Lezione 22: Java Messaging Services - 2

Corso di *Programmazione Distribuita*

Vittorio Scarano

Università di Salerno

Laurea in Informatica

Il programma della lezione

PLAN

COME ASSICURARE UN RECAPITO AFFIDABILE

- Meccanismo di filtering: ricevere solo i messaggi che si vuole
- Scegliere il *time-to-live* per evitare messaggi obsoleti
- Specificare la persistenza di messaggi (nonostante possibili malfunzionamento del provider)
- Controllo degli ack a vari livelli
- Durable subscription
- Priorità di messaggi

FILTERING DEI MESSAGGI

- Si fa in modo che i messaggi che arrivino siano solamente quelli a cui si è interessati
- Nessuno spreco di banda per ricevere cose non interessanti
- Si può fare selezione su headers (metadata fissati dallo standard) e sulle proprietà
- Il consumer può scegliere usando sintassi SQL come

```
context.createConsumer(queue, "JMSPriority < 6").receive();
context.createConsumer(queue, "JMSPriority < 6 AND orderAmount < 200").receive();
context.createConsumer(queue, "orderAmount BETWEEN 1000 AND 2000").receive();
```

- Il messaggio viene creato dal Producer usando metodi per settare proprietà e priorità (nell'header)

```
context.createTextMessage().setIntProperty("orderAmount", 1530);
context.createTextMessage().setJMSPriority(5);
```

EVITARE MESSAGGI OBSOLETI

- Un setting del *time-to-live* può essere di aiuto per evitare che messaggi obsoleti vengano recapitati ai destinatari
- Si setta il tempo in millisecondi, passato il quale il provider (il broker) rimuove il messaggio
- Si utilizza il metodo del producer:

```
context.createProducer().setTimeToLive(1000).send(queue,
    message);
```

GESTIRE LA PERSISTENZA

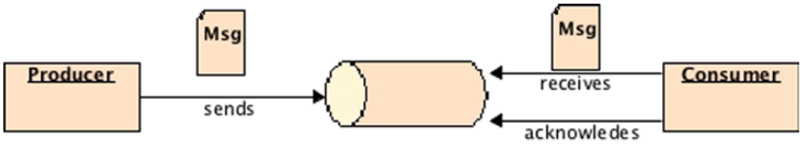
- Diverse qualità del servizio possono essere richieste al broker
  - con conseguente aggravio di carico, per i requisiti più stringenti
- Persistent delivery: recapito *exactly-once* del messaggio viene garantito
- Non-persistent delivery: recapito *at most once*
- Persistent delivery è il valore di default
  - che può essere "degradato" per migliorare le prestazioni

```
context.createProducer()
                    .setDeliveryMode(DeliveryMode.NON_PERSISTENT)
                    .send(queue,message);
```

CONTROLLO DEGLI ACKNOWLEDGMENT

- Si vuole ricevere una verifica del recapito del messaggio al destinatario
- Diverse modalità di acknowledgment
  - `AUTO_ACKNOWLEDGE`: automatico dalla sessione
  - `CLIENT_ACKNOWLEDGE`: automatico dalla sessione
  - `AUTO_ACKNOWLEDGE`: esplicito da parte del client `Message.acknowledge()`
  - `DUPS_OK _ACKNOWLEDGE`: possibili messaggi di ack duplicati
- Come si usa

```
@Inject
@JMSConnectionFactory("jms/connectionFactory")
@JMSSessionMode(JMSContext.AUTO_ACKNOWLEDGE)
private JMSContext context;
```



DURABLE CONSUMERS

- Nel modello publish-subscribe un consumer che non è in esecuzione perde i messaggi che vengono postati sul topic
- Con i durable consumer, si può controllare che i messaggi vengano mantenuti dal provider fino a quando tutti i consumer li hanno ricevuti
- Con i durable suscribers, un consumer che si riconnette riceve i messaggi che sono arrivati durante la disconnessione
- Creazione attraverso `JMSContext` con una id specifica “unica”

```
context.createDurableConsumer(topic, "uniqueID").receive();
```

