

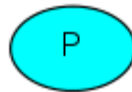
Utilizzare AMQP con RABBITMQ

Claudio Biancalana

Esercizio

Hello World (1)

- Precondizioni
 - In questa esercitazione si presuppone che RabbitMQ sia installato e in esecuzione su localhost su porta standard (5672). Nel caso in cui utilizzi un host, una porta o credenziali diversi, le impostazioni delle connessioni richiedono una modifica (che vedremo!).
- *Produrre* non significa altro che inviare. Un programma che invia messaggi è un *produttore* :



Esercizio

Hello World (2)

- Una coda vive all'interno di RabbitMQ.
- Una coda è limitata solo dai limiti della memoria e del disco dell'host, è essenzialmente un buffer di messaggi di grandi dimensioni.
- Molti produttori possono inviare messaggi che vanno a una coda e molti consumatori possono provare a ricevere dati da una coda.
- Ecco come rappresentiamo una coda:

queue_name



Esercizio

Hello World (3)

- *Consumare* ha un significato simile a ricevere.
- Un *consumatore* è un programma che attende principalmente di ricevere messaggi:



- Si noti che il produttore, il consumatore e il broker non devono necessariamente risiedere sullo stesso host; infatti nella maggior parte delle applicazioni non accade.
- Un'applicazione può essere sia un produttore che un consumatore.

Esercizio

Hello World (4)

- In questa parte del tutorial scriveremo due programmi in Java (saranno presentati anche gli speculari per C#); un produttore che invia un singolo messaggio e un consumatore che riceve i messaggi e li stampa.
- Esamineremo alcuni dei dettagli nell'API Java (e C#), concentrandoci su questa cosa molto semplice solo per iniziare. È un "Hello World" di messaggistica.
- Nel diagramma seguente, "P" è il nostro produttore e "C" è il nostro consumatore. La casella nel mezzo è una coda, un buffer di messaggi che RabbitMQ mantiene per conto del consumatore.



Librerie client di RabbitMQ per ambiente Java

- Scaricare la libreria client e le sue dipendenze (SLF4 API e SLF4J Simple).
 - <https://repo1.maven.org/maven2/com/rabbitmq/amqp-client/5.7.1/amqp-client-5.7.1.jar>
 - <https://repo1.maven.org/maven2/org/slf4j/slf4j-api/1.7.26/slf4j-api-1.7.26.jar>
 - <https://repo1.maven.org/maven2/org/slf4j/slf4j-simple/1.7.26/slf4j-simple-1.7.26.jar>
- Copiare i file nella directory di lavoro.
- Si noti che SLF4J Simple è sufficiente per le esercitazioni, ma è necessario utilizzare una libreria di log completa come log4j in produzione.

Send.java

```
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.Channel;

public class Send {
    private final static String QUEUE_NAME = "hello";
    public static void main(String[] argv) throws
Exception {
        ...
    }
}
```

Send.java

- Creare la connessione al server

```
ConnectionFactory factory = new ConnectionFactory();  
  
factory.setHost("localhost");  
  
try (Connection connection = factory.newConnection();  
     Channel channel = connection.createChannel()) {}
```


Connessione al server

- La «connection» estrae la connessione socket e si occupa per noi della negoziazione e dell'autenticazione della versione del protocollo e così via.
- In questa esercitazione ci colleghiamo a un broker sul computer locale, quindi localhost.
- Se volessimo collegarci a un broker su un altro computer, specificherebbero semplicemente il suo nome o indirizzo IP.
- Successivamente creiamo un canale, dove risiede la maggior parte dei metodi le operazioni di business
- E' possibile usare un'istruzione try-with-resources perché sia Connection che Channel implementano `java.io.Closeable`. In questo modo non è necessario chiuderli esplicitamente nel codice.

Invio messaggio

- Per inviare un messaggio, dobbiamo dichiarare una coda
- Possiamo quindi pubblicare un messaggio nella coda, tutto questo nell'istruzione try-with-resources:

```
channel.queueDeclare(QUEUE_NAME, false, false, false, null);  
String messaggio = "Hello World!";  
channel.basicPublish("", QUEUE_NAME, null, message.getBytes());  
System.out.println(" [x] Invio '" + messaggio + "' ");
```

- La dichiarazione di una coda è idempotente: verrà creata solo se non esiste già.
- Il contenuto del messaggio è un array di byte.

