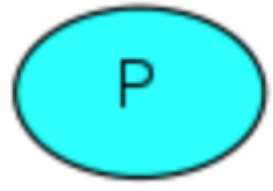
Agenda

1. What is rabbitmq introduction
2. Using rabbitmq with docker
3. AMQP
4. Messaging models

Introduction Rabbitmq

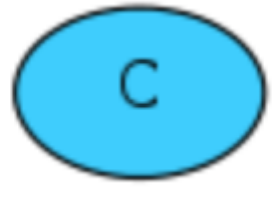
1. RabbitMQ is a message broker: it accepts and forwards messages
2. Analogy: post office:
   - RabbitMQ is a post box, a post office and a postman
3. Messages are binary blobs of data

Messaging Jargon



- producer

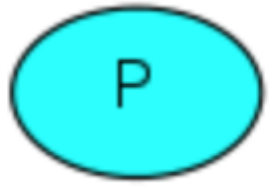
- queue



- consumer

Producer

- Producing means sending
- A program that sends messages is a producer



Queue

- Essentially a large message buffer, analogous to a post box which lives inside RabbitMQ
- Bound by the host's memory & disk limits
- Messages flow through RabbitMQ and your applications
  - But they can only be stored inside a queue
- Many producers can send messages to one queue
- Many consumers can try to receive data from one queue

Consumer

- has a similar meaning to receiver
- is a program that mostly waits to receive messages

- producer, consumer, and broker usually are on distinct hosts

Hello world of messaging

- Exchange messages via a queue
- Use springboot
- start.spring.io
- SELECT AMQP dependency

What to achieve

- P represents the producer and C the consumer
- The box in the middle is a queue
- P and C will both be implemented as a springBoot application

Docker to the rescue

1. Use rabbitmq docker image
2. Issue:
    1. docker run -d --hostname my-rabbit --name some-rabbit -p 5672:5672 -p 15672:15672 rabbitmq:3-management
3. Connect to the broker:
    1. at port 5672
    2. management console at port 15672

Building the consumer

- a.k.a. receiver

```
@RabbitListener(queues = "hello")
public class Demo1Receiver {

    @RabbitHandler
    public void receive(String in) {
        System.out.println(" [x] Received '" + in + "'");
    }
}
```

Building consumer part 2

- necessary configuration

```
@Configuration
public class Demo1ConsumerConfig {

    @Bean
    public Queue hello() {
        return new Queue("hello");
    }

    @Bean
    public Demo1Receiver receiver() {
        return new Demo1Receiver();
    }

}
```

Setup nuts and bolts

- create a SpringBoot entry point:

```
@SpringBootApplication
public class RabbitAmqpApplication {

    @Bean
    public CommandLineRunner tutorial() {
        return new RabbitAmqpRunner();
    }

    public static void main(String[] args) throws Exception {
        SpringApplication.run(RabbitAmqpApplication.class, args);
    }
}
```

Setup nuts and bolts part 2

- configuration of project in src/main/resources: application.yml

```
server:
  port: 8000

spring:
  rabbitmq:
    host: 127.0.0.1
    port: 5672
    username: guest
    password: guest

logging:
  level:
    org:  FINE
```

Building the Producer

- The Producer/Sender details:

```
public class Demo1Sender {

    @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 + "'");
    }
}
```

Building the Producer part 2

- Configure the producer part

```
@Configuration
public class Demo1ProducerConfig {

    @Bean
    public Queue hello() {
        return new Queue("hello");
    }

    @Bean
    public Demo1Sender sender() {
        return new Demo1Sender();
    }
}
```

Nuts and bolts

- The nuts and bolts are identical to the consumer the configuration is:

```
server:
  port: 8080

spring:
  rabbitmq:
    host: 127.0.0.1
    port: 5672
    username: guest
    password: guest

logging:
  level:
    org:  FINE
```

The important pieces

- The sender uses the RabbitTemplate

```
@Autowired
private RabbitTemplate template;

@Autowired
private Queue queue;

@Scheduled(fixedDelay = 1000, initialDelay = 500)
public void send() {
    ...
    this.template.convertAndSend(queue.getName(), "my message");
    ...
}
```