SCEGLIERE PRIORITÀ DEL MESSAGGIO

- Contenute nell'header del messaggio
- Valori da 0 (bassa priorità) a 9 (alta priorità)
- Esempio:

```
context.createProducer().setPriority(2).send(queue, message);
```

- Concatenazione di diversi meccanismi:

```
context.createProducer().setPriority(2)
                .setTimeToLive(1000)
                .setDeliveryMode(DeliveryMode.NON_PERSISTENT)
                .send(queue, message);
```

PLAN

EJB CHE SCAMBIANO MESSAGGI

- Message Driven Bean (MDB) è un consumatore di messaggi, asincrono, invocato dal container quando arriva un messaggio
- Parte delle specifiche di Enterprise JavaBeans: simili a stateless
- Tramite CDI può accedere a altri EJB, JDBC, risorse JMS, entity manager, etc.
- Il vantaggio di usare un MDB è che transazione, multithread, sicurezza etc. sono gestiti dal container

UN ESEMPIO DI UN SEMPLICE MDB


```
@MessageDriven(mappedName =
    "jms/javaee7/Topic")
public class BillingMDB
        implements MessageListener {

 public void onMessage(Message
     message) {
      System.out.println("Received: "
      + message.getBody(String.class));
 }
}
```

- Definisce un MDB
- Implementa un `MessageListener` invece di interfacce remote: non serve specificare che metodi vengono offerti
- Cosa fare quando si riceve un messaggio

COME È FATTO UN MDB

- MDB non parte del modello EJB Lite: serve una implementazione full EE
- Annotazione con `@javax.ejb.MessageDriven` (o XML equivalente)
- Implementare la interfaccia del listener
- Definita come public, non final o abstract
- Deve esserci un costruttore senza argomenti, per permetterne l'istanziazione automatica da parte del container
- La classe non deve avere il metodo `finalize()`

COME CONFIGURARE UN MDB

- Possibile usare tutti i setting realizzabili con JMS

Property	Description
`acknowledgeMode`	The acknowledgment mode (default is `AUTO_ACKNOWLEDGE`)
`messageSelector`	The message selector string used by the MDB
`destinationType`	The destination type, which can be `TOPIC` or `QUEUE`
`destinationLookup`	The lookup name of an administratively-defined `Queue` or `Topic`
`connectionFactoryLookup`	The lookup name of an administratively defined `ConnectionFactory`
`destination`	The name of the destination.
`subscriptionDurability`	The subscription durability (default is `NON_DURABLE`)
`subscriptionName`	The subscription name of the consumer
`shareSubscriptions`	Used if the message-driven bean is deployed into a clustered
`clientId`	Client identifier that will be used when connecting to the JMS provider

COME USARE LE PROPRIETÀ DI UN MDB

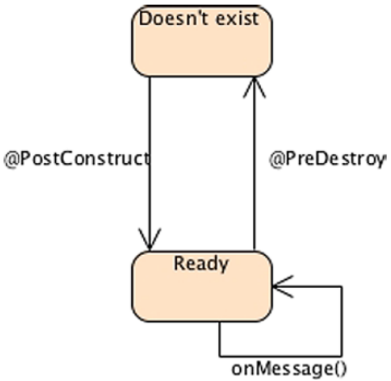

```
@MessageDriven(mappedName = "jms/javaee7/Topic",
 activationConfig={
     @ActivationConfigProperty(
         propertyName="acknowledgeMode",
         propertyValue = "Auto-acknowledge"),
     @ActivationConfigProperty(
         propertyName="messageSelector",
         propertyValue = "orderAmount < 3000")
})

public class BillingMDB implements MessageListener {
    public void onMessage(Message message) {
        System.out.println("Message received: " +
                           message.getBody(String.class));
    }
}
```

