedded.base-dn[0]=dc=spring,dc=io
spring.ldap.embedded.base-dn[1]=dc=vmware,dc=com
```

Yaml

```
spring.ldap.embedded.base-dn:
  - "dc=spring,dc=io"
  - "dc=vmware,dc=com"
```

By default, the server starts on a random port and triggers the regular LDAP support. There is no need to specify a `spring.ldap.urls` property.

If there is a `schema.ldif` file on your classpath, it is used to initialize the server. If you want to load the initialization script from a different resource, you can also use the `spring.ldap.embedded.ldif` property.

By default, a standard schema is used to validate `LDIF` files. You can turn off validation altogether by setting the `spring.ldap.embedded.validation.enabled` property. If you have custom attributes, you can use `spring.ldap.embedded.validation.schema` to define your custom attribute types or object classes.

9.2.8. InfluxDB

WARNING	Auto-configuration for InfluxDB is deprecated and scheduled for removal in Spring Boot 3.4 in favor of the new InfluxDB Java client that provides its own Spring Boot integration.

InfluxDB is an open-source time series database optimized for fast, high-availability storage and retrieval of time series data in fields such as operations monitoring, application metrics, Internet-of-Things sensor data, and real-time analytics.

Connecting to InfluxDB

Spring Boot auto-configures an `InfluxDB` instance, provided the `influxdb-java` client is on the classpath and the URL of the database is set using `spring.influx.url`.

If the connection to InfluxDB requires a user and password, you can set the `spring.influx.user` and `spring.influx.password` properties accordingly.

InfluxDB relies on OkHttp. If you need to tune the http client `InfluxDB` uses behind the scenes, you can register an `InfluxDbOkHttpClientBuilderProvider` bean.

If you need more control over the configuration, consider registering an `InfluxDbCustomizer` bean.

9.3. What to Read Next

You should now have a feeling for how to use Spring Boot with various data technologies. From here, you can read about Spring Boot's support for various messaging technologies and how to enable them in your application.

Chapter 10. Messaging

The Spring Framework provides extensive support for integrating with messaging systems, from simplified use of the JMS API using `JmsTemplate` to a complete infrastructure to receive messages asynchronously. Spring AMQP provides a similar feature set for the Advanced Message Queuing Protocol. Spring Boot also provides auto-configuration options for `RabbitTemplate` and RabbitMQ. Spring WebSocket natively includes support for STOMP messaging, and Spring Boot has support for that through starters and a small amount of auto-configuration. Spring Boot also has support for Apache Kafka and Apache Pulsar.

10.1. JMS

The `jakarta.jms.ConnectionFactory` interface provides a standard method of creating a `jakarta.jms.Connection` for interacting with a JMS broker. Although Spring needs a `ConnectionFactory` to work with JMS, you generally need not use it directly yourself and can instead rely on higher level messaging abstractions. (See the relevant section of the Spring Framework reference documentation for details.) Spring Boot also auto-configures the necessary infrastructure to send and receive messages.

10.1.1. ActiveMQ "Classic" Support

When ActiveMQ "Classic" is available on the classpath, Spring Boot can configure a `ConnectionFactory`.

NOTE	If you use `spring-boot-starter-activemq`, the necessary dependencies to connect to an ActiveMQ "Classic" instance are provided, as is the Spring infrastructure to integrate with JMS.

ActiveMQ "Classic" configuration is controlled by external configuration properties in `spring.activemq.*`. By default, ActiveMQ "Classic" is auto-configured to use the TCP transport, connecting by default to `tcp://localhost:61616`. The following example shows how to change the default broker URL:

Properties

```
spring.activemq.broker-url=tcp://192.168.1.210:9876
spring.activemq.user=admin
spring.activemq.password=secret
```

Yaml

```
spring:
  activemq:
    broker-url: "tcp://192.168.1.210:9876"
    user: "admin"
    password: "secret"
```

By default, a `CachingConnectionFactory` wraps the native `ConnectionFactory` with sensible settings that you can control by external configuration properties in `spring.jms.*`:

Properties

```
spring.jms.cache.session-cache-size=5
```

Yaml

```
spring:
  jms:
    cache:
      session-cache-size: 5
```

If you'd rather use native pooling, you can do so by adding a dependency to `org.messaginghub:pooled-jms` and configuring the `JmsPoolConnectionFactory` accordingly, as shown in the following example:

Properties

```
spring.activemq.pool.enabled=true
spring.activemq.pool.max-connections=50
```

Yaml

```
spring:
  activemq:
    pool:
      enabled: true
      max-connections: 50
```

TIP See `ActiveMQProperties` for more of the supported options. You can also register an arbitrary number of beans that implement `ActiveMQConnectionFactoryCustomizer` for more advanced customizations.

By default, ActiveMQ "Classic" creates a destination if it does not yet exist so that destinations are resolved against their provided names.

10.1.2. ActiveMQ Artemis Support

Spring Boot can auto-configure a `ConnectionFactory` when it detects that ActiveMQ Artemis is available on the classpath. If the broker is present, an embedded broker is automatically started and configured (unless the mode property has been explicitly set). The supported modes are `embedded` (to make explicit that an embedded broker is required and that an error should occur if the broker is not available on the classpath) and `native` (to connect to a broker using the `netty` transport protocol). When the latter is configured, Spring Boot configures a `ConnectionFactory` that connects to a broker running on the local machine with the default settings.

NOTE	If you use `spring-boot-starter-artemis`, the necessary dependencies to connect to an existing ActiveMQ Artemis instance are provided, as well as the Spring infrastructure to integrate with JMS. Adding `org.apache.activemq:artemis-jakarta-server` to your application lets you use embedded mode.

ActiveMQ Artemis configuration is controlled by external configuration properties in `spring.artemis.*`. For example, you might declare the following section in `application.properties`:

Properties

```
spring.artemis.mode=native
spring.artemis.broker-url=tcp://192.168.1.210:9876
spring.artemis.user=admin
spring.artemis.password=secret
```

Yaml

```
spring:
  artemis:
    mode: native
    broker-url: "tcp://192.168.1.210:9876"
    user: "admin"
    password: "secret"
```

When embedding the broker, you can choose if you want to enable persistence and list the destinations that should be made available. These can be specified as a comma-separated list to create them with the default options, or you can define bean(s) of type `org.apache.activemq.artemis.jms.server.config.JMSQueueConfiguration` or `org.apache.activemq.artemis.jms.server.config.TopicConfiguration`, for advanced queue and topic configurations, respectively.

By default, a `CachingConnectionFactory` wraps the native `ConnectionFactory` with sensible settings that you can control by external configuration properties in `spring.jms.*`:

Properties

```
spring.jms.cache.session-cache-size=5
```

Yaml

```
spring:
  jms:
    cache:
      session-cache-size: 5
```

If you'd rather use native pooling, you can do so by adding a dependency on `org.messaginghub:pooled-jms` and configuring the `JmsPoolConnectionFactory` accordingly, as shown in the following example:

Properties

