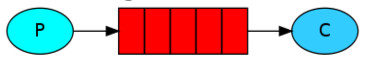
1 "Hello World!"

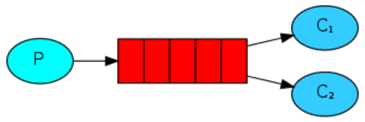
The simplest thing that does *something*



Python
Java
Ruby
PHP
C#
Javascript
Go
Elixir
Objective-C

2 Work queues

Distributing tasks among workers



Python
Java

Routing

(using Go RabbitMQ client)

In the **previous tutorial** we built a simple logging system. We were able to broadcast log messages to many receivers.

In this tutorial we're going to add a feature to it - we're going to make it possible to subscribe only to a subset of the messages. For example, we will be able to direct only critical error messages to the log file (to save disk space), while still being able to print all of the log messages on the console.

Prerequisites

This tutorial assumes RabbitMQ is **installed** and running on `localhost` on standard port (`5672`). In case you use a different host, port or credentials, connections settings would require adjusting.

Where to get help

If you're having trouble going through this tutorial you can **contact us** through the mailing list.

Bindings

In previous examples we were already creating bindings. You may recall code like:

```
err = ch.QueueBind(
  q.Name, // queue name
  "",     // routing key
  "logs", // exchange
  false,
  nil)
```

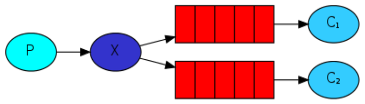
A binding is a relationship between an exchange and a queue. This can be simply read as: the queue is interested in messages from this exchange.

Bindings can take an extra `routing_key` parameter. To avoid the confusion with a `Channel.Publish` parameter we're going to call it a `binding key`. This is how we could create a binding with a key:

Ruby
PHP
C#
Javascript
Go
Elixir
Objective-C

3 Publish/Subscribe

Sending messages to many consumers at once



Python
Java
Ruby
PHP
C#
Javascript
Go
Elixir
Objective-C

4 Routing

Receiving messages selectively

```
err = ch.QueueBind(
  q.Name,    // queue name
  "black",   // routing key
  "logs",    // exchange
  false,
  nil)
```

The meaning of a binding key depends on the exchange type. The `fanout` exchanges, which we used previously, simply ignored its value.

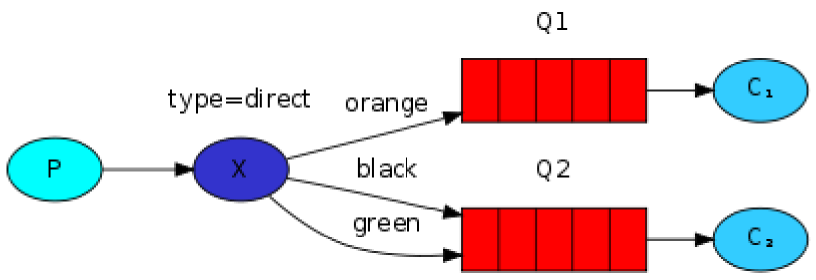
Direct exchange

Our logging system from the previous tutorial broadcasts all messages to all consumers. We want to extend that to allow filtering messages based on their severity. For example we may want the script which is writing log messages to the disk to only receive critical errors, and not waste disk space on warning or info log messages.

We were using a `fanout` exchange, which doesn't give us much flexibility - it's only capable of mindless broadcasting.

We will use a `direct` exchange instead. The routing algorithm behind a `direct` exchange is simple - a message goes to the queues whose `binding key` exactly matches the `routing key` of the message.

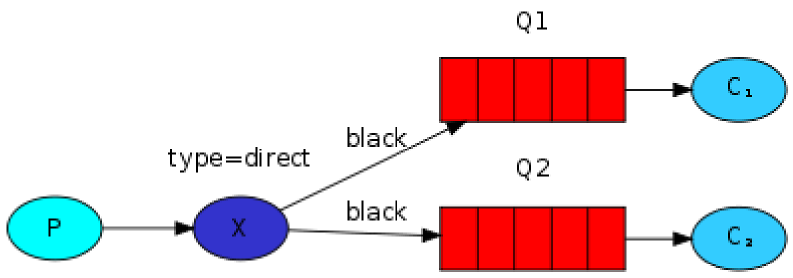
To illustrate that, consider the following setup:



In this setup, we can see the `direct` exchange `X` with two queues bound to it. The first queue is bound with binding key `orange`, and the second has two bindings, one with binding key `black` and the other one with `green`.