- Nella definizione del MDB
- Viene definita la configurazione . . .
- . . . con le sue proprietà
- . . . filtro di messaggi
- Metodo per gestire i messaggi

IL CONTESTO MessageDrivenContext

Method	Description
`getCallerPrincipal`	Returns the `java.security.Principal` associated with the invocation
`getRollbackOnly`	Tests whether the current transaction has been marked for rollback
`getTimerService`	Returns the `javax.ejb.TimerService` interface
`getUserTransaction`	Returns the `javax.transaction.UserTransaction` interface to use to demarcate transactions. Only MDBs with bean-managed transaction (BMT) can use this method
`isCallerInRole`	Tests whether the caller has a given security role
`Lookup`	Enables the MDB to look up its environment entries in the JNDI naming context
`setRollbackOnly`	Allows the instance to mark the current transaction as rollback. Only MDBs with BMT can use this method

IL CICLO DI VITA DI UN MDB



- Simile a quello degli stateless beans
- Possibile inserire interceptors dei metodi

POSSIBILE RICEVERE E INVIARE MESSAGGI


```
@MessageDriven(mappedName = "jms/javaee7/Topic",
 activationConfig = {
    @ActivationConfigProperty(
        propertyName="acknowledgeMode",
        propertyValue = "Auto-acknowledge"),
    @ActivationConfigProperty(
        propertyName="messageSelector",
        propertyValue = "orderAmount BETWEEN 3 AND 7")
})
public class BillingMDB implements MessageListener {
    @Inject
    @JMSConnectionFactory("jms/javaee7/ConnectionFactory")
    @JMSSessionMode(JMSContext.AUTO_ACKNOWLEDGE)
    private JMSContext context;

    @Resource(lookup = "jms/javaee7/Queue")
    private Destination printingQueue;

    public void onMessage(Message message) {
        System.out.println("Message received: "
                + message.getBody(String.class));
        sendPrintingMessage();
    }

    private void sendPrintingMessage() throws
         JMSException {
        context.createProducer().send(printingQueue,
             "Message has been received and resent");
    }
```

- MDB con il topic
- Configurazione
- Auto-ack
- Filtro messaggi
- Implementa listener
- Context iniettato dalla CF
- . . . nella variabile
- Destinazione iniettata dal container
- Metodo quando riceve messaggi
- Metodo per inviare

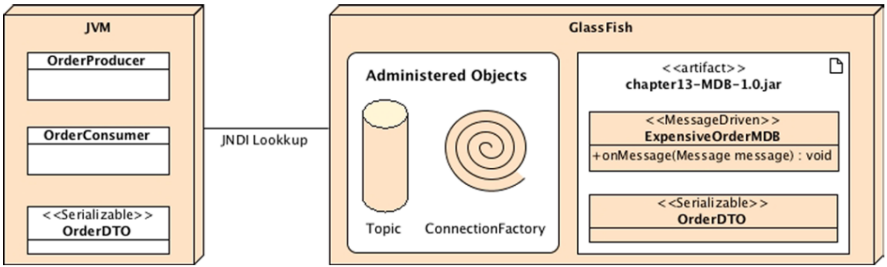
- In quanto EJB, le transazioni possono essere Bean-managed oppure Container-managed
- Transazioni per scambio di messaggi: un certo numero di messaggi vanno recapitati tutti insieme o nessuno
- Eccezioni specifiche: `JMSException` (checked) e `JMSRuntimeException` unchecked
- Eccezione `JMSException` tipicamente usata per fare rollback di transazioni (esplicitamente da parte del programmatore)

PLAN

PLAN

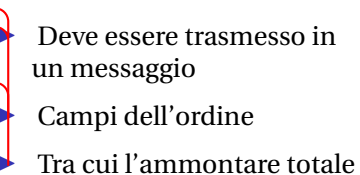
LO SCHEMA

- Producer stand-alone che invia messaggi a un topic, con un valore di `amount` (settato su linea di comando del producer)
- Il consumer `OrderConsumer` riceve tutti i messaggi
- ... mentre il MDB `ExpensiveOrderMDB` riceve solamente quelli sopra i 1000



