< ALL GUIDES

Messaging with RabbitMQ

This guide walks you through the process of creating a Spring Boot application that publishes and subscribes to a RabbitMQ AMQP server.

What You Will Build

You will build an application that publishes a message by using Spring AMQP's `RabbitTemplate` and subscribes to the message on a POJO by using `MessageListenerAdapter`.

What You Need

- About 15 minutes
- A favorite text editor or IDE
- Java 17 or later

How to Complete This Guide

Like most Spring Getting Started guides you can start from scratch and complete each step, or you can jump straight to the solution, by viewing the code in this repository.

To **see the end result in your local environment**, you can do one of the following:

- Download and unzip the source repository for this guide
- Clone the repository using Git: `git clone https://github.com/spring-guides/gs-messaging-rabbitmq.git`

Get the Code

Go To Repo

Setting up the RabbitMQ Broker

Before you can build your messaging application, you need to set up a server to handle receiving and sending messages. This guide assumes that you use Spring Boot Docker Compose support. A prerequisite of this approach is that your development machine has a Docker environment, such as Docker Desktop, available. Add a dependency `spring-boot-docker-compose` that does the following:

- Search for a `compose.yml` and other common compose filenames in your working directory
- Call `docker compose up` with the discovered `compose.yml`
- Create service connection beans for each supported container
- Call `docker compose stop` when the application is shutdown

To use Docker Compose support, you need only follow this guide. Based on the dependencies you pull in, Spring Boot finds the correct `compose.yml` file and start your Docker container when you run your application.

If you choose to run the RabbitMQ server yourself instead of using Spring Boot Docker Compose support, you have a few options:

- Download the server and manually run it
- Install with Homebrew, if you use a Mac
- Manually run the `compose.yaml` file with `docker-compose up`

If you go with any of these alternate approaches, you should remove the `spring-boot-docker-compose` dependency from the Maven or Gradle build file. You will also need to add configuration to an `application.properties` file, as described in greater detail in the Preparing to Build the Application section. As mentioned earlier, this guide assumes that you use Docker Compose support in Spring Boot, so additional changes to `application.properties` are not required at this point.

Starting with Spring Initializr

You can use this pre-initialized project and click Generate to download a ZIP fi[illegible]s project is configured to fit the examples in this guide.

1. Navigate to start.spring.io. This service pulls in all the dependencies you need for an application and does most of the setup for you.
2. Choose either Gradle or Maven and the language you want to use. This guide assumes that you chose Java.
3. Click **Dependencies** and select **Spring for RabbitMQ** and **Docker Compose Support**.
4. Click **Generate**.
5. Download the resulting ZIP file, which is an archive of an application that is configured with your choices.

> If your IDE has the Spring Initializr integration, you can complete this process from your IDE.

Create a RabbitMQ Message Receiver

With any messaging-based application, you need to create a receiver that responds to published messages. The following listing (from `src/main/java/com/example/messagingrabbitmq/Receiver.java`) shows how to do so:

```
package com.example.messagingrabbitmq;

import java.util.concurrent.CountDownLatch;
import org.springframework.stereotype.Component;

@Component
public class Receiver {

  private CountDownLatch latch = new CountDownLatch(1);

  public void receiveMessage(String message) {
    System.out.println("Received <" + message + ">");
    latch.countDown();
  }

     ic CountDownLatch getLatch() {
}
```

The `Receiver` is a POJO that defines a method for receiving messages. When you register it to receive messages, you can name it anything you want.

> For convenience, this POJO also has a `CountDownLatch`. This lets it signal that the message has been received. This is something you are not likely to implement in a production application.

Register the Listener and Send a Message

Spring AMQP’s `RabbitTemplate` provides everything you need to send and receive messages with RabbitMQ. However, you need to:

- Configure a message listener container.
- Declare the queue, the exchange, and the binding between them.
- Configure a component to send some messages to test the listener.

> Spring Boot automatically creates a connection factory and a RabbitTemplate, reducing the amount of code you have to write.

You will use `RabbitTemplate` to send messages, and you will register a `Receiver` with the message listener container to receive messages. The connection factory drives both, letting them connect to the RabbitMQ server. The following listing (from `src/main/java/com/example/messagingrabbitmq/MessagingRabbitmqApplication.java`) shows how to create the application class:

```
package com.example.messagingrabbitmq;

import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.Queue;
import org.springframework.amqp.core.TopicExchange;
```

```
org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer;
import org.springframework.amqp.rabbit.listener.adapter.MessageListenerAdapter;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class MessagingRabbitmqApplication {

  static final String topicExchangeName = "spring-boot-exchange";

  static final String queueName = "spring-boot";

  @Bean
  Queue queue() {
    return new Queue(queueName, false);
  }

  @Bean
  TopicExchange exchange() {
    return new TopicExchange(topicExchangeName);
  }

  @Bean
  Binding binding(Queue queue, TopicExchange exchange) {
    return BindingBuilder.bind(queue).to(exchange).with("foo.bar.#");
  }

  @Bean
  SimpleMessageListenerContainer container(ConnectionFactory connectionFactory,
      MessageListenerAdapter listenerAdapter) {
    SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();
    container.setConnectionFactory(connectionFactory);
    container.setQueueNames(queueName);
    container.setMessageListener(listenerAdapter);
  @Bean
  MessageListenerAdapter listenerAdapter(Receiver receiver) {
    return new MessageListenerAdapter(receiver, "receiveMessage");
  }

  public static void main(String[] args) throws InterruptedException {
    SpringApplication.run(MessagingRabbitmqApplication.class, args).close();
  }

}
```

The `@SpringBootApplication` annotation offers a number of benefits, as described in the reference documentation.