```
spring.artemis.pool.enabled=true
spring.artemis.pool.max-connections=50
```

Yaml

```
spring:
  artemis:
    pool:
      enabled: true
      max-connections: 50
```

See `ArtemisProperties` for more supported options.

No JNDI lookup is involved, and destinations are resolved against their names, using either the `name` attribute in the ActiveMQ Artemis configuration or the names provided through configuration.

10.1.3. Using a JNDI ConnectionFactory

If you are running your application in an application server, Spring Boot tries to locate a JMS `ConnectionFactory` by using JNDI. By default, the `java:/JmsXA` and `java:/XAConnectionFactory` location are checked. You can use the `spring.jms.jndi-name` property if you need to specify an alternative location, as shown in the following example:

Properties

```
spring.jms.jndi-name=java:/MyConnectionFactory
```

Yaml

```
spring:
  jms:
    jndi-name: "java:/MyConnectionFactory"
```

10.1.4. Sending a Message

Spring's `JmsTemplate` is auto-configured, and you can autowire it directly into your own beans, as shown in the following example:

Java

```
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final JmsTemplate jmsTemplate;

    public MyBean(JmsTemplate jmsTemplate) {
        this.jmsTemplate = jmsTemplate;
    }

    public void someMethod() {
        this.jmsTemplate.convertAndSend("hello");
    }

}
```

Kotlin

```
import org.springframework.jms.core.JmsTemplate
import org.springframework.stereotype.Component

@Component
class MyBean(private val jmsTemplate: JmsTemplate) {

    fun someMethod() {
        jmsTemplate.convertAndSend("hello")
    }

}
```

NOTE	`JmsMessagingTemplate` can be injected in a similar manner. If a `DestinationResolver` or a `MessageConverter` bean is defined, it is associated automatically to the auto-configured `JmsTemplate`.

10.1.5. Receiving a Message

When the JMS infrastructure is present, any bean can be annotated with `@JmsListener` to create a listener endpoint. If no `JmsListenerContainerFactory` has been defined, a default one is configured automatically. If a `DestinationResolver`, a `MessageConverter`, or a `jakarta.jms.ExceptionListener` beans are defined, they are associated automatically with the default factory.

By default, the default factory is transactional. If you run in an infrastructure where a `JtaTransactionManager` is present, it is associated to the listener container by default. If not, the `sessionTransacted` flag is enabled. In that latter scenario, you can associate your local data store transaction to the processing of an incoming message by adding `@Transactional` on your listener

method (or a delegate thereof). This ensures that the incoming message is acknowledged, once the local transaction has completed. This also includes sending response messages that have been performed on the same JMS session.

The following component creates a listener endpoint on the `someQueue` destination:

Java

```
import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    @JmsListener(destination = "someQueue")
    public void processMessage(String content) {
        // ...
    }

}
```

Kotlin

```
import org.springframework.jms.annotation.JmsListener
import org.springframework.stereotype.Component

@Component
class MyBean {

    @JmsListener(destination = "someQueue")
    fun processMessage(content: String?) {
        // ...
    }

}
```

TIP See the Javadoc of `@EnableJms` for more details.

If you need to create more `JmsListenerContainerFactory` instances or if you want to override the default, Spring Boot provides a `DefaultJmsListenerContainerFactoryConfigurer` that you can use to initialize a `DefaultJmsListenerContainerFactory` with the same settings as the one that is auto-configured.

For instance, the following example exposes another factory that uses a specific `MessageConverter`:

Java

```
import jakarta.jms.ConnectionFactory;

import
org.springframework.boot.autoconfigure.jms.DefaultJmsListenerContainerFactoryConfigure
r;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jms.config.DefaultJmsListenerContainerFactory;

@Configuration(proxyBeanMethods = false)
public class MyJmsConfiguration {

    @Bean
    public DefaultJmsListenerContainerFactory
myFactory(DefaultJmsListenerContainerFactoryConfigurer configurer) {
        DefaultJmsListenerContainerFactory factory = new
DefaultJmsListenerContainerFactory();
        ConnectionFactory connectionFactory = getCustomConnectionFactory();
        configurer.configure(factory, connectionFactory);
        factory.setMessageConverter(new MyMessageConverter());
        return factory;
    }

    private ConnectionFactory getCustomConnectionFactory() {
        return ...
    }

}
```

Kotlin

```
import jakarta.jms.ConnectionFactory
import
org.springframework.boot.autoconfigure.jms.DefaultJmsListenerContainerFactoryConfigure
r
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.jms.config.DefaultJmsListenerContainerFactory

@Configuration(proxyBeanMethods = false)
class MyJmsConfiguration {

    @Bean
    fun myFactory(configurer: DefaultJmsListenerContainerFactoryConfigurer):
DefaultJmsListenerContainerFactory {
        val factory = DefaultJmsListenerContainerFactory()
        val connectionFactory = getCustomConnectionFactory()
        configurer.configure(factory, connectionFactory)
        factory.setMessageConverter(MyMessageConverter())
        return factory
    }

    fun getCustomConnectionFactory() : ConnectionFactory? {
        return ...
    }

}
```

Then you can use the factory in any `@JmsListener`-annotated method as follows:

Java

```
import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    @JmsListener(destination = "someQueue", containerFactory = "myFactory")
    public void processMessage(String content) {
        // ...
    }

}
```

Kotlin

```
import org.springframework.jms.annotation.JmsListener
import org.springframework.stereotype.Component

@Component
class MyBean {

    @JmsListener(destination = "someQueue", containerFactory = "myFactory")
    fun processMessage(content: String?) {
        // ...
    }

}
```

10.2. AMQP

The Advanced Message Queuing Protocol (AMQP) is a platform-neutral, wire-level protocol for message-oriented middleware. The Spring AMQP project applies core Spring concepts to the development of AMQP-based messaging solutions. Spring Boot offers several conveniences for working with AMQP through RabbitMQ, including the `spring-boot-starter-amqp` "Starter".

10.2.1. RabbitMQ Support

RabbitMQ is a lightweight, reliable, scalable, and portable message broker based on the AMQP protocol. Spring uses RabbitMQ to communicate through the AMQP protocol.

RabbitMQ configuration is controlled by external configuration properties in `spring.rabbitmq.*`. For example, you might declare the following section in `application.properties`:

Properties

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=admin
spring.rabbitmq.password=secret
```

Yaml

```
spring:
  rabbitmq:
    host: "localhost"
    port: 5672
    username: "admin"
    password: "secret"
```

Alternatively, you could configure the same connection using the `addresses` attribute:

Properties

```
spring.rabbitmq.addresses=amqp://admin:secret@localhost
```

Yaml

```
spring:
  rabbitmq:
    addresses: "amqp://admin:secret@localhost"
```

NOTE	When specifying addresses that way, the `host` and `port` properties are ignored. If the address uses the `amqps` protocol, SSL support is enabled automatically.

