**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”.

**RabbitMQ support**

[RabbitMQ](https://www.rabbitmq.com/) 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 inapplication.properties:

spring.rabbitmq.host=localhost

spring.rabbitmq.port=5672

spring.rabbitmq.username=admin

spring.rabbitmq.password=secret

@SpringBootApplication

@EnableScheduling

@EnableRabbit

public class RabbitAmqpTutorialsApplication {

static final String queueName = "queueName";

@Bean

Queue queue() {

return new Queue(queueName, false);

}

// Configuring Connection Factory

@Bean

SimpleMessageListenerContainer container(ConnectionFactory connectionFactory,

MessageListenerAdapter listenerAdapter) {

SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();

container.setConnectionFactory(connectionFactory);

container.setQueueNames(queueName);

container.setMessageListener(listenerAdapter);

return container;

}

}

*// Sender*

package org.springframework.amqp.tutorials.tut1;

import org.springframework.amqp.core.Queue;

import org.springframework.amqp.rabbit.core.RabbitTemplate;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.scheduling.annotation.Scheduled;

public class Sender {

@Autowired

private RabbitTemplate template;

@Autowired

private Queue queue;

@Scheduled(fixedDelay = 1000, initialDelay = 500)

public void send() {

String message = "Hello World!";

this.template.convertAndSend(queue.getName(), message);

System.out.println(" [x] Sent '" + message + "'");

}

}

@RabbitListener(queues = "someQueue", **containerFactory="myFactory"**)

**public** **void** processMessage(String content) {

*// ...*

}

**Configuring Multiple DB’s**

To configure multiple DBs you need to configure multiple Entity Managers.

|  |
| --- |
| @Configuration |
|  | @EnableJpaRepositories( |
|  | entityManagerFactoryRef = "orderEntityManager", |
|  | transactionManagerRef = "orderTransactionManager", |
|  | basePackageClasses = Order.class) |
|  | public class OrderConfig { |

}

|  |
| --- |
| @Configuration |
|  | @EnableJpaRepositories( |
|  | entityManagerFactoryRef = "customerEntityManager", |
|  | transactionManagerRef = "customerTransactionManager", |
|  | basePackageClasses = Customer.class) |
|  | public class CustomerConfig { |

}

**Distributed Transactions with JTA**

Spring Boot supports distributed JTA transactions across multiple XA resources by using either an [Atomikos](http://www.atomikos.com/) or [Bitronix](https://github.com/bitronix/btm) embedded transaction manager. JTA transactions are also supported when deploying to a suitable Java EE Application Server.

When a JTA environment is detected, Spring’s JtaTransactionManager is used to manage transactions. Auto-configured JMS, DataSource, and JPA beans are upgraded to support XA transactions. You can use standard Spring idioms, such as @Transactional, to participate in a distributed transaction. If you are within a JTA environment and still want to use local transactions, you can set the spring.jta.enabled property to false to disable the JTA auto-configuration.

**Using an Atomikos Transaction Manager**

spring-boot-starter-jta-atomikos:

[Atomikos](https://www.atomikos.com/) is a popular open source transaction manager which can be embedded into your Spring Boot application. You can use thespring-boot-starter-jta-atomikos Starter to pull in the appropriate Atomikos libraries. Spring Boot auto-configures Atomikos and ensures that appropriate depends-on settings are applied to your Spring beans for correct startup and shutdown ordering.

By default, Atomikos transaction logs are written to a transaction-logs directory in your application’s home directory (the directory in which your application jar file resides). You can customize the location of this directory by setting a spring.jta.log-dir property in your application.properties file. Properties starting with spring.jta.atomikos.properties can also be used to customize the Atomikos UserTransactionServiceImp. See the [AtomikosProperties Javadoc](https://docs.spring.io/spring-boot/docs/2.0.2.RELEASE/api/org/springframework/boot/jta/atomikos/AtomikosProperties.html) for complete details.

|  |
| --- |
| [Note] |
| To ensure that multiple transaction managers can safely coordinate the same resource managers, each Atomikos instance must be configured with a unique ID. By default, this ID is the IP address of the machine on which Atomikos is running. To ensure uniqueness in production, you should configure the spring.jta.transaction-manager-id property with a different value for each instance of your application. |

<http://www.thedevpiece.com/design-patterns-that-every-developer-should-know/>

<http://www.thedevpiece.com/configuring-multiple-datasources-using-springboot-and-atomikos/>

<http://www.baeldung.com/hibernate-lazy-eager-loading>

<https://stackoverflow.com/questions/26601032/default-fetch-type-for-one-to-one-many-to-one-and-one-to-many-in-hibernate>

<http://www.baeldung.com/hibernate-date-time>

@Fetch(FetchMode.SELECT) (by default it is lazy loading)

It will generate 1+n query where n is number of parents.

@Fetch(FetchMode.SUBSELECT)

It will generate 1+1 query for parent and child.

@Fetch(FetchMode.JOIN) (It is eager loading, it will override even if it has FetchType.LAZY)

It will generate 1 query for all the parent and child

It will use left outer join internally.

**Mockito:**

<https://dzone.com/articles/spring-boot-unit-testing-and-mocking-with-mockito>

<https://walkingtree.tech/design-patterns-microservices/>

<https://walkingtree.tech/securing-microservices-oauth2/>

**What is OAuth2**?

OAuth 2 is an authorization framework, a security concept for rest API( Read as MicroService), about how you authorize a user to get access to a resource from your resource server by using token.

OAuth 2 has 4 different roles in this process.

* **Resource Owner**
* **Client** *@EnableOAuth2Client* RequestInterceptor
* **Authorization Server** @EnableAuthorizationServer
* **Resource Server** @EnableResourceServer both server and client

**Resource Owner**: Resource owner is the user,  who authorizes an application to access their account.

**Client**: Client is the application, which is used by the user to get resources from the resource server.

**Authorization Server**: Authorization server will issue access tokens by authenticating the user and obtain authorization grant.

Authorization server issues two type of tokens, access\_token and referesh\_token.

* The responsibility of access token is to access resource before it gets expired.
* The responsibility of Refresh Token is to request for a new access token when the access token is expired. An authorization grant is a credential representing the resource owner’s authorization (to access its protected resources) used by the client to obtain an access token.

The specification defines 4 grant types:

* **Authorization code**
* **Implicit**
* **Resource owner password credentials**
* **Client credentials**

<https://github.com/oktadeveloper/spring-boot-microservices-example.git>

<https://stackoverflow.com/questions/45859676/spring-cloud-feign-oauth2-request-interceptor-is-not-working>

<https://jmnarloch.wordpress.com/2015/10/14/spring-cloud-feign-oauth2-authentication/>