The important pieces part 2

- The receiver/consumer

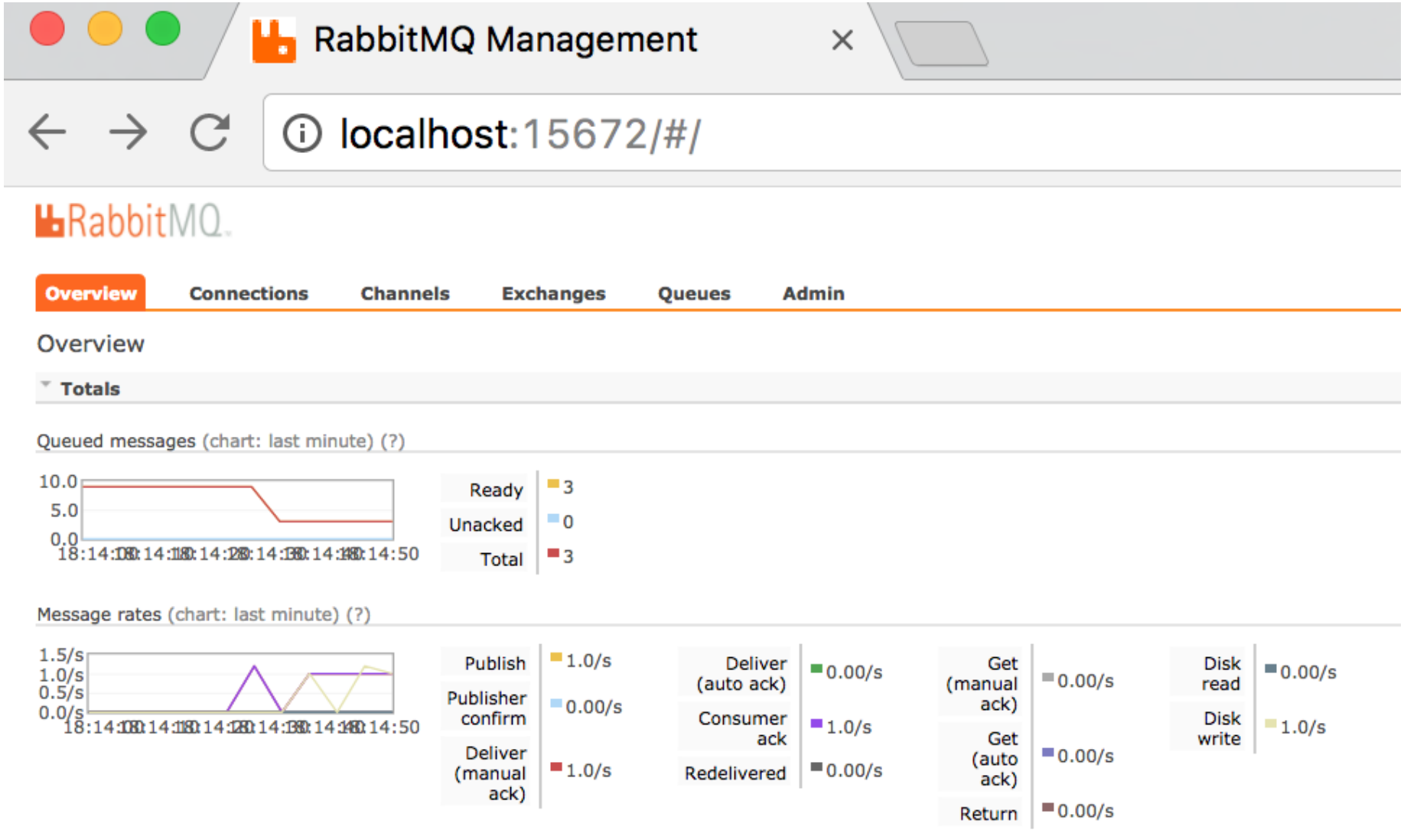
```
@RabbitListener(queues = "hello")
public class Demo1Receiver {

    @RabbitHandler
    public void receive(String in) {
        System.out.println(" [x] Received '" + in + "'");
    }
}
```

Running the 2 applications

```
Tomcat1 started on port(s): 8080 (http)
 [x] Sent 'Hello World!'
 [x] Sent 'Hello World!'

 Tomcat2 started on port(s): 8000 (http)
 [x] Received 'Hello World!'
 [x] Received 'Hello World!'
```

rabbitmanagement

- Admin webconsole to look at rabbitmq

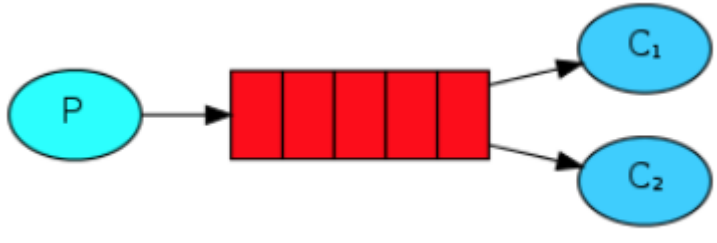

RabbitMQ Management
localhost:15672/#/
RabbitMQ
Overview
Connections
Channels
Exchanges
Queues
Admin
Overview
Totals
Queued messages (chart: last minute) (?)
10.0
5.0
0.0
18:14:50
Ready 3
Unacked 0
Total 3
Message rates (chart: last minute) (?)
1.5/s
1.0/s
0.5/s
0.0/s
18:14:50
Publish 1.0/s
Publisher confirm 0.00/s
Deliver (manual ack) 1.0/s
Deliver (auto ack) 0.00/s
Consumer ack 1.0/s
Redelivered 0.00/s
Get (manual ack) 0.00/s
Get (auto ack) 0.00/s
Return 0.00/s
Disk read 0.00/s
Disk write 1.0/s