L'ORDINE


```
public class OrderDTO implements Serializable {
    private Long orderId;
    private Date creationDate;
    private String customerName;
    private Float totalAmount;

    // Constructors, getters, setters
}
```



IL PRODUTTORE


```
public class OrderProducer {
 public static void main(String[] args)
                                throws NamingException {

  Float totalAmount = Float.valueOf(args[0]);

  OrderDTO order = new OrderDTO(12341, new Date(),
                        "Betty Moreau", totalAmount);

  Context jndiContext = new InitialContext();

  ConnectionFactory connectionFactory =
     (ConnectionFactory)
     jndiContext.lookup("jms/javaee7/ConnectionFactory");

  Destination topic = (Destination)
     jndiContext.lookup("jms/javaee7/Topic");

  try (JMSContext jmsContext =
       connectionFactory.createContext()) {
      jmsContext.createProducer()
           .setProperty("orderAmount",totalAmount).
           .send(topic, order);
  }
 }
}
```

- Parametro passato su linea di comando
- Si crea un nuovo ordine
- Si ottiene il contesto JNDI
- Lookup degli oggetti administered: CF
- . . . e topic
- Con il contesto creato. . .
- Si crea un produttore con una proprietà
- e si invia il messaggio

IL CONSUMATORE "STANDARD" JSE


```
public class OrderConsumer {
 public static void main(String[] args)
                                throws NamingException {

  Context jndiContext = new InitialContext();

  ConnectionFactory connectionFactory =
    (ConnectionFactory)
       jndiContext.lookup("jms/javaee7/ConnectionFactory");

  Destination topic = (Destination)
       jndiContext.lookup("jms/javaee7/Topic");

  try (JMSContext jmsContext=
                    connectionFactory.createContext()) {
   while (true) {
    OrderDTO order = jmsContext.createConsumer(topic)
                     .receiveBody(OrderDTO.class);
    System.out.println("Order received: " + order);
   }
  }
 }
}
```

- Il contesto dall'ambiente di esecuzione
- Lookup oggetti administered: CF e destinazione
- Con il contesto ottenuto
- Va in loop infinito…
- Crea un consumer
- Dal quale riceve messaggi
- Che stampa a video

IL MESSAGE-DRIVEN BEAN

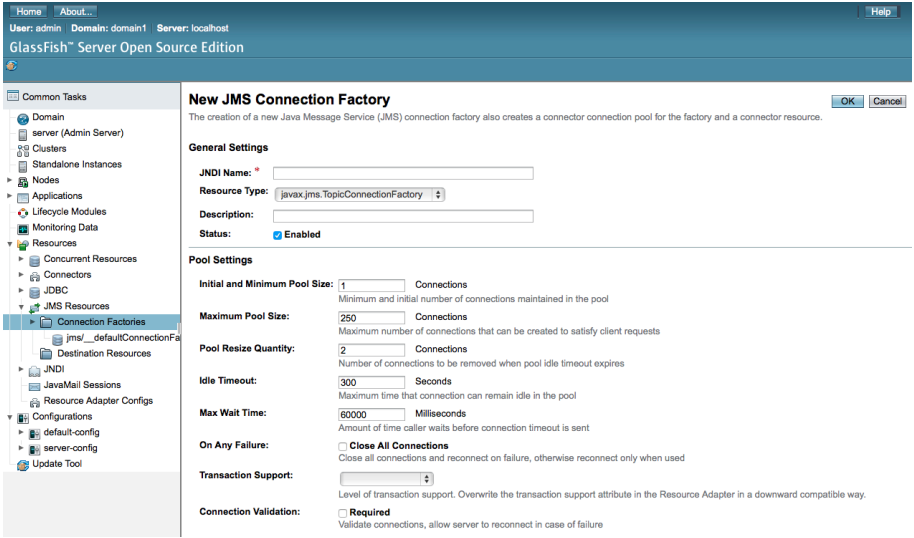

```
@MessageDriven(mappedName = "jms/javaee7/Topic",
 activationConfig = {
    @ActivationConfigProperty(
        propertyName="acknowledgeMode",
        propertyValue = "Auto-acknowledge"),
    @ActivationConfigProperty(
        propertyName="messageSelector",
        propertyValue = "orderAmount > 1000")
})
public class ExpensiveOrderMDB
                        implements MessageListener {

 public void onMessage(Message message) {
   try {
      OrderDTO order = message.getBody(OrderDTO.class);
      System.out.println("Expensive order received: "
                              + order.toString());
    } catch (JMSException e) {
            e.printStackTrace();
  }
 }
}
```