See `RabbitProperties` for more of the supported property-based configuration options. To configure lower-level details of the RabbitMQ `ConnectionFactory` that is used by Spring AMQP, define a `ConnectionFactoryCustomizer` bean.

If a `ConnectionNameStrategy` bean exists in the context, it will be automatically used to name connections created by the auto-configured `CachingConnectionFactory`.

To make an application-wide, additive customization to the `RabbitTemplate`, use a `RabbitTemplateCustomizer` bean.

TIP	See Understanding AMQP, the protocol used by RabbitMQ for more details.

10.2.2. Sending a Message

Spring's `AmqpTemplate` and `AmqpAdmin` are auto-configured, and you can autowire them directly into your own beans, as shown in the following example:

Java

```
import org.springframework.amqp.core.AmqpAdmin;
import org.springframework.amqp.core.AmqpTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final AmqpAdmin amqpAdmin;

    private final AmqpTemplate amqpTemplate;

    public MyBean(AmqpAdmin amqpAdmin, AmqpTemplate amqpTemplate) {
        this.amqpAdmin = amqpAdmin;
        this.amqpTemplate = amqpTemplate;
    }

    public void someMethod() {
        this.amqpAdmin.getQueueInfo("someQueue");
    }

    public void someOtherMethod() {
        this.amqpTemplate.convertAndSend("hello");
    }

}
```

Kotlin

```
import org.springframework.amqp.core.AmqpAdmin
import org.springframework.amqp.core.AmqpTemplate
import org.springframework.stereotype.Component

@Component
class MyBean(private val amqpAdmin: AmqpAdmin, private val amqpTemplate: AmqpTemplate)
{

    fun someMethod() {
        amqpAdmin.getQueueInfo("someQueue")
    }

    fun someOtherMethod() {
        amqpTemplate.convertAndSend("hello")
    }

}
```

NOTE	`RabbitMessagingTemplate` can be injected in a similar manner. If a `MessageConverter` bean is defined, it is associated automatically to the auto-configured `AmqpTemplate`.

If necessary, any `org.springframework.amqp.core.Queue` that is defined as a bean is automatically used to declare a corresponding queue on the RabbitMQ instance.

To retry operations, you can enable retries on the `AmqpTemplate` (for example, in the event that the broker connection is lost):

Properties

```
spring.rabbitmq.template.retry.enabled=true
spring.rabbitmq.template.retry.initial-interval=2s
```

Yaml

```
spring:
  rabbitmq:
    template:
      retry:
        enabled: true
        initial-interval: "2s"
```

Retries are disabled by default. You can also customize the `RetryTemplate` programmatically by declaring a `RabbitRetryTemplateCustomizer` bean.

If you need to create more `RabbitTemplate` instances or if you want to override the default, Spring Boot provides a `RabbitTemplateConfigurer` bean that you can use to initialize a `RabbitTemplate` with the same settings as the factories used by the auto-configuration.

10.2.3. Sending a Message To A Stream

To send a message to a particular stream, specify the name of the stream, as shown in the following example:

Properties

```
spring.rabbitmq.stream.name=my-stream
```

Yaml

```
spring:
  rabbitmq:
    stream:
      name: "my-stream"
```

If a `MessageConverter`, `StreamMessageConverter`, or `ProducerCustomizer` bean is defined, it is associated automatically to the auto-configured `RabbitStreamTemplate`.

If you need to create more `RabbitStreamTemplate` instances or if you want to override the default, Spring Boot provides a `RabbitStreamTemplateConfigurer` bean that you can use to initialize a `RabbitStreamTemplate` with the same settings as the factories used by the auto-configuration.

10.2.4. Receiving a Message

When the Rabbit infrastructure is present, any bean can be annotated with `@RabbitListener` to create a listener endpoint. If no `RabbitListenerContainerFactory` has been defined, a default `SimpleRabbitListenerContainerFactory` is automatically configured and you can switch to a direct container using the `spring.rabbitmq.listener.type` property. If a `MessageConverter` or a `MessageRecoverer` bean is defined, it is automatically associated with the default factory.

The following sample component creates a listener endpoint on the `someQueue` queue:

Java

```
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    @RabbitListener(queues = "someQueue")
    public void processMessage(String content) {
        // ...
    }

}
```

Kotlin

```
import org.springframework.amqp.rabbit.annotation.RabbitListener
import org.springframework.stereotype.Component

@Component
class MyBean {

    @RabbitListener(queues = ["someQueue"])
    fun processMessage(content: String?) {
        // ...
    }

}
```

TIP See the Javadoc of `@EnableRabbit` for more details.

If you need to create more `RabbitListenerContainerFactory` instances or if you want to override the default, Spring Boot provides a `SimpleRabbitListenerContainerFactoryConfigurer` and a `DirectRabbitListenerContainerFactoryConfigurer` that you can use to initialize a

`SimpleRabbitListenerContainerFactory` and a `DirectRabbitListenerContainerFactory` with the same settings as the factories used by the auto-configuration.

TIP	It does not matter which container type you chose. Those two beans are exposed by the auto-configuration.

For instance, the following configuration class exposes another factory that uses a specific `MessageConverter`:

Java

```
import org.springframework.amqp.rabbit.config.SimpleRabbitListenerContainerFactory;
import org.springframework.amqp.rabbit.connection.ConnectionFactory;
import
org.springframework.boot.autoconfigure.amqp.SimpleRabbitListenerContainerFactoryConfig
urer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration(proxyBeanMethods = false)
public class MyRabbitConfiguration {

    @Bean
    public SimpleRabbitListenerContainerFactory
myFactory(SimpleRabbitListenerContainerFactoryConfigurer configurer) {
        SimpleRabbitListenerContainerFactory factory = new
SimpleRabbitListenerContainerFactory();
        ConnectionFactory connectionFactory = getCustomConnectionFactory();
        configurer.configure(factory, connectionFactory);
        factory.setMessageConverter(new MyMessageConverter());
        return factory;
    }

    private ConnectionFactory getCustomConnectionFactory() {
        return ...
    }

}
```

Kotlin

```
import org.springframework.amqp.rabbit.config.SimpleRabbitListenerContainerFactory
import org.springframework.amqp.rabbit.connection.ConnectionFactory
import
org.springframework.boot.autoconfigure.amqp.SimpleRabbitListenerContainerFactoryConfig
urer
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration

@Configuration(proxyBeanMethods = false)
class MyRabbitConfiguration {

    @Bean
    fun myFactory(configurer: SimpleRabbitListenerContainerFactoryConfigurer):
SimpleRabbitListenerContainerFactory {
        val factory = SimpleRabbitListenerContainerFactory()
        val connectionFactory = getCustomConnectionFactory()
        configurer.configure(factory, connectionFactory)
        factory.setMessageConverter(MyMessageConverter())
        return factory
    }

    fun getCustomConnectionFactory() : ConnectionFactory? {
        return ...
    }

}
```

Then you can use the factory in any `@RabbitListener`-annotated method, as follows:

Java

```
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    @RabbitListener(queues = "someQueue", containerFactory = "myFactory")
    public void processMessage(String content) {
        // ...
    }

}
```

Kotlin

```
import org.springframework.amqp.rabbit.annotation.RabbitListener
import org.springframework.stereotype.Component

@Component
class MyBean {

    @RabbitListener(queues = ["someQueue"], containerFactory = "myFactory")
    fun processMessage(content: String?) {
        // ...
    }

}
```

You can enable retries to handle situations where your listener throws an exception. By default, `RejectAndDontRequeueRecoverer` is used, but you can define a `MessageRecoverer` of your own. When retries are exhausted, the message is rejected and either dropped or routed to a dead-letter exchange if the broker is configured to do so. By default, retries are disabled. You can also customize the `RetryTemplate` programmatically by declaring a `RabbitRetryTemplateCustomizer` bean.