Messaging patterns

1. Work queues
2. Publish/Subscribe
3. Routing
4. Topics
5. RPC

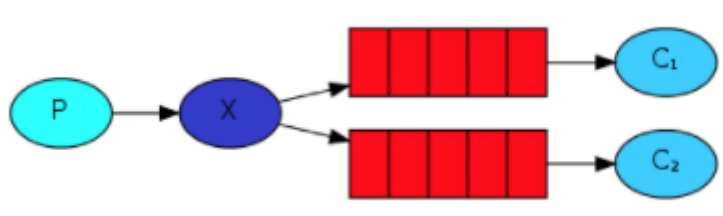
Work Queues

- Distributing tasks among workers
  - the competing consumers pattern



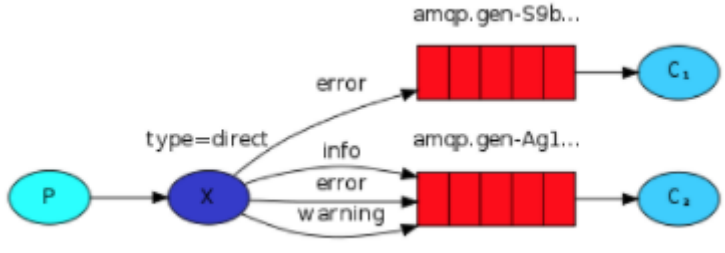
Publish/Subscribe

- Sending messages to many consumers at once



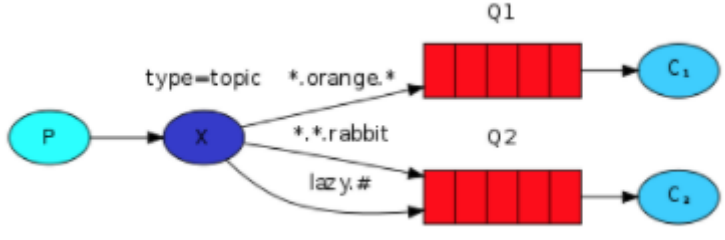
Routing

- Receiving messages selectively



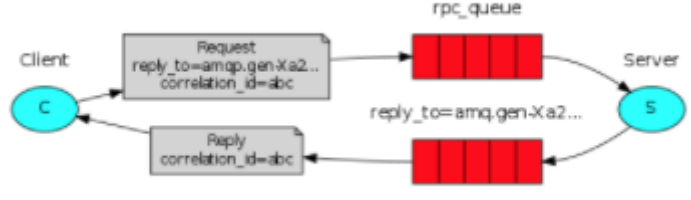
Topics

- Receiving messages based on a pattern (topics)



RPC

- Request/reply pattern

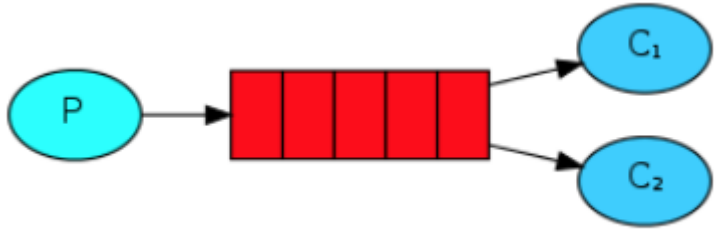


Messaging patterns

1. Work queues
2. Publish/Subscribe
3. Routing
4. Topics
5. RPC

Work Queues

- Distributing tasks among workers
  - the competing consumers pattern



Distribute tasks

- Main idea work queue, aka: Task Queues
  - distribute time-consuming tasks among multiple workers
  - avoid doing a resource-intensive task immediately and having to wait for it to complete -> schedule the task to be done later
  - encapsulate a task as a message and send it to a queue

Configuration Sender

```
@Configuration
public class Demo2SenderConfig {

    @Bean
    public Queue hello() {
        return new Queue("hello");
    }
    @Bean
    public Demo2Sender sender() {
        return new Demo2Sender();
    }
}
```

Sending message

```
public class Demo2Sender {
    @Autowired
    private RabbitTemplate template;

    @Autowired
    private Queue queue;

    @Scheduled(fixedDelay = 1000, initialDelay = 500)
    public void send() {
        //Constructing message
        template.convertAndSend(queue.getName(), message);
    }
}
```

Configuration Receiver

```
@Configuration
public class Demo2ReceiverConfig {

    @Bean
    public Queue hello() {
        return new Queue("hello");
    }
    @Bean
    public Demo2Receiver receiver1() {
        return new Demo2Receiver(1);
    }
    @Bean
    public Demo2Receiver receiver2() {
        return new Demo2Receiver(2);
    }
}
```

Listening to workqueue

```
@RabbitListener(queues = "hello")
public class Demo2Receiver {
    private final int instance;

    public Demo2Receiver(int i) {
        this.instance = i;
    }
    @RabbitHandler
    public void receive(String in)  {
        doWork(in);
   }
    private void doWork(String in)  {
        //working long and hard
    }
}
```

Behind the covers 1

- Message acknowledgment
  - a task can take time -> what if a consumers starts a task and dies only partly done
- Spring-amqp has a conservative approach to message acknowledgement
  - if listener throws exceptionthe container calls:

```
channel.basicReject(deliveryTag, requeue)
```

- Requeue is true by default or the listener throws an AmqpRejectAndDontRequeueException

Behind the covers 1

- The behaviour last slide is typically what you want
- After successfully processing the message the listener calls:

```
channel.basicAck()
```

- It's a common mistake to miss the basicAck
- Messages are redelivered when your client quits and RabbitMQ will eat more and more memory as it won't be able to release any unacked messages

Message durability

Property	default	Description
durable	true	When declareExchange is true the durable flag is set to this value
deliveryMode	PERSISTENT	PERSISTENT or NON_PERSISTENT to determine whether or not RabbitMQ should persist the messages

