33. 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' and Spring Boot also provides auto-configuration options for `RabbitTemplate` and RabbitMQ. There is also support for STOMP messaging natively in Spring WebSocket and Spring Boot has support for that through starters and a small amount of auto-configuration.

33.1 JMS

The `javax.jms.ConnectionFactory` interface provides a standard method of creating a `javax.jms.Connection` for interacting with a JMS broker. Although Spring needs a `ConnectionFactory` to work with JMS, you generally won't need to use it directly yourself and you 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.

33.1.1 ActiveMQ support

Spring Boot can also configure a `ConnectionFactory` when it detects that ActiveMQ is available on the classpath. If the broker is present, an embedded broker is started and configured automatically (as long as no broker URL is specified through configuration).

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

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

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

33.1.2 Artemis support

Apache Artemis was formed in 2015 when HornetQ was donated to the Apache Foundation. All the features listed in the Section 33.1.3, "HornetQ support" section below can be applied to Artemis. Simply replace `spring.hornetq.*` properties with `spring.artemis.*` and use `spring-boot-starter-artemis` instead of `spring-boot-starter-hornetq`. If you want to embed Artemis, make sure to add `org.apache.activemq:artemis-jms-server` to the dependencies of your application.

You should not try and use Artemis and HornetQ and the same time.

33.1.3 HornetQ support

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

If you are using `spring-boot-starter-hornetq` the necessary dependencies to connect to an existing HornetQ instance are provided, as well as the Spring infrastructure to integrate with JMS. Adding `org.hornetq:hornetq-jms-server` to your application allows you to use the embedded mode.

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

```
spring.hornetq.mode=native
spring.hornetq.host=192.168.1.210
```

When embedding the broker, you can choose if you want to enable persistence, and the list of 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.hornetq.jms.server.config.JMSQueueConfiguration` or `org.hornetq.jms.server.config.TopicConfiguration`, for advanced queue and topic configurations respectively.

See `HornetQProperties` for more of the supported options.

No JNDI lookup is involved at all and destinations are resolved against their names, either using the 'name' attribute in the HornetQ configuration or the names provided through configuration.

33.1.4 Using a JNDI ConnectionFactory

If you are running your application in an Application Server Spring Boot will attempt to locate a JMS `ConnectionFactory` using JNDI. By default the locations `java:/JmsXA` and `java:/XAConnectionFactory` will be checked. You can use the `spring.jms.jndi-name` property if you need to specify an alternative location:

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

33.1.5 Sending a message

Spring's `JmsTemplate` is auto-configured and you can autowire it directly into your own beans:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final JmsTemplate jmsTemplate;

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

    // ...
```

`JmsMessagingTemplate` can be injected in a similar manner.

33.1.6 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.

The default factory is transactional by default. If you are running in an infrastructure where a `JtaTransactionManager` is present, it will be associated to the listener container by default. If not, the `sessionTransacted` flag will be 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 will make sure 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:

```
@Component
public class MyBean {

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

}
```

Check 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 exposes another factory that uses a specific `MessageConverter`:

```
@Configuration
static class JmsConfiguration {

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

}
```

That you can use in any `@JmsListener`-annotated method as follows:

```
@Component
public class MyBean {

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

}
```

33.2 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.

33.2.1 RabbitMQ support

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

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

See `RabbitProperties` for more of the supported options.

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

33.2.2 Sending a message

Spring's `AmqpTemplate` and `AmqpAdmin` are auto-configured and you can autowire them directly into your own beans:

```
import org.springframework.amqp.core.AmqpAdmin;
import org.springframework.amqp.core.AmqpTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class MyBean {

    private final AmqpAdmin amqpAdmin;
    private final AmqpTemplate amqpTemplate;

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

    // ...

}
```

`RabbitMessagingTemplate` can be injected in a similar manner.

Any `org.springframework.amqp.core.Queue` that is defined as a bean will be automatically used to declare a corresponding queue on the RabbitMQ instance if necessary.

33.2.3 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 one is configured automatically.

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

```
@Component
public class MyBean {

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

}
```

Check 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` that you can use to initialize a `SimpleRabbitListenerContainerFactory` with the same settings as the one that is auto-configured.

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

```
@Configuration
static class RabbitConfiguration {

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

}
```

That you can use in any `@RabbitListener`-annotated method as follows:

```
@Component
public class MyBean {

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

}
```

Prev Up Next

32. Caching Home 34. Sending email