IMPORTANT

By default, if retries are disabled and the listener throws an exception, the delivery is retried indefinitely. You can modify this behavior in two ways: Set the `defaultRequeueRejected` property to `false` so that zero re-deliveries are attempted or throw an `AmqpRejectAndDontRequeueException` to signal the message should be rejected. The latter is the mechanism used when retries are enabled and the maximum number of delivery attempts is reached.

10.3. Apache Kafka Support

Apache Kafka is supported by providing auto-configuration of the `spring-kafka` project.

Kafka configuration is controlled by external configuration properties in `spring.kafka.*`. For example, you might declare the following section in `application.properties`:

Properties

```
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.consumer.group-id=myGroup
```

Yaml

```
spring:
  kafka:
    bootstrap-servers: "localhost:9092"
    consumer:
      group-id: "myGroup"
```

TIP	To create a topic on startup, add a bean of type `NewTopic`. If the topic already exists, the bean is ignored.

See `KafkaProperties` for more supported options.

10.3.1. Sending a Message

Spring's `KafkaTemplate` is auto-configured, and you can autowire it directly in your own beans, as shown in the following example:

Java

```
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final KafkaTemplate<String, String> kafkaTemplate;

    public MyBean(KafkaTemplate<String, String> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

    public void someMethod() {
        this.kafkaTemplate.send("someTopic", "Hello");
    }

}
```

Kotlin

```
import org.springframework.kafka.core.KafkaTemplate
import org.springframework.stereotype.Component

@Component
class MyBean(private val kafkaTemplate: KafkaTemplate<String, String>) {

    fun someMethod() {
        kafkaTemplate.send("someTopic", "Hello")
    }

}
```

NOTE	If the property `spring.kafka.producer.transaction-id-prefix` is defined, a `KafkaTransactionManager` is automatically configured. Also, if a `RecordMessageConverter` bean is defined, it is automatically associated to the auto-configured `KafkaTemplate`.

10.3.2. Receiving a Message

When the Apache Kafka infrastructure is present, any bean can be annotated with `@KafkaListener` to create a listener endpoint. If no `KafkaListenerContainerFactory` has been defined, a default one is automatically configured with keys defined in `spring.kafka.listener.*`.

The following component creates a listener endpoint on the `someTopic` topic:

Java

```
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    @KafkaListener(topics = "someTopic")
    public void processMessage(String content) {
        // ...
    }

}
```

Kotlin

```
import org.springframework.kafka.annotation.KafkaListener
import org.springframework.stereotype.Component

@Component
class MyBean {

    @KafkaListener(topics = ["someTopic"])
    fun processMessage(content: String?) {
        // ...
    }

}
```

If a `KafkaTransactionManager` bean is defined, it is automatically associated to the container factory. Similarly, if a `RecordFilterStrategy`, `CommonErrorHandler`, `AfterRollbackProcessor` or `ConsumerAwareRebalanceListener` bean is defined, it is automatically associated to the default factory.

Depending on the listener type, a `RecordMessageConverter` or `BatchMessageConverter` bean is associated to the default factory. If only a `RecordMessageConverter` bean is present for a batch listener, it is wrapped in a `BatchMessageConverter`.

TIP A custom `ChainedKafkaTransactionManager` must be marked `@Primary` as it usually references the auto-configured `KafkaTransactionManager` bean.

10.3.3. Kafka Streams

Spring for Apache Kafka provides a factory bean to create a `StreamsBuilder` object and manage the lifecycle of its streams. Spring Boot auto-configures the required `KafkaStreamsConfiguration` bean as long as `kafka-streams` is on the classpath and Kafka Streams is enabled by the `@EnableKafkaStreams` annotation.

Enabling Kafka Streams means that the application id and bootstrap servers must be set. The former can be configured using `spring.kafka.streams.application-id`, defaulting to `spring.application.name` if not set. The latter can be set globally or specifically overridden only for streams.

Several additional properties are available using dedicated properties; other arbitrary Kafka properties can be set using the `spring.kafka.streams.properties` namespace. See also Additional Kafka Properties for more information.

To use the factory bean, wire `StreamsBuilder` into your `@Bean` as shown in the following example:

Java

```
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KeyValue;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.Produced;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.annotation.EnableKafkaStreams;
import org.springframework.kafka.support.serializer.JsonSerde;

@Configuration(proxyBeanMethods = false)
@EnableKafkaStreams
public class MyKafkaStreamsConfiguration {

    @Bean
    public KStream<Integer, String> kStream(StreamsBuilder streamsBuilder) {
        KStream<Integer, String> stream = streamsBuilder.stream("ks1In");
        stream.map(this::uppercaseValue).to("ks1Out", Produced.with(Serdes.Integer(),
new JsonSerde<>()));
        return stream;
    }

    private KeyValue<Integer, String> uppercaseValue(Integer key, String value) {
        return new KeyValue<>(key, value.toUpperCase());
    }

}
```

Kotlin

```
import org.apache.kafka.common.serialization.Serdes
import org.apache.kafka.streams.KeyValue
import org.apache.kafka.streams.StreamsBuilder
import org.apache.kafka.streams.kstream.KStream
import org.apache.kafka.streams.kstream.Produced
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.kafka.annotation.EnableKafkaStreams
import org.springframework.kafka.support.serializer.JsonSerde

@Configuration(proxyBeanMethods = false)
@EnableKafkaStreams
class MyKafkaStreamsConfiguration {

    @Bean
    fun kStream(streamsBuilder: StreamsBuilder): KStream<Int, String> {
        val stream = streamsBuilder.stream<Int, String>("ks1In")
        stream.map(this::uppercaseValue).to("ks1Out", Produced.with(Serdes.Integer(),
JsonSerde()))
        return stream
    }

    private fun uppercaseValue(key: Int, value: String): KeyValue<Int?, String?> {
        return KeyValue(key, value.uppercase())
    }

}
```

By default, the streams managed by the `StreamBuilder` object are started automatically. You can customize this behavior using the `spring.kafka.streams.auto-startup` property.