- Marking messages as persistent is no 100% guarantee that a message won't be lost
- There is still a short time window when RabbitMQ has accepted a message and hasn't saved it yet

Fair or Round-robin dispatch

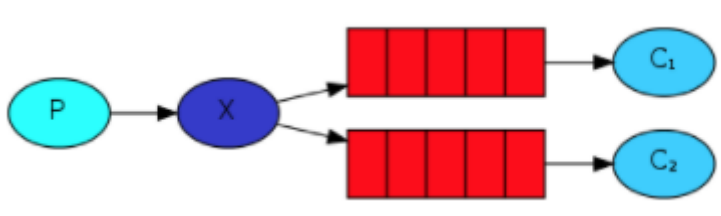
- Fair (default)
  - don't dispatch message(s) to worker until it processed and acknowledged the previous one
- Round-robin (non-default)
  - each is send to the next consumer, in sequence
  - on average every consumer will get the same number
  - blindly sends every n-th message to n-th consumer

Messaging patterns

1. Work queues
2. Publish/Subscribe
3. Routing
4. Topics
5. RPC

Publish/Subscribe

- Sending messages to many consumers at once

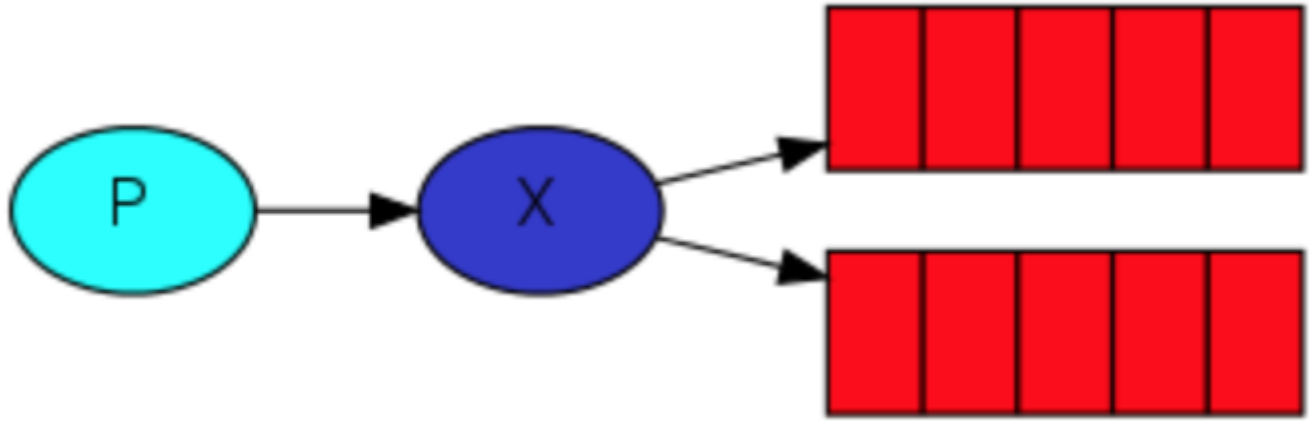


Publish/Subscribe

- also called the fanout pattern
- published messages are going to be broadcast to all the receivers

Exchanges

- The full messaging model in Rabbit includes an exchange
  - The producer never sends any messages directly to a queue
  - it can only send messages to an exchange



About exchanges

- it receives messages from producers
- it pushes them to queues
- must know exactly the rules what to do with a message
  - appended it to a particular queue? or
  - to many queues or not at all
- rules for that are defined by the exchange type

Exchange types

- Exchange types available
  1. direct
  2. topic
  3. headers
  4. fanout <- Take a look at this one

Fanout exchange

- Configuration of the sender

```
@Configuration
public class Demo3SenderConfig {

    @Bean
    public FanoutExchange fanout() {
        return new FanoutExchange("tut.fanout");
    }

    @Bean
    public Demo3Sender sender() {
        return new Demo3Sender();
    }
}
```

Sending to the exchange

```
public class Demo3Sender {

    @Autowired
    private RabbitTemplate template;

    @Autowired
    private FanoutExchange fanout;

    @Scheduled(fixedDelay = 1000, initialDelay = 500)
    public void send() {
        //create a message
        template.convertAndSend(fanout.getName(), "", message);
    }
}
```

The default exchange

- a default exchange, which is identified by the empty string ("")
- recall when sending messages "directly" to a queue:

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

@Override
public void convertAndSend(String routingKey, final Object object
    convertAndSend(this.exchange, routingKey, object, (Correlatio
}

private volatile String exchange = DEFAULT_EXCHANGE;