In such a setup a message published to the exchange with a routing key `orange` will be routed to queue `Q1`. Messages with a routing key of `black` or `green` will go to `Q2`. All other messages will be discarded.

Multiple bindings



It is perfectly legal to bind multiple queues with the same binding key. In our example we could add a binding between `X` and `Q1` with binding key `black`. In that case, the `direct` exchange will behave like `fanout` and will broadcast the message to all the matching queues. A message with routing key `black` will be delivered to both `Q1` and `Q2`.

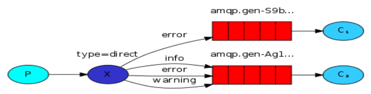
Emitting logs

We'll use this model for our logging system. Instead of `fanout` we'll send messages to a `direct` exchange. We will supply the log severity as a `routing key`. That way the receiving script will be able to select the severity it wants to receive. Let's focus on emitting logs first.

As always, we need to create an exchange first:

```
err = ch.ExchangeDeclare(
  "logs_direct", // name
  "direct",      // type
  true,          // durable
  false,         // auto-deleted
  false,         // internal
  false,         // no-wait
  nil,           // arguments
)
```

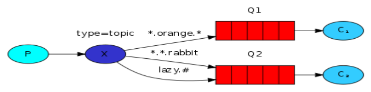
And we're ready to send a message:



Python
Java
Ruby
PHP
C#
Javascript
Go
Elixir
Objective-C

5 Topics

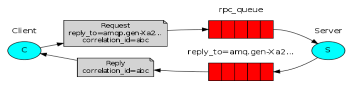
Receiving messages based on a pattern



Python
Java
Ruby
PHP
C#
Javascript
Go
Elixir
Objective-C

6 RPC

Remote procedure call implementation



Python
Java
Ruby
PHP
C#
Javascript
Go
Elixir

```
err = ch.ExchangeDeclare(
  "logs_direct", // name
  "direct",      // type
  true,          // durable
  false,         // auto-deleted
  false,         // internal
  false,         // no-wait
  nil,           // arguments
)
failOnError(err, "Failed to declare an exchange")

body := bodyFrom(os.Args)
err = ch.Publish(
  "logs_direct",         // exchange
  severityFrom(os.Args), // routing key
  false, // mandatory
  false, // immediate
  amqp.Publishing{
    ContentType: "text/plain",
    Body:        []byte(body),
  })
```

To simplify things we will assume that 'severity' can be one of 'info', 'warning', 'error'.

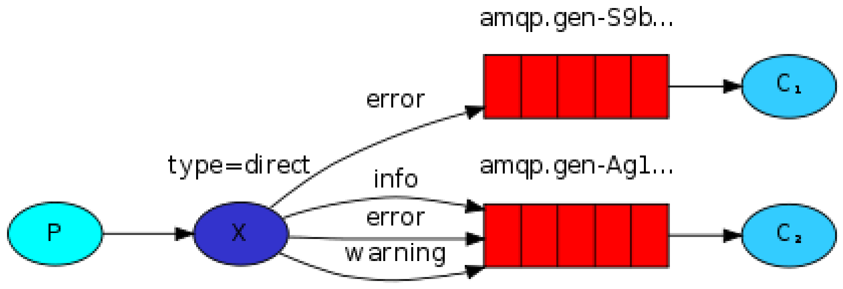
Subscribing

Receiving messages will work just like in the previous tutorial, with one exception - we're going to create a new binding for each severity we're interested in.

```
q, err := ch.QueueDeclare(
  "",    // name
  false, // durable
  false, // delete when usused
  true,  // exclusive
  false, // no-wait
  nil,   // arguments
)
failOnError(err, "Failed to declare a queue")

if len(os.Args) < 2 {
  log.Printf("Usage: %s [info] [warning] [error]", os.Args[0])
  os.Exit(0)
}
for _, s := range os.Args[1:] {
  log.Printf("Binding queue %s to exchange %s with routing key %s",
     q.Name, "logs_direct", s)
  err = ch.QueueBind(
    q.Name,        // queue name
    s,             // routing key
    "logs_direct", // exchange
    false,
    nil)
  failOnError(err, "Failed to bind a queue")
}
```