L'invio non funziona!

- Se è la prima volta che viene utilizzato RabbitMQ e non viene stampato il messaggio "Invio", forse il broker non è stato avviato con sufficiente spazio libero su disco (per impostazione predefinita ha bisogno di almeno 200 Mb) e si rifiuta quindi di accettare messaggi.
- Controllare il file di registro del broker per confermare e ridurre il limite, se necessario.

`disk_free_limit`

Disk free space limit of the partition on which RabbitMQ is storing data. When available disk space falls below this limit, flow control is triggered. The value can be set relative to the total amount of RAM or as an absolute value in bytes or, alternatively, in information units (e.g. '50MB' or '5GB'):

```
disk_free_limit.relative = 3.0
```

```
disk_free_limit.absolute = 2GB
```

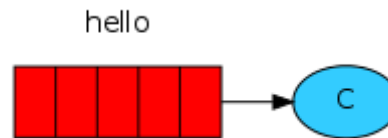
By default free disk space must exceed 50MB. See the [Disk Alarms](#) documentation.

Default:

```
disk_free_limit.absolute = 50MB
```

Receiver.java

- Receiver è il consumatore che ascolta i messaggi da RabbitMQ, quindi a differenza del pubblicatore, che pubblica un singolo messaggio, lo terremo in esecuzione per ascoltare i messaggi e stamparli.



- Il codice ha quasi le stesse importazioni di Send.java

```
import com.rabbitmq.client.Channel;  
import com.rabbitmq.client.Connection;  
import com.rabbitmq.client.ConnectionFactory;  
import com.rabbitmq.client.DeliverCallback;
```

Receiver.java

- L'impostazione è simile al pubblicatore; vengono aperte una connessione e un canale e poi si dichiara la coda dalla quale verranno consumati i messaggi.

```
public class Receiver {  
    private final static String QUEUE_NAME = "hello";  
    public static void main(String[] argv) throws Exception {  
        ConnectionFactory factory = new ConnectionFactory();  
        factory.setHost("localhost");  
        Connection connection = factory.newConnection();  
        Channel channel = connection.createChannel();  
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);  
        System.out.println(" [*] In attesa");  
    }  
}
```

Considerazioni

- Si noti che anche la coda deve essere dichiarata nel consumatore.... poiché potremmo avviare il consumatore prima del pubblicatore e desideriamo assicurarci che la coda esista prima di recuperare i messaggi.
- Perché non utilizziamo un'istruzione try-with-resource per chiudere automaticamente il canale e la connessione?
 - In questo modo faremmo semplicemente andare avanti il programma, chiudere tutto ed uscire!
 - Ciò sarebbe imbarazzante perché vogliamo che il processo rimanga vivo mentre il consumatore ascolta in modo asincrono l'arrivo dei messaggi 😊

Ricevere i messaggi DeliverCallback

Stiamo per dire al server di consegnarci i messaggi dalla coda. Dal momento che ci invierà i messaggi in modo asincrono, implementiamo una callback nella forma di un oggetto che bufferizzerà i messaggi fino a quando non saremo pronti per usarli. Questo è ciò che fa la sottoclasse DeliverCallback.

```
DeliverCallback deliverCallback = (consumerTag, delivery) -> {  
    String message = new String(delivery.getBody(), "UTF-8");  
    System.out.println(" [x] Received '" + message + "'");  
};  
channel.basicConsume(QUEUE_NAME, true, deliverCallback,  
consumerTag -> { });
```

Esempio in C#

- Per eseguire il medesimo esempio in ambiente C#
 - `dotnet new console --name Send`
 - `rename Send/Program.cs Send/Send.cs`
 - `dotnet new console --name Receive`
 - `rename Receive/Program.cs Receive/Receive.cs`
- Aggiungere le dipendenze
 - `cd Send`
 - `dotnet add package RabbitMQ.Client`
 - `dotnet restore`
 - `cd ../Receive`
 - `dotnet add package RabbitMQ.Client`
 - `dotnet restore`