private static final String DEFAULT_EXCHANGE = "";
```

Configuring the receiver

```
@Configuration  public class Demo3ReceiverConfig {
    @Bean public FanoutExchange fo() {
        return new FanoutExchange("tut.fanout");
    }
    @Bean public Queue autoDeleteQueue1() {
        return new AnonymousQueue();
    }
    // idem @Bean public Queue autoDeleteQueue1() {...}
    @Bean
    public Binding binding1(FanoutExchange fo, Queue autoDeleteQu
        return BindingBuilder.bind(autoDeleteQueue1).to(fo);
    }
    // idem @Bean public Binding binding2(...) {...}
    @Bean public Demo3Receiver receiver() {
        return new Demo3Receiver();
    }
```

Temporary queues 1

- a queue name is important when sharing a queue between producers and consumers but not with fanout
- fanout

  1. hear about all messages, not just a subset
  2. only currently flowing messages are of interest not the old ones

- To solve that we need two things

Temporary queues 2

- How to get a temporarily queue:
- when connecting to Rabbit we need a fresh, empty queue, let the server choose a random queue name
- once disconnecting the consumer the queue should be automatically deleted
- These are the properties of an AnonymousQueue, which is a non-durable, exclusive, autodelete queue with a generated name

Defining 2 Temporary queues

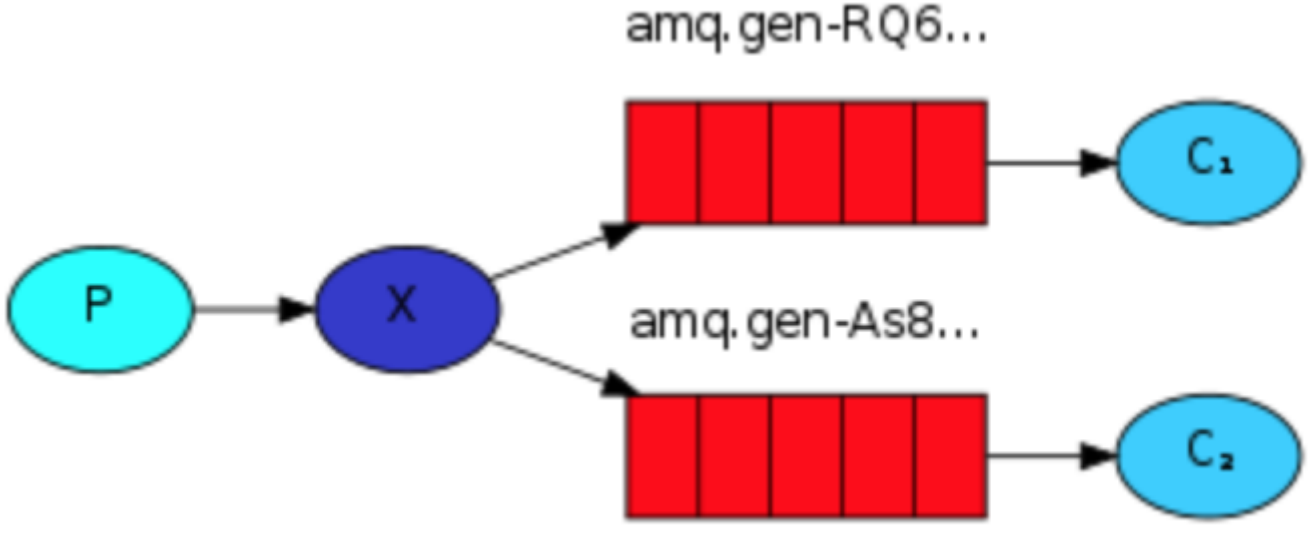
```
@Bean
public Queue autoDeleteQueue1() {
    return new AnonymousQueue();
}

@Bean
public Queue autoDeleteQueue2() {
    return new AnonymousQueue();
}
```

Binding Exchange to Queue

- The exchange needs to be told to send messages to our queue
- That relationship between exchange and a queue is called a binding

```
@Bean
public Binding binding1(FanoutExchange fanout, Queue autoDeleteQu
    return BindingBuilder.bind(autoDeleteQueue1).to(fanout);
}
```

Coping with random names



The Sender

```
public class Demo3Sender {
    @Autowired
    private RabbitTemplate template;
    @Autowired
    private FanoutExchange fanout;