10.3.4. Additional Kafka Properties

The properties supported by auto configuration are shown in the "Integration Properties" section of the Appendix. Note that, for the most part, these properties (hyphenated or camelCase) map directly to the Apache Kafka dotted properties. See the Apache Kafka documentation for details.

Properties that don't include a client type (`producer`, `consumer`, `admin`, or `streams`) in their name are considered to be common and apply to all clients. Most of these common properties can be overridden for one or more of the client types, if needed.

Apache Kafka designates properties with an importance of HIGH, MEDIUM, or LOW. Spring Boot auto-configuration supports all HIGH importance properties, some selected MEDIUM and LOW properties, and any properties that do not have a default value.

Only a subset of the properties supported by Kafka are available directly through the `KafkaProperties` class. If you wish to configure the individual client types with additional properties that are not directly supported, use the following properties:

Properties

```
spring.kafka.properties[prop.one]=first
spring.kafka.admin.properties[prop.two]=second
spring.kafka.consumer.properties[prop.three]=third
spring.kafka.producer.properties[prop.four]=fourth
spring.kafka.streams.properties[prop.five]=fifth
```

Yaml

```
spring:
  kafka:
    properties:
      "[prop.one]": "first"
    admin:
      properties:
        "[prop.two]": "second"
    consumer:
      properties:
        "[prop.three]": "third"
    producer:
      properties:
        "[prop.four]": "fourth"
    streams:
      properties:
        "[prop.five]": "fifth"
```

This sets the common `prop.one` Kafka property to `first` (applies to producers, consumers, admins, and streams), the `prop.two` admin property to `second`, the `prop.three` consumer property to `third`, the `prop.four` producer property to `fourth` and the `prop.five` streams property to `fifth`.

You can also configure the Spring Kafka `JsonDeserializer` as follows:

Properties

```
spring.kafka.consumer.value-
deserializer=org.springframework.kafka.support.serializer.JsonDeserializer
spring.kafka.consumer.properties[spring.json.value.default.type]=com.example.Invoice
spring.kafka.consumer.properties[spring.json.trusted.packages]=com.example.main,com.ex
ample.another
```

Yaml

```
spring:
  kafka:
    consumer:
      value-deserializer:
"org.springframework.kafka.support.serializer.JsonDeserializer"
      properties:
        "[spring.json.value.default.type]": "com.example.Invoice"
        "[spring.json.trusted.packages]": "com.example.main,com.example.another"
```

Similarly, you can disable the `JsonSerializer` default behavior of sending type information in headers:

Properties

```
spring.kafka.producer.value-
serializer=org.springframework.kafka.support.serializer.JsonSerializer
spring.kafka.producer.properties[spring.json.add.type.headers]=false
```

Yaml

```
spring:
  kafka:
    producer:
      value-serializer: "org.springframework.kafka.support.serializer.JsonSerializer"
      properties:
        "[spring.json.add.type.headers]": false
```

IMPORTANT	Properties set in this way override any configuration item that Spring Boot explicitly supports.

10.3.5. Testing with Embedded Kafka

Spring for Apache Kafka provides a convenient way to test projects with an embedded Apache Kafka broker. To use this feature, annotate a test class with `@EmbeddedKafka` from the `spring-kafka-test` module. For more information, please see the Spring for Apache Kafka reference manual.

To make Spring Boot auto-configuration work with the aforementioned embedded Apache Kafka broker, you need to remap a system property for embedded broker addresses (populated by the `EmbeddedKafkaBroker`) into the Spring Boot configuration property for Apache Kafka. There are several ways to do that:

- Provide a system property to map embedded broker addresses into `spring.kafka.bootstrap-servers` in the test class:

Java

```
static {
    System.setProperty(EmbeddedKafkaBroker.BROKER_LIST_PROPERTY,
"spring.kafka.bootstrap-servers");
}
```

Kotlin

```
init {
    System.setProperty(EmbeddedKafkaBroker.BROKER_LIST_PROPERTY,
"spring.kafka.bootstrap-servers")
}
```

- Configure a property name on the `@EmbeddedKafka` annotation:

Java

```
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.kafka.test.context.EmbeddedKafka;

@SpringBootTest
@EmbeddedKafka(topics = "someTopic", bootstrapServersProperty =
"spring.kafka.bootstrap-servers")
class MyTest {

    // ...

}
```

Kotlin

```
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.kafka.test.context.EmbeddedKafka

@SpringBootTest
@EmbeddedKafka(topics = ["someTopic"], bootstrapServersProperty =
"spring.kafka.bootstrap-servers")
class MyTest {

    // ...

}
```

- Use a placeholder in configuration properties:

Properties

```
spring.kafka.bootstrap-servers=${spring.embedded.kafka.brokers}
```

Yaml

```
spring:
  kafka:
    bootstrap-servers: "${spring.embedded.kafka.brokers}"
```

10.4. Apache Pulsar Support

Apache Pulsar is supported by providing auto-configuration of the Spring for Apache Pulsar project.

Spring Boot will auto-configure and register the classic (imperative) Spring for Apache Pulsar components when `org.springframework.pulsar:spring-pulsar` is on the classpath. It will do the same for the reactive components when `org.springframework.pulsar:spring-pulsar-reactive` is on the classpath.

There are `spring-boot-starter-pulsar` and `spring-boot-starter-pulsar-reactive` "Starters" for conveniently collecting the dependencies for imperative and reactive use, respectively.

10.4.1. Connecting to Pulsar

When you use the Pulsar starter, Spring Boot will auto-configure and register a `PulsarClient` bean.

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.

NOTE	The value must be a valid Pulsar Protocol URL

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

If you need more control over the configuration, consider registering one or more `PulsarClientBuilderCustomizer` beans.

Authentication

To connect to a Pulsar cluster that requires authentication, you need to specify which authentication plugin to use by setting the `pluginClassName` and any parameters required by the plugin. You can set the parameters as a map of parameter names to parameter values. The following example shows how to configure the `AuthenticationOAuth2` plugin.

Properties

```
spring.pulsar.client.authentication.plugin-class-
name=org.apache.pulsar.client.impl.auth.oauth2.AuthenticationOAuth2
spring.pulsar.client.authentication.param[issuerUrl]=https://auth.server.cloud/
spring.pulsar.client.authentication.param[privateKey]=file:///Users/some-key.json
spring.pulsar.client.authentication.param.audience=urn:sn:acme:dev:my-instance
```

Yaml

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

NOTE

You need to ensure that names defined under `spring.pulsar.client.authentication.param.*` exactly match those expected by your auth plugin (which is typically camel cased). Spring Boot will not attempt any kind of relaxed binding for these entries.