Send.cs

```
using System;
using RabbitMQ.Client;
using System.Text;

class Send
{
    public static void Main()
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        using (var connection = factory.CreateConnection())
        using (var channel = connection.CreateModel())
        {
            channel.QueueDeclare(queue: "hello", durable: false, exclusive: false, autoDelete: false,
arguments: null);

            string message = "Hello World!";
            var body = Encoding.UTF8.GetBytes(message);

            channel.BasicPublish(exchange: "", routingKey: "hello", basicProperties: null, body:
body);

            Console.WriteLine(" [x] Sent {0}", message);
        }

        Console.WriteLine(" Press [enter] to exit.");
        Console.ReadLine();
    }
}
```

Receive.cs

```
using RabbitMQ.Client;
using RabbitMQ.Client.Events;
using System;
using System.Text;

class Receive
{
    public static void Main()
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        using (var connection = factory.CreateConnection())
        using (var channel = connection.CreateModel())
        {
            channel.QueueDeclare(queue: "hello", durable: false, exclusive: false, autoDelete: false,
arguments: null);

            var consumer = new EventingBasicConsumer(channel);
            consumer.Received += (model, ea) =>
            {
                var body = ea.Body;
                var message = Encoding.UTF8.GetString(body);
                Console.WriteLine(" [x] Received {0}", message);
            };
            channel.BasicConsume(queue: "hello", autoAck: true, consumer: consumer);

            Console.WriteLine(" Press [enter] to exit.");
            Console.ReadLine();
        }
    }
}
```

Mostrare le code

Come controllare lo stato delle code?

Linux: `sudo rabbitmqctl list_queues`

Windows: `rabbitmqctl.bat list_queues`

```
Timeout: 60.0 seconds ...
```


```
Listing queues for vhost / ...
```

```
name    messages
```

```
hello 0
```

```
task_queue 0
```

Da interfaccia web

 RabbitMQ™

3.8.0 Erlang 22.1.1

Overview

Connections

Channels

Exchanges

Queues

Admin

Queues

▼ All queues (2)

Pagination

Page

1 ▼

 of 1 - Filter: ☐ Regex ?

| Overview | | | | Messages | | | Message rates | | | +/- |
|------------|---------|--------------|------------------|----------|---------|-------|---------------|---------------|--------|-----|
| Name | Type | Features | State | Ready | Unacked | Total | incoming | deliver / get | ack | |
| hello | classic | | <div></div> idle | 0 | 0 | 0 | 0.00/s | 0.00/s | 0.00/s | |
| task_queue | classic | <div>D</div> | <div></div> idle | 0 | 0 | 0 | 0.00/s | 0.00/s | 0.00/s | |

► Add a new queue

Work Queues pattern

- In questo esercizio creeremo una *coda di lavoro* che verrà utilizzata per distribuire compiti che richiedono tempo tra i worker.
- L'idea principale dietro Work Queues (aka: *Task Queues*) è quella di evitare di svolgere immediatamente un'attività ad alta intensità di risorse e di dover attendere il completamento.
- L'idea è di programmare l'attività da svolgere in seguito.
- Incapsuliamo *un'attività* come messaggio e la inviamo a una coda.
- Un processo worker in esecuzione in background farà emergere le attività in coda ed eventualmente eseguirà il lavoro.
- Quando si eseguono molti worker, le attività verranno condivise tra loro.