- MDB con il nome del topic da usare
- Configurazione: auto-ack
- …e selettore messaggi con ammontare alto
- Definizione dell'MDB
- Riceve il messaggio
- Lo stampa a video

PLAN

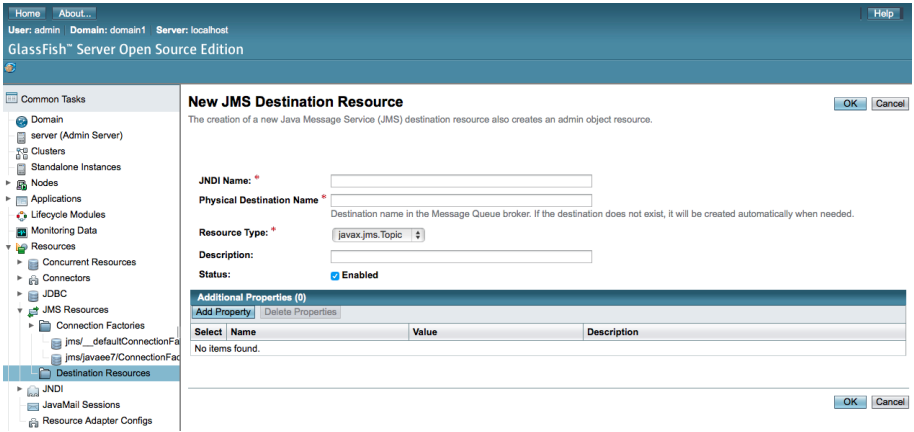
- Necessario che il server abbia:
  - Il Connection Factory
  - Il Topic
- Configurazione possibile da Console Web e da linea di comando
- Nella creazione del Topic necessario anche creare la destinazione fisica (creata di default con linea di comando)