For example, if you want to configure the issuer url for the `AuthenticationOAuth2` auth plugin you must use `spring.pulsar.client.authentication.param.issuerUrl`. If you use other forms, such as `issuerurl` or `issuer-url`, the setting will not be applied to the plugin.

This lack of relaxed binding also makes using environment variables for authentication parameters problematic because the case sensitivity is lost during translation. If you use environment variables for the parameters then you will need to follow these steps in the Spring for Apache Pulsar reference documentation for it to work properly.

SSL

By default, Pulsar clients communicate with Pulsar services in plain text. You can follow these steps in the Spring for Apache Pulsar reference documentation to enable TLS encryption.

For complete details on the client and authentication see the Spring for Apache Pulsar reference documentation.

10.4.2. Connecting to Pulsar Reactively

When the Reactive auto-configuration is activated, Spring Boot will auto-configure and register a `ReactivePulsarClient` bean.

The `ReactivePulsarClient` adapts an instance of the previously described `PulsarClient`. Therefore, follow the previous section to configure the `PulsarClient` used by the `ReactivePulsarClient`.

10.4.3. Connecting to Pulsar Administration

Spring for Apache Pulsar's `PulsarAdministration` client is also 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.admin.service-url` property to a different value in the form `(http|https)://<host>:<port>`.

If you need more control over the configuration, consider registering one or more `PulsarAdminBuilderCustomizer` beans.

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 authentication configuration by replacing `spring.pulsar.client.authentication` with `spring.pulsar.admin.authentication`.

> **TIP** To create a topic on startup, add a bean of type `PulsarTopic`. If the topic already exists, the bean is ignored.

10.4.4. Sending a Message

Spring's `PulsarTemplate` is auto-configured, and you can use it to send messages, as shown in the following example:

Java

```
import org.apache.pulsar.client.api.PulsarClientException;

import org.springframework.pulsar.core.PulsarTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final PulsarTemplate<String> pulsarTemplate;

    public MyBean(PulsarTemplate<String> pulsarTemplate) {
        this.pulsarTemplate = pulsarTemplate;
    }

    public void someMethod() throws PulsarClientException {
        this.pulsarTemplate.send("someTopic", "Hello");
    }

}
```

Kotlin

```
import org.apache.pulsar.client.api.PulsarClientException
import org.springframework.pulsar.core.PulsarTemplate
import org.springframework.stereotype.Component

@Component
class MyBean(private val pulsarTemplate: PulsarTemplate<String>) {

    @Throws(PulsarClientException::class)
    fun someMethod() {
        pulsarTemplate.send("someTopic", "Hello")
    }

}
```

The `PulsarTemplate` relies on a `PulsarProducerFactory` to create the underlying Pulsar producer. Spring Boot auto-configuration also provides this producer factory, which by default, caches the producers that it creates. You can configure the producer factory and cache settings by specifying any of the `spring.pulsar.producer.*` and `spring.pulsar.producer.cache.*` prefixed application properties.

If you need more control over the producer factory configuration, consider registering one or more `ProducerBuilderCustomizer` beans. These customizers are applied to all created producers. You can also pass in a `ProducerBuilderCustomizer` when sending a message to only affect the current producer.

If you need more control over the message being sent, you can pass in a `TypedMessageBuilderCustomizer` when sending a message.

10.4.5. Sending a Message Reactively

When the Reactive auto-configuration is activated, Spring's `ReactivePulsarTemplate` is auto-configured, and you can use it to send messages, as shown in the following example:

Java

```
import org.springframework.pulsar.reactive.core.ReactivePulsarTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final ReactivePulsarTemplate<String> pulsarTemplate;

    public MyBean(ReactivePulsarTemplate<String> pulsarTemplate) {
        this.pulsarTemplate = pulsarTemplate;
    }

    public void someMethod() {
        this.pulsarTemplate.send("someTopic", "Hello").subscribe();
    }

}
```

Kotlin

```
import org.springframework.pulsar.reactive.core.ReactivePulsarTemplate
import org.springframework.stereotype.Component

@Component
class MyBean(private val pulsarTemplate: ReactivePulsarTemplate<String>) {

    fun someMethod() {
        pulsarTemplate.send("someTopic", "Hello").subscribe()
    }

}
```

The `ReactivePulsarTemplate` relies on a `ReactivePulsarSenderFactory` to actually create the underlying sender. Spring Boot auto-configuration also provides this sender factory, which by default, caches the producers that it creates. You can configure the sender factory and cache settings by specifying any of the `spring.pulsar.producer.*` and `spring.pulsar.producer.cache.*` prefixed application properties.

If you need more control over the sender factory configuration, consider registering one or more

`ReactiveMessageSenderBuilderCustomizer` beans. These customizers are applied to all created senders. You can also pass in a `ReactiveMessageSenderBuilderCustomizer` when sending a message to only affect the current sender.

If you need more control over the message being sent, you can pass in a `MessageSpecBuilderCustomizer` when sending a message.

10.4.6. Receiving a Message

When the Apache Pulsar infrastructure is present, any bean can be annotated with `@PulsarListener` to create a listener endpoint. The following component creates a listener endpoint on the `someTopic` topic:

Java

```
import org.springframework.pulsar.annotation.PulsarListener;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    @PulsarListener(topics = "someTopic")
    public void processMessage(String content) {
        // ...
    }

}
```

Kotlin

```
import org.springframework.pulsar.annotation.PulsarListener
import org.springframework.stereotype.Component

@Component
class MyBean {

    @PulsarListener(topics = ["someTopic"])
    fun processMessage(content: String?) {
        // ...
    }

}
```

Spring Boot auto-configuration provides all the components necessary for `PulsarListener`, such as the `PulsarListenerContainerFactory` and the consumer factory it uses to construct the underlying Pulsar consumers. You can configure these components by specifying any of the `spring.pulsar.listener.*` and `spring.pulsar.consumer.*` prefixed application properties.

If you need more control over the consumer factory configuration, consider registering one or

more `ConsumerBuilderCustomizer` beans. These customizers are applied to all consumers created by the factory, and therefore all `@PulsarListener` instances. You can also customize a single listener by setting the `consumerCustomizer` attribute of the `@PulsarListener` annotation.

10.4.7. Receiving a Message Reactively

When the Apache Pulsar infrastructure is present and the Reactive auto-configuration is activated, any bean can be annotated with `@ReactivePulsarListener` to create a reactive listener endpoint. The following component creates a reactive listener endpoint on the `someTopic` topic:

Java

```
import reactor.core.publisher.Mono;

import org.springframework.pulsar.reactive.config.annotation.ReactivePulsarListener;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    @ReactivePulsarListener(topics = "someTopic")
    public Mono<Void> processMessage(String content) {
        // ...
        return Mono.empty();
    }

}
```

Kotlin

```
import org.springframework.pulsar.reactive.config.annotation.ReactivePulsarListener
import org.springframework.stereotype.Component
import reactor.core.publisher.Mono

@Component
class MyBean {

    @ReactivePulsarListener(topics = ["someTopic"])
    fun processMessage(content: String?): Mono<Void> {
        // ...
        return Mono.empty()
    }

}
```

Spring Boot auto-configuration provides all the components necessary for `ReactivePulsarListener`, such as the `ReactivePulsarListenerContainerFactory` and the consumer factory it uses to construct the underlying reactive Pulsar consumers. You can configure these components by specifying any of the `spring.pulsar.listener. and spring.pulsar.consumer.` prefixed application properties.