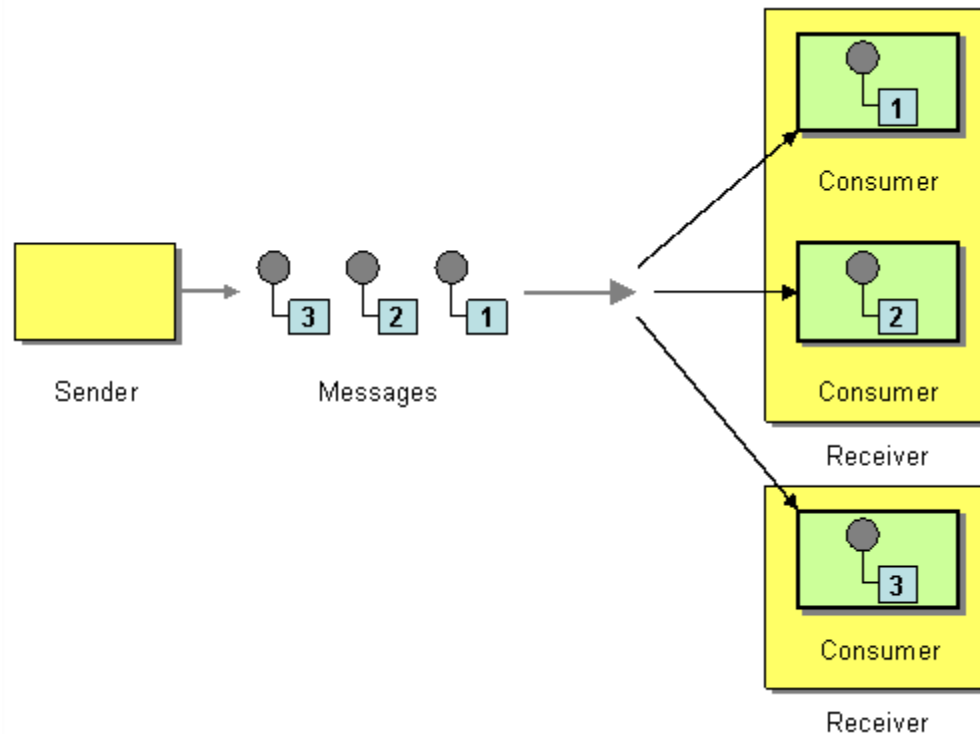
Messaging Pattern (EAP) (1)

<https://www.enterpriseintegrationpatterns.com/patterns/messaging/CompetingConsumers.html>

- **Create multiple *Competing Consumers* on a single channel so that the consumers can process multiple messages concurrently.**
- *Competing Consumers* are multiple consumers that are all created to receive messages from a single [Point-to-Point Channel](#). When the channel delivers a message, any of the consumers could potentially receive it. The messaging system's implementation determines which consumer actually receives the message, but in effect the consumers compete with each other to be the receiver. Once a consumer receives a message, it can delegate to the rest of its application to help process the message. (This solution only works with [Point-to-Point Channels](#); multiple consumers on a [Publish-Subscribe Channel](#) just create more copies of each message.)

Messaging Pattern (EAP) (2)

<https://www.enterpriseintegrationpatterns.com/patterns/messaging/CompetingConsumers.html>



- Questo concetto è particolarmente utile nelle applicazioni Web in cui è impossibile gestire un'attività complessa durante una breve finestra di richiesta HTTP.

Preparazione

- Nella parte precedente di questo esercizio abbiamo inviato un messaggio contenente "Hello World!". Ora invieremo stringhe che rappresentano attività complesse. Non abbiamo un compito nel mondo reale, come le immagini da ridimensionare o i file pdf da renderizzare, quindi fingeremo di essere occupati, utilizzando la funzione `Thread.sleep ()`. Prenderemo il numero di punti nella stringa come sua complessità; ogni punto rappresenterà un secondo di "lavoro". Ad esempio, un'attività descritta da 'Hello ...' richiederà tre secondi.
- Modificheremo leggermente il codice *Send.java* dal nostro esempio precedente, per consentire l'invio di messaggi arbitrari dalla riga di comando. Questo programma pianificherà le attività sulla nostra coda di lavoro, quindi chiamiamolo *NewTask.java*


NewTask.java

```
String message = String.join ( "" , argv);  
channel.basicPublish ( "" , "hello..." , null , message.getBytes ());  
System.out.println ( "[x] Sent '" + message + "'" );
```

- Anche il nostro vecchio programma *Receiver.java* richiede alcune modifiche: deve *simulare* un secondo di lavoro per ogni punto nel corpo del messaggio.
- Gestirà i messaggi consegnati ed eseguirà l'attività, quindi chiamiamolo *Worker.java*

Worker.java

```
DeliverCallback deliverCallback = (consumerTag, delivery) -> {  
    String message = new String(delivery.getBody(), "UTF-8");  
  
    System.out.println(" [x] Received '" + message + "'");  
    try {  
        doWork(message) ; •      ○      ○  
    } finally {  
        System.out.println(" [x] Done");  
    }  
};  
  
boolean autoAck = true; // vedremo dopo cosa significa  
channel.basicConsume(TASK_QUEUE_NAME, autoAck,  
deliverCallback, consumerTag -> { });
```



Il lavoro sporco
avviene qui!