Putting it all together



The code for `emit_log_direct.go` script:

```
package main

import (
        "fmt"
        "log"
        "os"
        "strings"

        "github.com/streadway/amqp"
)
```

```
func failOnError(err error, msg string) {
        if err != nil {
                log.Fatalf("%s: %s", msg, err)
                panic(fmt.Sprintf("%s: %s", msg, err))
        }
}

func main() {
        conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/")
        failOnError(err, "Failed to connect to RabbitMQ")
        defer conn.Close()

        ch, err := conn.Channel()
        failOnError(err, "Failed to open a channel")
        defer ch.Close()

        err = ch.ExchangeDeclare(
                "logs_direct", // name
                "direct",      // type
                true,          // durable
                false,         // auto-deleted
                false,         // internal
                false,         // no-wait
                nil,           // arguments
        )
        failOnError(err, "Failed to declare an exchange")

        body := bodyFrom(os.Args)
        err = ch.Publish(
                "logs_direct",         // exchange
                severityFrom(os.Args), // routing key
                false, // mandatory
                false, // immediate
                amqp.Publishing{
                        ContentType: "text/plain",
                        Body:        []byte(body),
                })
        failOnError(err, "Failed to publish a message")

        log.Printf(" [x] Sent %s", body)
}
```

```go
func bodyFrom(args []string) string {
	var s string
	if (len(args) < 3) || os.Args[2] == "" {
		s = "hello"
	} else {
		s = strings.Join(args[2:], " ")
	}
	return s
}

func severityFrom(args []string) string {
	var s string
	if (len(args) < 2) || os.Args[1] == "" {
		s = "info"
	} else {
		s = os.Args[1]
	}
	return s
}
```

The code for `receive_logs_direct.go`:

```go
package main

import (
	"fmt"
	"log"
	"os"

	"github.com/streadway/amqp"
)

func failOnError(err error, msg string) {
	if err != nil {
		log.Fatalf("%s: %s", msg, err)
		panic(fmt.Sprintf("%s: %s", msg, err))
	}
}

func main() {
```

```
conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/")
failOnError(err, "Failed to connect to RabbitMQ")
defer conn.Close()

ch, err := conn.Channel()
failOnError(err, "Failed to open a channel")
defer ch.Close()

err = ch.ExchangeDeclare(
        "logs_direct", // name
        "direct",      // type
        true,          // durable
        false,         // auto-deleted
        false,         // internal
        false,         // no-wait
        nil,           // arguments
)
failOnError(err, "Failed to declare an exchange")

q, err := ch.QueueDeclare(
        "",    // name
        false, // durable
        false, // delete when usused
        true,  // exclusive
        false, // no-wait
        nil,   // arguments
)
failOnError(err, "Failed to declare a queue")

if len(os.Args) < 2 {
        log.Printf("Usage: %s [info] [warning] [error]", os.Args[0])
        os.Exit(0)
}
for _, s := range os.Args[1:] {
        log.Printf("Binding queue %s to exchange %s with routing key %s",
                q.Name, "logs_direct", s)
        err = ch.QueueBind(
                q.Name,        // queue name
                s,             // routing key
                "logs_direct", // exchange
                false,
                        nil)
                failOnError(err, "Failed to bind a queue")
        }

        msgs, err := ch.Consume(
                q.Name, // queue
                "",     // consumer
                true,   // auto ack
                false,  // exclusive
                false,  // no local
                false,  // no wait
                nil,    // args
        )
        failOnError(err, "Failed to register a consumer")

        forever := make(chan bool)

        go func() {
                for d := range msgs {
                        log.Printf(" [x] %s", d.Body)
                }
        }()

        log.Printf(" [*] Waiting for logs. To exit press CTRL+C")
        <-forever
}
```

If you want to save only 'warning' and 'error' (and not 'info') log messages to a file, just open a console and type:

```
$ go run receive_logs_direct.go warning error > logs_from_rabbit.log
```

If you'd like to see all the log messages on your screen, open a new terminal and do:

```
$ go run receive_logs_direct.go info warning error
 [*] Waiting for logs. To exit press CTRL+C
```

And, for example, to emit an `error` log message just type:

```
$ go run emit_log_direct.go error "Run. Run. Or it will explode."
 [x] Sent 'error':'Run. Run. Or it will explode.'
```

(Full source code for **(emit_log_direct.go source)** and **(receive_logs_direct.go source)**)

Move on to **tutorial 5** to find out how to listen for messages based on a pattern.