If you need more control over the consumer factory configuration, consider registering one or more `ReactiveMessageConsumerBuilderCustomizer` beans. These customizers are applied to all consumers created by the factory, and therefore all `@ReactivePulsarListener` instances. You can also customize a single listener by setting the `consumerCustomizer` attribute of the `@ReactivePulsarListener` annotation.

10.4.8. Reading a Message

The Pulsar reader interface enables applications to manually manage cursors. When you use a reader to connect to a topic you need to specify which message the reader begins reading from when it connects to a topic.

When the Apache Pulsar infrastructure is present, any bean can be annotated with `@PulsarReader` to consume messages using a reader. The following component creates a reader endpoint that starts reading messages from the beginning of the `someTopic` topic:

Java

```
import org.springframework.pulsar.annotation.PulsarReader;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    @PulsarReader(topics = "someTopic", startMessageId = "earliest")
    public void processMessage(String content) {
        // ...
    }

}
```

Kotlin

```
import org.springframework.pulsar.annotation.PulsarReader
import org.springframework.stereotype.Component

@Component
class MyBean {

    @PulsarReader(topics = ["someTopic"], startMessageId = "earliest")
    fun processMessage(content: String?) {
        // ...
    }

}
```

The `@PulsarReader` relies on a `PulsarReaderFactory` to create the underlying Pulsar reader. Spring Boot auto-configuration provides this reader factory which can be customized by setting any of the `spring.pulsar.reader.*` prefixed application properties.

If you need more control over the reader factory configuration, consider registering one or more `ReaderBuilderCustomizer` beans. These customizers are applied to all readers created by the factory, and therefore all `@PulsarReader` instances. You can also customize a single listener by setting the `readerCustomizer` attribute of the `@PulsarReader` annotation.

10.4.9. Reading a Message Reactively

When the Apache Pulsar infrastructure is present and the Reactive auto-configuration is activated, Spring's `ReactivePulsarReaderFactory` is provided, and you can use it to create a reader in order to read messages in a reactive fashion. The following component creates a reader using the provided factory and reads a single message from 5 minutes ago from the `someTopic` topic:

Java

```
import java.time.Instant;
import java.util.List;

import org.apache.pulsar.client.api.Message;
import org.apache.pulsar.client.api.Schema;
import org.apache.pulsar.reactive.client.api.StartAtSpec;
import reactor.core.publisher.Mono;

import
org.springframework.pulsar.reactive.core.ReactiveMessageReaderBuilderCustomizer;
import org.springframework.pulsar.reactive.core.ReactivePulsarReaderFactory;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final ReactivePulsarReaderFactory<String> pulsarReaderFactory;

    public MyBean(ReactivePulsarReaderFactory<String> pulsarReaderFactory) {
        this.pulsarReaderFactory = pulsarReaderFactory;
    }

    public void someMethod() {
        ReactiveMessageReaderBuilderCustomizer<String> readerBuilderCustomizer =
(readerBuilder) -> readerBuilder
            .topic("someTopic")
            .startAtSpec(StartAtSpec.ofInstant(Instant.now().minusSeconds(5)));
        Mono<Message<String>> message = this.pulsarReaderFactory
            .createReader(Schema.STRING, List.of(readerBuilderCustomizer))
            .readOne();
        // ...
    }

}
```

Kotlin

```
import org.apache.pulsar.client.api.Schema
import org.apache.pulsar.reactive.client.api.ReactiveMessageReaderBuilder
import org.apache.pulsar.reactive.client.api.StartAtSpec
import org.springframework.pulsar.reactive.core.ReactiveMessageReaderBuilderCustomizer
import org.springframework.pulsar.reactive.core.ReactivePulsarReaderFactory
import org.springframework.stereotype.Component
import java.time.Instant

@Component
class MyBean(private val pulsarReaderFactory: ReactivePulsarReaderFactory<String>) {

    fun someMethod() {
        val readerBuilderCustomizer = ReactiveMessageReaderBuilderCustomizer {
            readerBuilder: ReactiveMessageReaderBuilder<String> ->
                readerBuilder
                    .topic("someTopic")
                    .startAtSpec(StartAtSpec.ofInstant(Instant.now().minusSeconds(5)))
        }
        val message = pulsarReaderFactory
                .createReader(Schema.STRING, listOf(readerBuilderCustomizer))
                .readOne()
        // ...
    }

}
```

Spring Boot auto-configuration provides this reader factory which can be customized by setting any of the `spring.pulsar.reader.*` prefixed application properties.

If you need more control over the reader factory configuration, consider passing in one or more `ReactiveMessageReaderBuilderCustomizer` instances when using the factory to create a reader.

If you need more control over the reader factory configuration, consider registering one or more `ReactiveMessageReaderBuilderCustomizer` beans. These customizers are applied to all created readers. You can also pass one or more `ReactiveMessageReaderBuilderCustomizer` when creating a reader to only apply the customizations to the created reader.

TIP For more details on any of the above components and to discover other available features, see the Spring for Apache Pulsar reference documentation.

10.4.10. Additional Pulsar Properties

The properties supported by auto-configuration are shown in the "Integration Properties" section of the Appendix. Note that, for the most part, these properties (hyphenated or camelCase) map directly to the Apache Pulsar configuration properties. See the Apache Pulsar documentation for details.

Only a subset of the properties supported by Pulsar are available directly through the `PulsarProperties` class. If you wish to tune the auto-configured components with additional properties that are not directly supported, you can use the customizer supported by each aforementioned component.

10.5. RSocket

RSocket is a binary protocol for use on byte stream transports. It enables symmetric interaction models through async message passing over a single connection.

The `spring-messaging` module of the Spring Framework provides support for RSocket requesters and responders, both on the client and on the server side. See the RSocket section of the Spring Framework reference for more details, including an overview of the RSocket protocol.

10.5.1. RSocket Strategies Auto-configuration

Spring Boot auto-configures an `RSocketStrategies` bean that provides all the required infrastructure for encoding and decoding RSocket payloads. By default, the auto-configuration will try to configure the following (in order):

1. CBOR codecs with Jackson
2. JSON codecs with Jackson

The `spring-boot-starter-rsocket` starter provides both dependencies. See the Jackson support section to know more about customization possibilities.

Developers can customize the `RSocketStrategies` component by creating beans that implement the `RSocketStrategiesCustomizer` interface. Note that their `@Order` is important, as it determines the order of codecs.