CONFIGURAZIONE VIA WEB CONSOLE - 01

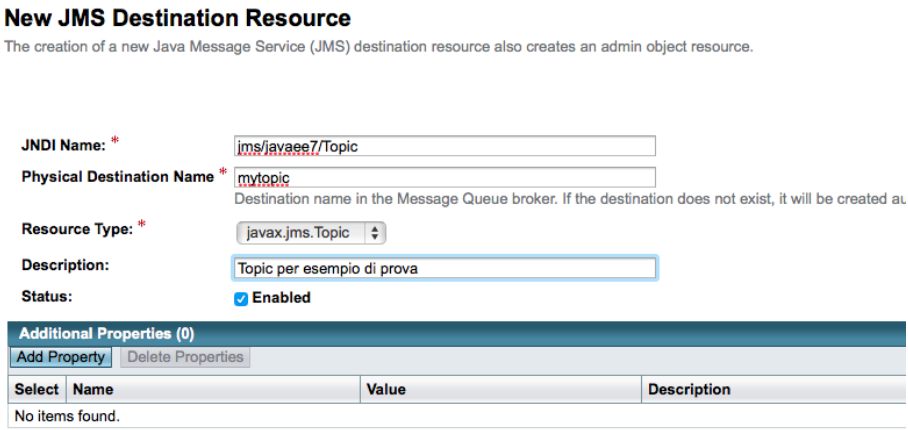


CONFIGURAZIONE VIA WEB CONSOLE - 02

CONFIGURAZIONE VIA WEB CONSOLE - 03



CONFIGURAZIONE VIA WEB CONSOLE - 04

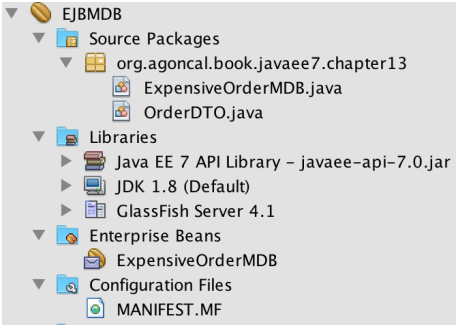


CONFIGURAZIONE VIA SHELL

- Uso della shell `asadmin`
- Comandi utili:
  - `asadmin create-jms-resource --restype javax.jms.ConnectionFactory jms/javaee7/ConnectionFactory`
  - `asadmin create-jms-resource --restype javax.jms.Topic jms/javaee7/Topic`
  - `asadmin list-jms-resources`

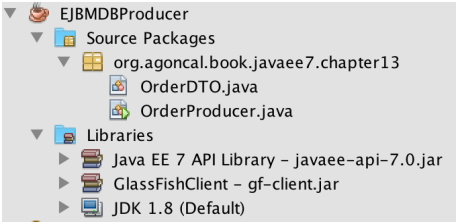
PLAN

IL MESSAGE-DRIVEN BEAN



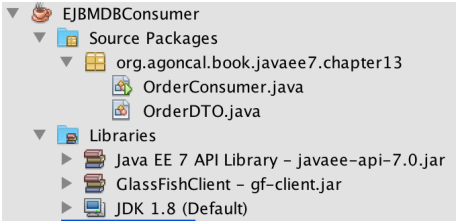
- Progetto per EJB
- Libreria di Java EE 7 aggiunta
- Build deploy

IL PRODUTTORE



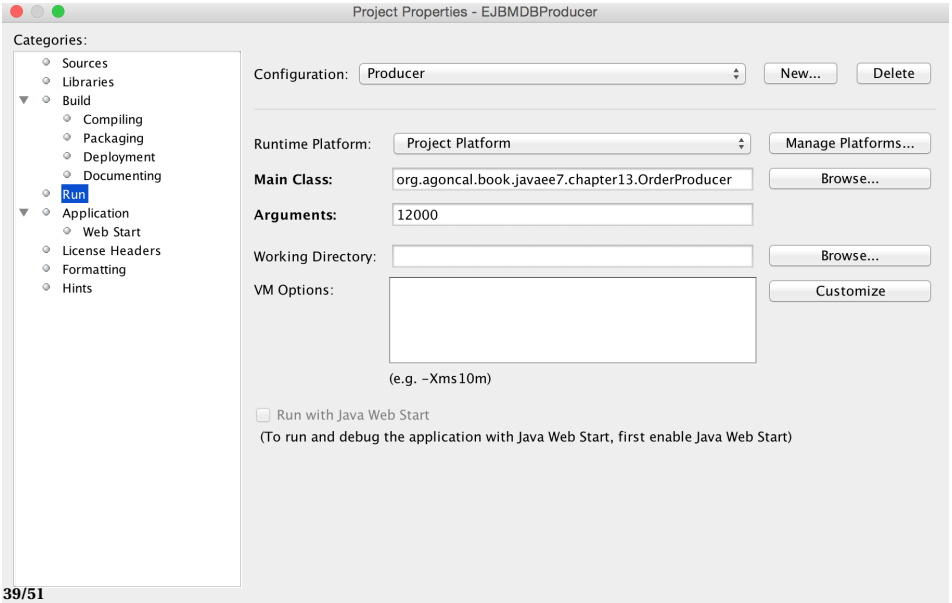
- Progetto standard JSE
- Libreria di Java EE 7 aggiunta
- Libreria di Glassfish client aggiunta
- Build e run
- Parametro su linea di comando da configurazione di lancio (right-click, Properties, Run)