The bean defined in the `listenerAdapter()` method is registered as a message listener in the container (defined in `container()`). It listens for messages on the `spring-boot` queue. Because the `Receiver` class is a POJO, it needs to be wrapped in the `MessageListenerAdapter`, where you specify that it invokes `receiveMessage`.

> JMS queues and AMQP queues have different semantics. For example, JMS sends queued messages to only one consumer. While AMQP queues do the same thing, AMQP producers do not send messages directly to queues. Instead, a message is sent to an exchange, which can go to a single queue or fan out to multiple queues, emulating the concept of JMS topics.

The message listener container and receiver beans are all you need to listen for messages. To send a message, you also need a Rabbit template.

The `queue()` method creates an AMQP queue. The `exchange()` method creates a topic exchange. The `binding()` method binds these two together, defining the behavior that occurs when `RabbitTemplate` publishes to an exchange.

Binding be declared as top-level Spring beans in order to be set up properly.

In this case, we use a topic exchange, and the queue is bound with a routing key of `foo.bar.#`, which means that any messages sent with a routing key that begins with `foo.bar.` are routed to the queue.

Send a Test Message

In this sample, test messages are sent by a `CommandLineRunner`, which also waits for the latch in the receiver and closes the application context. The following listing (from `src/main/java/com.example.messagingrabbitmq/Runner.java`) shows how it works:

COPY

```
package com.example.messagingrabbitmq;

import java.util.concurrent.TimeUnit;

import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class Runner implements CommandLineRunner {

  private final RabbitTemplate rabbitTemplate;
  private final Receiver receiver;

  public Runner(Receiver receiver, RabbitTemplate rabbitTemplate) {
    this.receiver = receiver;
    this.rabbitTemplate = rabbitTemplate;
  }

  @Override
  public void run(String... args) throws Exception {
    System.out.println("Sending message...");
    receiver.getLatch().await(10000, TimeUnit.MILLISECONDS);
  }


}
```

Notice that the template routes the message to the exchange with a routing key of `foo.bar.baz`, which matches the binding.

In tests, you can mock out the runner so that the receiver can be tested in isolation.

Run the Application

The `main()` method starts that process by creating a Spring application context. This starts the message listener container, which starts listening for messages. There is a `Runner` bean, which is then automatically run. It retrieves the `RabbitTemplate` from the application context and sends a `Hello from RabbitMQ!` message on the `spring-boot` queue. Finally, it closes the Spring application context, and the application ends.

You can run the main method through your IDE. Note that, if you have cloned the project from the solution repository, your IDE may look in the wrong place for the `compose.yaml` file. You can configure your IDE to look in the correct place or you could use the command line to run the application. The `./gradlew bootRun` and `./mvnw spring-boot:run` commands will launch the application and automatically find the compose.yaml file.

Preparing to Build the Application

To run the code without Spring Boot Docker Compose support, you need a version of RabbitMQ running locally to connect to. To do this, you can use Docker Compose, but you must first make two changes to the `compose.yaml` file. First, modify the `ports` entry in `compose.yaml` to be `'5672:5672'`. Second, add a `container_name`.

The `compose.yaml` should now be:

```
services:
  rabbitmq:
    container_name: 'guide-rabbit'
    image: 'rabbitmq:latest'
```

```
      - 'RABBITMQ_DEFAULT_USER=myuser'
    ports:
      - '5672:5672'
```

You can now run `docker-compose up` to start the RabbitMQ service. Now you should have an external RabbitMQ server that is ready to accept requests.

Additionally, you need to tell Spring how to connect to the RabbitMQ server (this was handled automatically with Spring Boot Docker Compose support). Add the following code to a new `application.properties` file in `src/main/resources`:

```
spring.rabbitmq.password=secret
spring.rabbitmq.username=myuser
```

Building the Application

This section describes different ways to run this guide:

1. Building and executing a JAR file
2. Building and executing a Docker container using Cloud Native Buildpacks

Regardless of how you choose to run the application, the output should be the same.

To run the application, you can package the application as an executable jar. The `./gradlew clean build` command compiles the application to an executable jar. You can then run the jar with the `java -jar build/libs/messaging-rabbitmq-0.0.1-SNAPSHOT.jar` command.

Alternatively, if you have a Docker environment available, you could create a Docker image directly from your Maven or Gradle plugin, using buildpacks. With Cloud Native Buildpacks, you can create Docker compatible images that you can run anywhere. Spring Boot includes buildpack support directly for both Maven and Gradle. This means you can type a single command and quickly get a sensible image into a locally running Docker daemon. To create a Docker image using Cloud Native Buildpacks, run the `./gradlew bootBuildImage` command. With a Docker environment enabled, you can run the application with the `docker run --network container:guide-rabbit docker.io/library/messaging-rabbitmq:0.0.1-SNAPSHOT` command.

existing network that our external container is using. You can find more information in the Docker documentation.

Regardless of how you chose to build and run the application, you should see the following output:

```
Sending message...
Received <Hello from RabbitMQ!>
```

Summary

Congratulations! You have just developed a simple publish-and-subscribe application with Spring and RabbitMQ. You can do more with Spring and RabbitMQ than what is covered here, but this guide should provide a good start.

See Also

Additional Spring AMQP Samples

The following guides may also be helpful:

- Messaging with Redis
- Messaging with JMS
- Building an Application with Spring Boot

Want to write a new guide or contribute to an existing one? Check out our contribution guidelines.

Reactive

Event Driven

Cloud

Web Applications

Serverless

Batch

Guides

Blog

Community

Events

Authors

Spring Consulting

Spring Academy For Teams

Spring Advisories

Thank You

Get the Spring newsletter

Stay connected with the Spring newsletter

SUBSCRIBE

Terms of Use • Privacy • Trademark Guidelines • Your California Privacy Rights