Fake task per la simulazione

```
private static void doWork(String task) throws  
InterruptedException {  
    for (char ch: task.toCharArray()) {  
        if (ch == '.') Thread.sleep(1000);  
    }  
}
```

Politica di routing

- Per impostazione predefinita, RabbitMQ invierà ogni messaggio al consumatore successivo, in sequenza. In media ogni consumatore riceverà lo stesso numero di messaggi.
- Questo modo di distribuire i messaggi è chiamato **round-robin**.

Esercizio: Provare questa modalità di lavoro con tre o più worker!

Esempio in C#

- Per eseguire il medesimo esempio in ambiente C#
 - `dotnet new console --name NewTask`
 - `rename NewTask/Program.cs NewTask/NewTask.cs`
 - `dotnet new console --name Worker`
 - `rename Worker/Program.cs Worker/Worker.cs`
- Aggiungere le dipendenze
 - `cd NewTask`
 - `dotnet add package RabbitMQ.Client`
 - `dotnet restore`
 - `cd ../Worker`
 - `dotnet add package RabbitMQ.Client`
 - `dotnet restore`

NewTask.cs

```
using System;
using RabbitMQ.Client;
using System.Text;

class NewTask
{
    public static void Main(string[] args)
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        using(var connection = factory.CreateConnection())
        using(var channel = connection.CreateModel())
        {
            channel.QueueDeclare(queue: "task_queue", durable: true, exclusive: false, autoDelete: false, arguments: null);

            var message = GetMessage(args);
            var body = Encoding.UTF8.GetBytes(message);

            var properties = channel.CreateBasicProperties();
            properties.Persistent = true;

            channel.BasicPublish(exchange: "", routingKey: "task_queue", basicProperties: properties, body: body);
            Console.WriteLine(" [x] Sent {0}", message);
        }

        Console.WriteLine(" Press [enter] to exit.");
        Console.ReadLine();
    }

    private static string GetMessage(string[] args)
    {
        return ((args.Length > 0) ? string.Join(" ", args) : "Hello World!");
    }
}
```

Worker.cs

```
using System;
using RabbitMQ.Client;
using RabbitMQ.Client.Events;
using System.Text;
using System.Threading;

class Worker
{
    public static void Main()
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        using(var connection = factory.CreateConnection())
        using(var channel = connection.CreateModel())
        {
            channel.QueueDeclare(queue: "task_queue", durable: true, exclusive: false, autoDelete: false, arguments: null);

            channel.BasicQos(prefetchSize: 0, prefetchCount: 1, global: false);

            Console.WriteLine(" [*] Waiting for messages.");

            var consumer = new EventingBasicConsumer(channel);
            consumer.Received += (model, ea) =>
            {
                var body = ea.Body;
                var message = Encoding.UTF8.GetString(body);
                Console.WriteLine(" [x] Received {0}", message);

                int dots = message.Split('.').Length - 1;
                Thread.Sleep(dots * 1000);

                Console.WriteLine(" [x] Done");

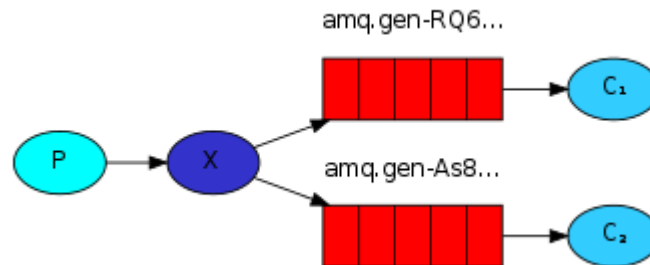
                channel.BasicAck(deliveryTag: ea.DeliveryTag, multiple: false);
            };
            channel.BasicConsume(queue: "task_queue", autoAck: false, consumer: consumer);

            Console.WriteLine(" Press [enter] to exit.");
            Console.ReadLine();
        }
    }
}
```