    @Scheduled(fixedDelay = 1000, initialDelay = 500)
    public void send() {
        //Construct message
        //You has to send to a named exchange
        template.convertAndSend(fanout.getName(), "", message);
    }
}
```

The Receiver

- Get a message from a queue with a random name:

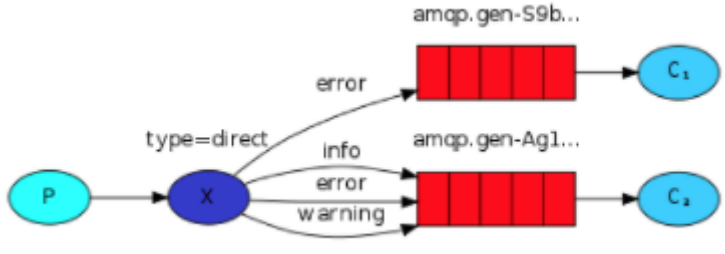
```
public class Demo3Receiver {
    @RabbitListener(queues = "#{autoDeleteQueue1.name}")
    public void receive1(String in){
        receive(in, 1);
    }
    @RabbitListener(queues = "#{autoDeleteQueue2.name}")
    public void receive2(String in){
        receive(in, 2);
    }
    public void receive(String in, int receiver){
        doWork(in);
    }
    private void doWork(String in) throws InterruptedException {
        //do something with the message
    }
}
```

Messaging patterns

1. Work queues
2. Publish/Subscribe
3. Routing
4. Topics
5. RPC

Routing

- Receiving messages selectively
- Subscribing to only a subset of the messages



Bindings

- is a relationship between an exchange and a queue
- read as: the queue is interested in messages from this exchange
- can take an extra routingKey parameter

RoutingKey

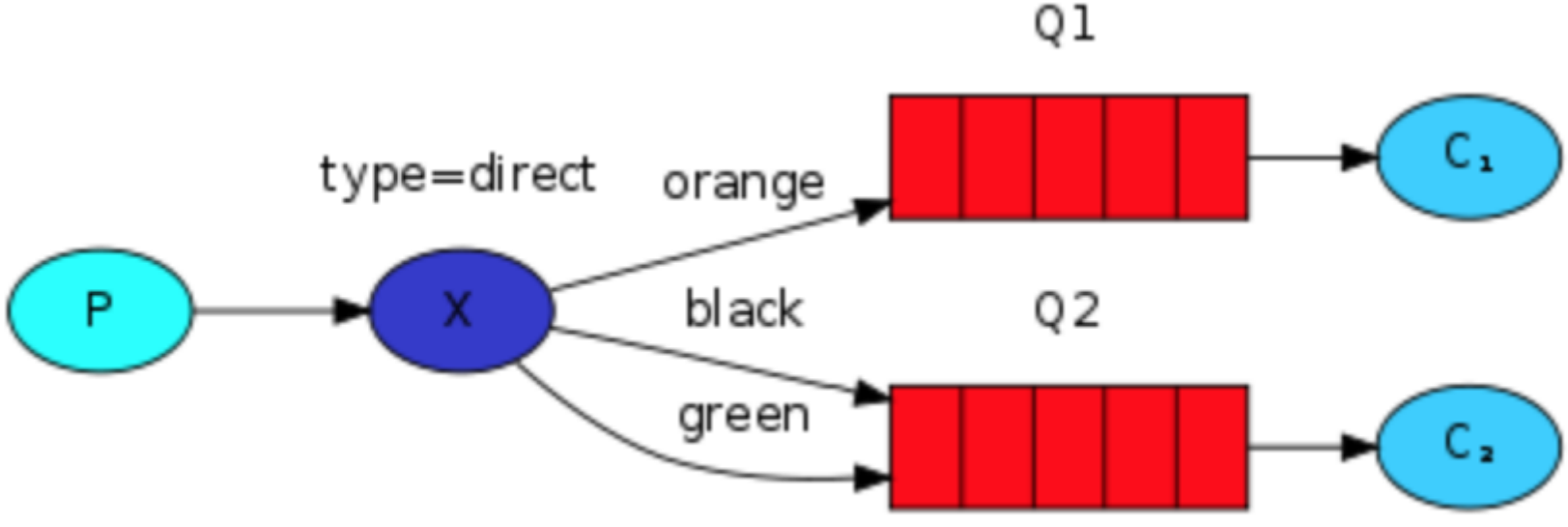
- bind queue "to" the exchange "with a routing key" as follows

```
@Bean
    public Binding binding1a(DirectExchange direct,
                             Queue autoDeleteQueue1) {
        return BindingBuilder
                .bind(autoDeleteQueue1)
                .to(direct)
                .with("orange");
    }
```

- meaning of a routing key depends on the exchange type

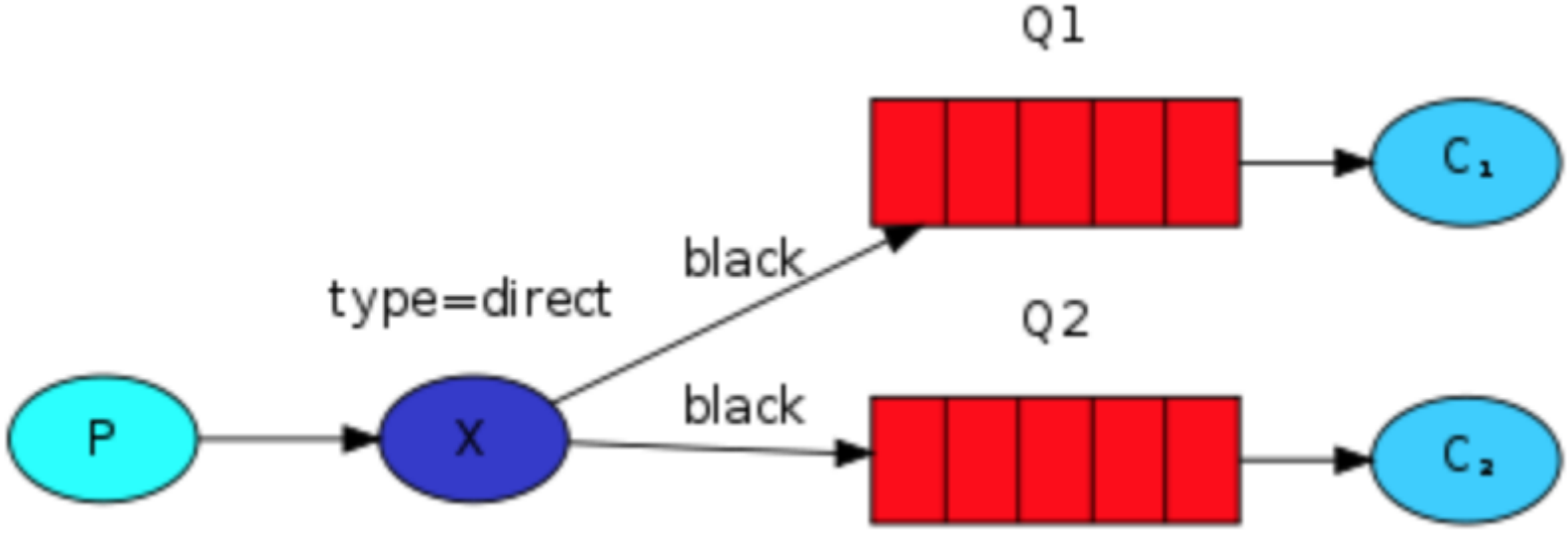
Filtering messages

- Use routing key to filter messages



Multiple bindings

- You can bind multiple queues with the same binding key



Java Configuration