10.5.2. RSocket server Auto-configuration

Spring Boot provides RSocket server auto-configuration. The required dependencies are provided by the `spring-boot-starter-rsocket`.

Spring Boot allows exposing RSocket over WebSocket from a WebFlux server, or standing up an independent RSocket server. This depends on the type of application and its configuration.

For WebFlux application (that is of type `WebApplicationType.REACTIVE`), the RSocket server will be plugged into the Web Server only if the following properties match:

Properties

```
spring.rsocket.server.mapping-path=/rsocket
spring.rsocket.server.transport=websocket
```

Yaml

```
spring:
  rsocket:
    server:
      mapping-path: "/rsocket"
      transport: "websocket"
```

WARNING	Plugging RSocket into a web server is only supported with Reactor Netty, as RSocket itself is built with that library.

Alternatively, an RSocket TCP or websocket server is started as an independent, embedded server. Besides the dependency requirements, the only required configuration is to define a port for that server:

Properties

```
spring.rsocket.server.port=9898
```

Yaml

```
spring:
  rsocket:
    server:
      port: 9898
```

10.5.3. Spring Messaging RSocket support

Spring Boot will auto-configure the Spring Messaging infrastructure for RSocket.

This means that Spring Boot will create a `RSocketMessageHandler` bean that will handle RSocket requests to your application.

10.5.4. Calling RSocket Services with RSocketRequester

Once the `RSocket` channel is established between server and client, any party can send or receive requests to the other.

As a server, you can get injected with an `RSocketRequester` instance on any handler method of an RSocket `@Controller`. As a client, you need to configure and establish an RSocket connection first. Spring Boot auto-configures an `RSocketRequester.Builder` for such cases with the expected codecs and applies any `RSocketConnectorConfigurer` bean.

The `RSocketRequester.Builder` instance is a prototype bean, meaning each injection point will provide you with a new instance . This is done on purpose since this builder is stateful and you should not create requesters with different setups using the same instance.

The following code shows a typical example:

Java

```
import reactor.core.publisher.Mono;

import org.springframework.messaging.rsocket.RSocketRequester;
import org.springframework.stereotype.Service;

@Service
public class MyService {

    private final RSocketRequester rsocketRequester;

    public MyService(RSocketRequester.Builder rsocketRequesterBuilder) {
        this.rsocketRequester = rsocketRequesterBuilder.tcp("example.org", 9898);
    }

    public Mono<User> someRSocketCall(String name) {
        return
this.rsocketRequester.route("user").data(name).retrieveMono(User.class);
    }

}
```

Kotlin

```
import org.springframework.messaging.rsocket.RSocketRequester
import org.springframework.stereotype.Service
import reactor.core.publisher.Mono

@Service
class MyService(rsocketRequesterBuilder: RSocketRequester.Builder) {

    private val rsocketRequester: RSocketRequester

    init {
        rsocketRequester = rsocketRequesterBuilder.tcp("example.org", 9898)
    }

    fun someRSocketCall(name: String): Mono<User> {
        return rsocketRequester.route("user").data(name).retrieveMono(
            User::class.java
        )
    }

}
```

10.6. Spring Integration

Spring Boot offers several conveniences for working with Spring Integration, including the `spring-`

`boot-starter-integration` “Starter”. Spring Integration provides abstractions over messaging and also other transports such as HTTP, TCP, and others. If Spring Integration is available on your classpath, it is initialized through the `@EnableIntegration` annotation.

Spring Integration polling logic relies on the auto-configured `TaskScheduler`. The default `PollerMetadata` (poll unbounded number of messages every second) can be customized with `spring.integration.poller.*` configuration properties.

Spring Boot also configures some features that are triggered by the presence of additional Spring Integration modules. If `spring-integration-jmx` is also on the classpath, message processing statistics are published over JMX. If `spring-integration-jdbc` is available, the default database schema can be created on startup, as shown in the following line:

Properties

```
spring.integration.jdbc.initialize-schema=always
```

Yaml

```
spring:
  integration:
    jdbc:
      initialize-schema: "always"
```

If `spring-integration-rsocket` is available, developers can configure an RSocket server using `"spring.rsocket.server.*"` properties and let it use `IntegrationRSocketEndpoint` or `RSocketOutboundGateway` components to handle incoming RSocket messages. This infrastructure can handle Spring Integration RSocket channel adapters and `@MessageMapping` handlers (given `"spring.integration.rsocket.server.message-mapping-enabled"` is configured).

Spring Boot can also auto-configure an `ClientRSocketConnector` using configuration properties:

Properties

```
# Connecting to a RSocket server over TCP
spring.integration.rsocket.client.host=example.org
spring.integration.rsocket.client.port=9898
```

Yaml

```
# Connecting to a RSocket server over TCP
spring:
  integration:
    rsocket:
      client:
        host: "example.org"
        port: 9898
```

Properties

```
# Connecting to a RSocket Server over WebSocket
spring.integration.rsocket.client.uri=ws://example.org
```

Yaml

```
# Connecting to a RSocket Server over WebSocket
spring:
  integration:
    rsocket:
      client:
        uri: "ws://example.org"
```

See the `IntegrationAutoConfiguration` and `IntegrationProperties` classes for more details.

10.7. WebSockets

Spring Boot provides WebSockets auto-configuration for embedded Tomcat, Jetty, and Undertow. If you deploy a war file to a standalone container, Spring Boot assumes that the container is responsible for the configuration of its WebSocket support.

Spring Framework provides rich WebSocket support for MVC web applications that can be easily accessed through the `spring-boot-starter-websocket` module.

WebSocket support is also available for reactive web applications and requires to include the WebSocket API alongside `spring-boot-starter-webflux`:

```
<dependency>
    <groupId>jakarta.websocket</groupId>
    <artifactId>jakarta.websocket-api</artifactId>
</dependency>
```

10.8. What to Read Next

The next section describes how to enable IO capabilities in your application. You can read about caching, mail, validation, rest clients and more in this section.

Chapter 11. IO

Most applications will need to deal with input and output concerns at some point. Spring Boot provides utilities and integrations with a range of technologies to help when you need IO capabilities. This section covers standard IO features such as caching and validation as well as more advanced topics such as scheduling and distributed transactions. We will also cover calling remote REST or SOAP services and sending email.

11.1. Caching

The Spring Framework provides support for transparently adding caching to an application. At its core, the abstraction applies caching to methods, thus reducing the number of executions based on the information available in the cache. The caching logic is applied transparently, without any interference to the invoker. Spring Boot auto-configures the cache infrastructure as long as caching support is enabled by using the `@EnableCaching` annotation.

NOTE	Check the relevant section of the Spring Framework reference for more details.

In a nutshell, to add caching to an operation of your service add the relevant annotation to its method, as shown in the following example:

Java

```
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Component;

@Component
public class MyMathService {

    @Cacheable("piDecimals")
    public int computePiDecimal(int precision) {
        ...
    }

}
```