Esercizio

Publish/Subscribe

- Un sistema di log, in cui ogni copia in esecuzione del programma ricevente riceverà i messaggi.
- Ad esempio con questa modalità saremo in grado di eseguire un ricevitore e indirizzare i log su disco; e allo stesso tempo saremo in grado di eseguire un altro ricevitore e vedere i registri sullo schermo.
- In sostanza, i messaggi di registro pubblicati verranno trasmessi a tutti i ricevitori.
- Per farlo, faremo utilizzo di un exchange di tipo fanout



EmitLog Java

```
public class EmitLog {  
  
    private static final String EXCHANGE_NAME = "logs";  
  
    public static void main(String[] argv) throws Exception {  
        ConnectionFactory factory = new ConnectionFactory();  
        factory.setHost("localhost");  
        try (Connection connection = factory.newConnection();  
            Channel channel = connection.createChannel()) {  
            channel.exchangeDeclare(EXCHANGE_NAME, "fanout");  
  
            String message = argv.length < 1 ? "info: Hello World!" :  
                String.join(" ", argv);  
  
            channel.basicPublish(EXCHANGE_NAME, "", null, message.getBytes("UTF-8"));  
            System.out.println(" [x] Sent '" + message + "'");  
        }  
    }  
}
```

ReceiveLogs Java

```
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.DeliverCallback;

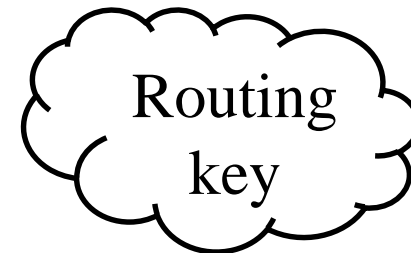
public class ReceiveLogs {
    private static final String EXCHANGE_NAME = "logs";

    public static void main(String[] argv) throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();

        channel.exchangeDeclare(EXCHANGE_NAME, "fanout");
        String queueName = channel.queueDeclare().getQueue();
        channel.queueBind(queueName, EXCHANGE_NAME, "");

        System.out.println(" [*] Waiting for messages. To exit press CTRL+C");

        DeliverCallback deliverCallback = (consumerTag, delivery) -> {
            String message = new String(delivery.getBody(), "UTF-8");
            System.out.println(" [x] Received '" + message + "'");
        };
        channel.basicConsume(queueName, true, deliverCallback, consumerTag -> { });
    }
}
```



Esecuzione Java

- Variabili di ambiente:
 - Linux
 - `export CP=.:amqp-client-5.7.1.jar:slf4j-api-1.7.26.jar:slf4j-simple-1.7.26.jar`
 - Windows
 - `set CP=.;amqp-client-5.7.1.jar;slf4j-api-1.7.26.jar;slf4j-simple-1.7.26.jar`

Esecuzione Java

- Compilare
 - `javac -cp $CP EmitLog.java ReceiveLogs.java`
- Eseguire
 - `java -cp $CP ReceiveLogs`
 - `java -cp $CP EmitLog`

```
sudo rabbitmqctl list_bindings
```

```
# => Listing bindings ...
```

```
# => logs    exchange      amq.gen-JzTY20B  queue           []
```

```
# => logs    exchange      amq.gen-vso0PV   queue           []
```

```
# => ...done.
```

EmitLog.cs

C#

```
using System;
using RabbitMQ.Client;
using System.Text;

class EmitLog
{
    public static void Main(string[] args)
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        using(var connection = factory.CreateConnection())
        using(var channel = connection.CreateModel())
        {
            channel.ExchangeDeclare(exchange: "logs", type: ExchangeType.Fanout);

            var message = GetMessage(args);
            var body = Encoding.UTF8.GetBytes(message);
            channel.BasicPublish(exchange: "logs",
                routingKey: "",
                basicProperties: null,
                body: body);

            Console.WriteLine(" [x] Sent {0}", message);
        }

        Console.WriteLine(" Press [enter] to exit.");
        Console.ReadLine();
    }

    private static string GetMessage(string[] args)
    {
        return ((args.Length > 0)
            ? string.Join(" ", args)
            : "info: Hello World!");
    }
}
```