```
@Configuration
public class Demo4ReceiverConfig {
    @Bean
    public DirectExchange direct() {
        return new DirectExchange("tut.direct");
    }
    @Bean //Create 2 queues
    public Queue autoDeleteQueue1() {
        return new AnonymousQueue();
    }
    @Bean //Create a couple of bindings to colours
    public Binding binding1a(DirectExchange direct,
                             Queue autoDeleteQueue1) {
        return BindingBuilder.bind(autoDeleteQueue1)
                             .to(direct).with("orange");
    }
```

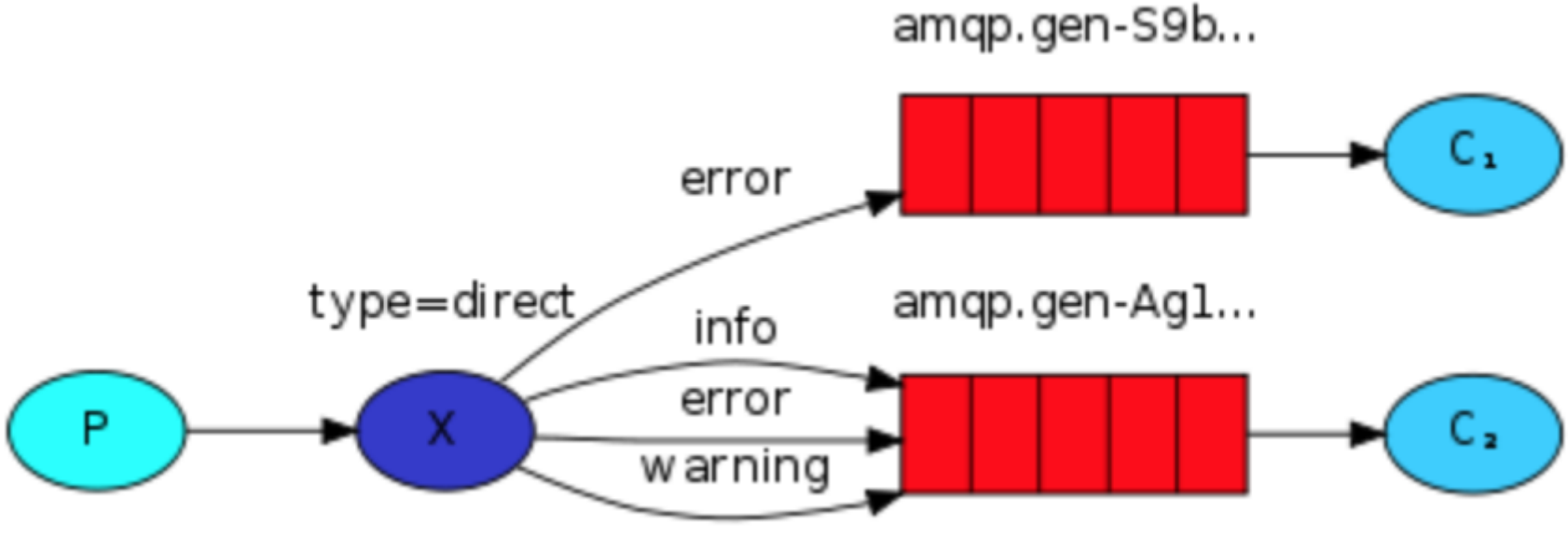
Create Sender

```
public class Demo4Sender {
    @Autowired
    private RabbitTemplate template;
    @Autowired
    private DirectExchange direct;
    private final String[] keys = {"orange", "black", "green"};
    @Scheduled(fixedDelay = 1000, initialDelay = 500)
    public void send() {
        //Create messages and couple it to a colour with key
        template.convertAndSend(direct.getName(), key, message);
        System.out.println(" [x] Sent '" + message + "'");
    }
}
```

Receiving messages