ReceivedLogs.cs

```
using System;
using RabbitMQ.Client;
using RabbitMQ.Client.Events;
using System.Text;

class ReceiveLogs
{
    public static void Main()
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        using(var connection = factory.CreateConnection())
        using(var channel = connection.CreateModel())
        {
            channel.ExchangeDeclare(exchange: "logs", type: ExchangeType.Fanout);

            var queueName = channel.QueueDeclare().QueueName;
            channel.QueueBind(queue: queueName,
                exchange: "logs",
                routingKey: "");

            Console.WriteLine("[*] Waiting for logs.");

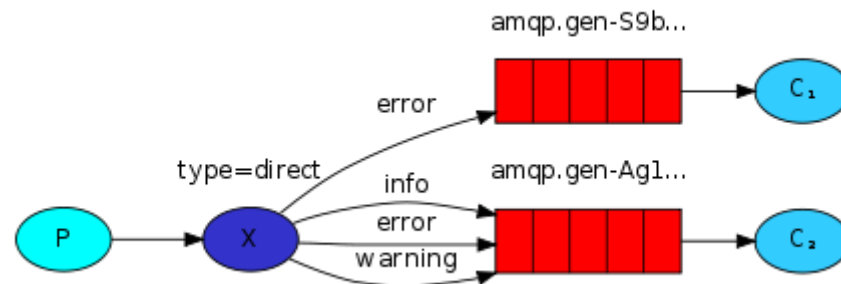
            var consumer = new EventingBasicConsumer(channel);
            consumer.Received += (model, ea) =>
            {
                var body = ea.Body;
                var message = Encoding.UTF8.GetString(body);
                Console.WriteLine(" [x] {0}", message);
            };
            channel.BasicConsume(queue: queueName,
                autoAck: true,
                consumer: consumer);

            Console.WriteLine(" Press [enter] to exit.");
            Console.ReadLine();
        }
    }
}
```

Esercizio

Gestire il log

- Registrare i messaggi di log di livelli
 - Info (I)
 - Warning (W)
 - Error (E)
- I messaggi di Errore (E) devono essere gestiti in modo prioritario da un consumatore specifico.
- Soluzione:
 - Una coda per un consumatore IWE
 - Una coda per un consumatore E



Produttore (1)

Java

- Dichiarare l'exchange
 - **channel.exchangeDeclare(EXCHANGE_NAME, "direct");**
- Invio messaggi
 - **channel.basicPublish(EXCHANGE_NAME, severity, null, message.getBytes());**
- Con severity intendiamo le tre categorie
 - Info
 - Warning
 - Error

Produttore (2)

Java

```
import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

public class EmitLogDirect {

    private static final String EXCHANGE_NAME = "direct_logs";

    public static void main(String[] argv) throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        try (Connection connection = factory.newConnection();
            Channel channel = connection.createChannel()) {
            channel.exchangeDeclare(EXCHANGE_NAME, "direct");

            String severity = getSeverity(argv);
            String message = getMessage(argv);

            channel.basicPublish(EXCHANGE_NAME, severity, null, message.getBytes("UTF-8"));
            System.out.println(" [x] Sent '" + severity + "':" + message + "'");
        }
    }
    //..
}
```

Consumatore (1)

Java

Associazione della coda ai relativi messaggi di severità

- `String queueName = channel.queueDeclare().getQueue();`
- `for(String severity : argv)`
 `channel.queueBind(queueName, EXCHANGE_NAME, severity);`

Consumatore (2)

Java

```
import com.rabbitmq.client.*;

public class ReceiveLogsDirect {

    private static final String EXCHANGE_NAME = "direct_logs";

    public static void main(String[] argv) throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        Connection connection = factory.newConnection();
        Channel channel = connection.createChannel();

        channel.exchangeDeclare(EXCHANGE_NAME, "direct");
        String queueName = channel.queueDeclare().getQueue();

        if (argv.length < 1) {
            System.err.println("Usage: ReceiveLogsDirect [info] [warning] [error]");
            System.exit(1);
        }

        for (String severity : argv) {
            channel.queueBind(queueName, EXCHANGE_NAME, severity);
        }

        System.out.println("[*] Waiting for messages. To exit press CTRL+C");

        DeliverCallback deliverCallback = (consumerTag, delivery) -> {
            String message = new String(delivery.getBody(), "UTF-8");
            System.out.println("[x] Received '" +
                delivery.getEnvelope().getRoutingKey() + "':" + message + "'");
        };
        channel.basicConsume(queueName, true, deliverCallback, consumerTag -> { });
    }
}
```

Esecuzione Java

- Compilare
 - `javac -cp $CP ReceiveLogsDirect.java EmitLogDirect.java`
- Eseguire
 - `java -cp $CP ReceiveLogsDirect info warning error`
 - `java -cp $CP ReceiveLogsDirect error`
 - `java -cp $CP EmitLogDirect error "Attenzione ERRORE grave... Guru Meditation!"`
 - `java -cp $CP EmitLogDirect info "Messaggio di INFO"`
 - `java -cp $CP EmitLogDirect warning "Messaggio di WARNING"`

Produttore C#

- Dichiarare l'exchange

```
channel.ExchangeDeclare(exchange: "direct_logs", type: "direct");
```

- Invio di messaggi

```
var body = Encoding.UTF8.GetBytes(message);  
channel.BasicPublish(exchange: "direct_logs",  
                    routingKey: severity,  
                    basicProperties: null,  
                    body: body);
```

EmitLogDirect.cs

```
using System;
using System.Linq;
using RabbitMQ.Client;
using System.Text;

class EmitLogDirect
{
    public static void Main(string[] args)
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        using(var connection = factory.CreateConnection())
        using(var channel = connection.CreateModel())
        {
            channel.ExchangeDeclare(exchange: "direct_logs", type: "direct");

            var severity = (args.Length > 0) ? args[0] : "info";
            var message = (args.Length > 1)
                ? string.Join(" ", args.Skip( 1 ).ToArray())
                : "Hello World!";
            var body = Encoding.UTF8.GetBytes(message);
            channel.BasicPublish(exchange: "direct_logs",
                routingKey: severity,
                basicProperties: null,
                body: body);

            Console.WriteLine("[x] Sent '{0}':'{1}'", severity, message);
        }

        Console.WriteLine(" Press [enter] to exit.");
        Console.ReadLine();
    }
}
```

Consumatore

C#

Associazione della coda ai relativi messaggi di severità

```
var queueName = channel.QueueDeclare().QueueName;  
  
foreach(var severity in args)  
{  
    channel.QueueBind(queue: queueName,  
        exchange: "direct_logs",  
        routingKey: severity);  
}
```

ReceiveLogsDirect.cs (1)

```
using System;
using RabbitMQ.Client;
using RabbitMQ.Client.Events;
using System.Text;

class ReceiveLogsDirect
{
    public static void Main(string[] args)
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        using(var connection = factory.CreateConnection())
        using(var channel = connection.CreateModel())
        {
            channel.ExchangeDeclare(exchange: "direct_logs",
                                   type: "direct");
            var queueName = channel.QueueDeclare().QueueName;

            if(args.Length < 1)
            {
                Console.Error.WriteLine("Usage: {0} [info] [warning] [error]",
                                         Environment.GetCommandLineArgs()[0]);
                Console.WriteLine(" Press [enter] to exit.");
                Console.ReadLine();
                Environment.ExitCode = 1;
                return;
            }
        }
    }
}
```


ReceiveLogsDirect.cs (2)

```
foreach(var severity in args)
{
    channel.QueueBind(queue: queueName,
        exchange: "direct_logs",
        routingKey: severity);
}

Console.WriteLine("[*] Waiting for messages.");

var consumer = new EventingBasicConsumer(channel);
consumer.Received += (model, ea) =>
{
    var body = ea.Body;
    var message = Encoding.UTF8.GetString(body);
    var routingKey = ea.RoutingKey;
    Console.WriteLine("[x] Received '{0}':'{1}'",
        routingKey, message);
};
channel.BasicConsume(queue: queueName,
    autoAck: true,
    consumer: consumer);

Console.WriteLine(" Press [enter] to exit.");
Console.ReadLine();
}
}
}
```

Esecuzione C#

- Eseguiare
 - cd ReceiveLogsDirect
 - dotnet run info warning error
 - cd EmitLogDirect
 - dotnet run error “Attenzione ERRORE grave... Guru Meditation!”
 - dotnet run info “Messaggio di INFO”
 - dotnet run warning “Messaggi di WARNING”