```
public class Demo4Receiver {
    @RabbitListener(queues = "#{autoDeleteQueue1.name}")
    public void receive1(String in) throws InterruptedException {
        receive(in, 1);
    }
    @RabbitListener(queues = "#{autoDeleteQueue2.name}")
    public void receive2(String in) throws InterruptedException {
        receive(in, 2);
    }
    public void receive(String in, int receiver) throws Interrupt
        doWork(in);
    }
    private void doWork(String in) throws InterruptedException {
        //act on message content
    }
}
```

Usecase: filter logging stream

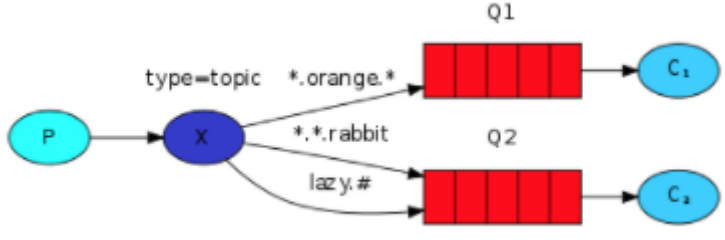


Messaging patterns

1. Work queues
2. Publish/Subscribe
3. Routing
4. Topics
5. RPC

Topics

- Receiving messages based on a pattern (topics)



Topics

- improving our messaging flexibility
- requirement: subscribe to queues not only based on the routing key, but also based on the source which produced the message
- this requirement can be implemented with topics

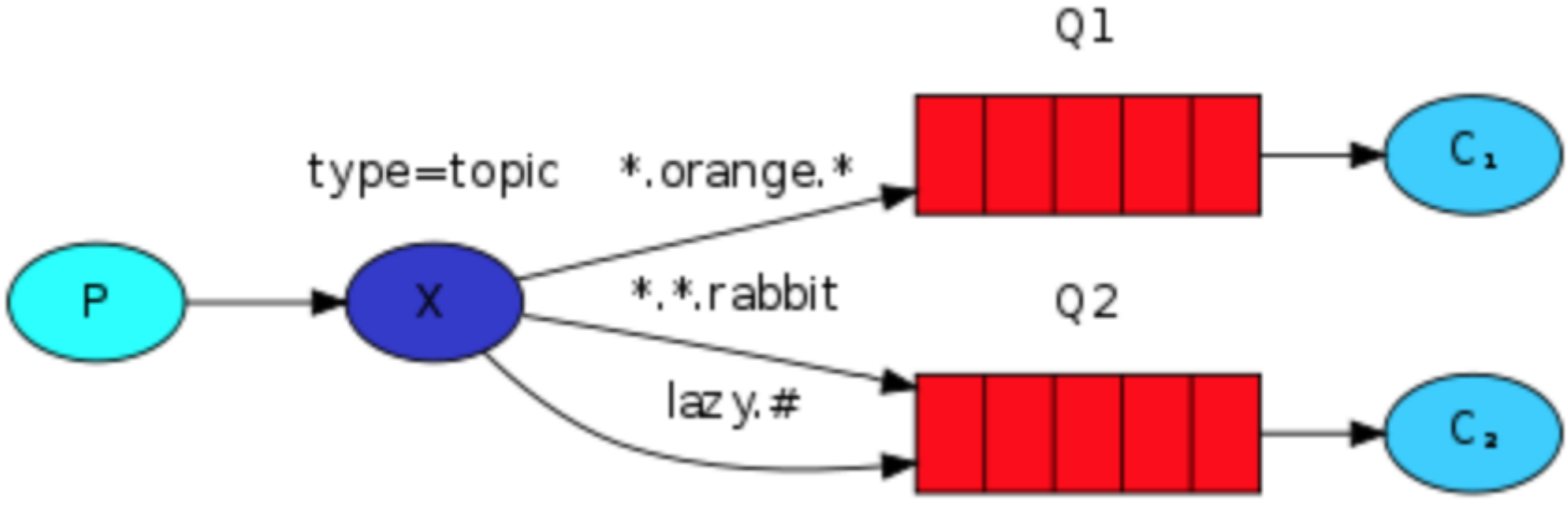
Topic exchange

- the routing_key must be a list of words, delimited by dots
- words usually specify some features connected to the message
- the routing key size is up to the limit of 255 bytes

The binding key

- must also be in the same form
- There are two important special cases for binding keys:
  1. "*" (star) can substitute for exactly one word
  2. "#" (hash) can substitute for zero or more words

Example



- messages are sent with routing key:
- "<speed>.<colour>.<species>"

Topic exchange

- powerful and can behave like other exchanges
- when a queue is bound with "#" (hash) binding key it will behave like in fanout exchange
- when no special characters "*" (star) and "#" (hash) are used the exchange will behave just like a direct exchange

Java Configuration

```
@Bean public Binding binding1a(TopicExchange topic,
                               Queue autoDeleteQueue1) {
    return BindingBuilder.bind(autoDeleteQueue1)
                 .to(topic).with("*.orange.*");
}
@Bean public Binding binding1b(TopicExchange topic,
                               Queue autoDeleteQueue1) {
    return BindingBuilder.bind(autoDeleteQueue1)
                 .to(topic).with("*.*.rabbit");
}
@Bean public Binding binding2a(TopicExchange topic,
                               Queue autoDeleteQueue2) {
    return BindingBuilder.bind(autoDeleteQueue2)
                    .to(topic).with("lazy.#");
}
```

The Sender

```
public class Demo5Sender {
    @Autowired private RabbitTemplate template;

    @Autowired  private TopicExchange topic;

    private final String[] keys = {"quick.orange.rabbit",
            "lazy.orange.elephant", "quick.orange.fox",
            "lazy.brown.fox", "lazy.pink.rabbit", "quick.brown.fo

    @Scheduled(fixedDelay = 1000, initialDelay = 500)
    public void send() {
       //Create message and use keys
        template.convertAndSend(topic.getName(), key, message);
    }
}
```