IL CONSUMATORE

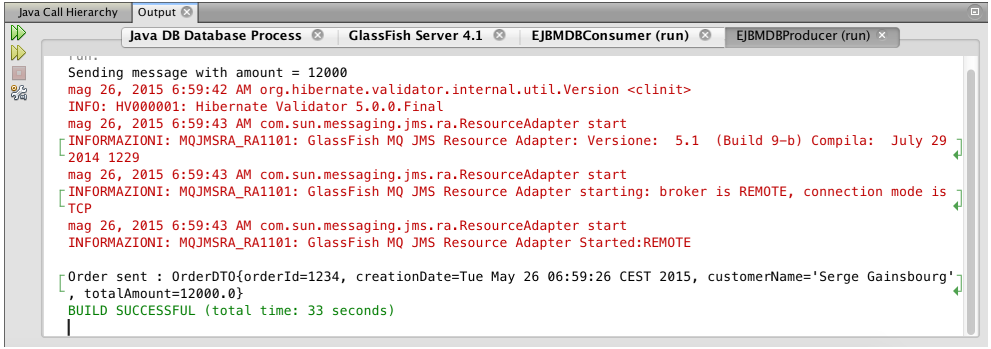


- Progetto standard JSE
- Libreria di Java EE 7 aggiunta
- Libreria di Glassfish client aggiunta
- Build e run

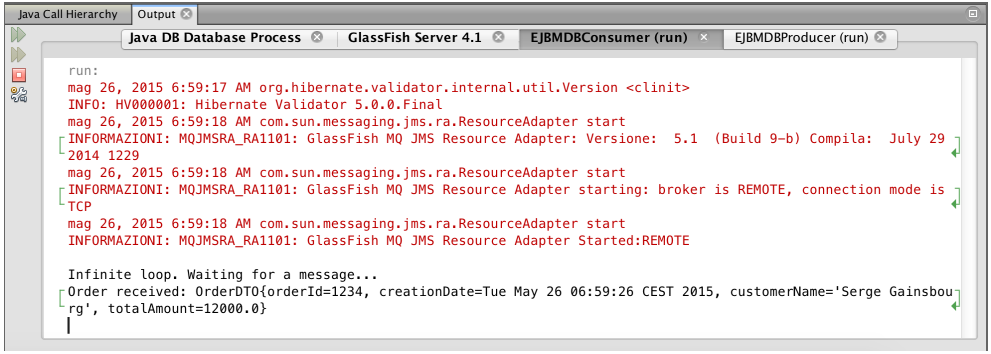
ESECUZIONE - 00: SET PARAMETRI



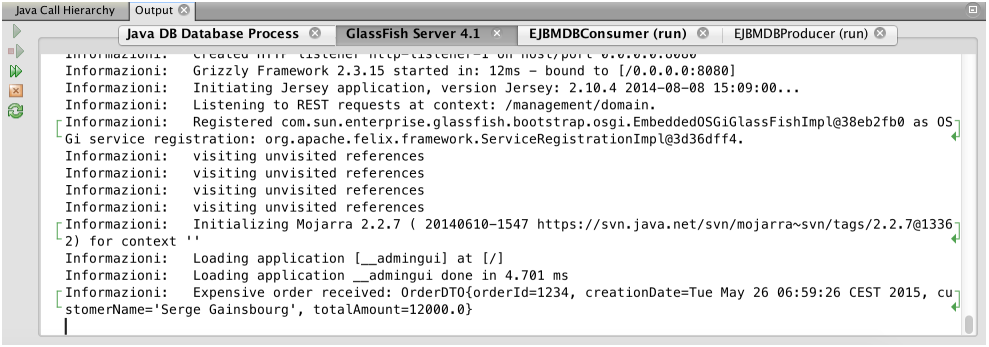
ESECUZIONE - 01: IL PRODUTTORE (12000)



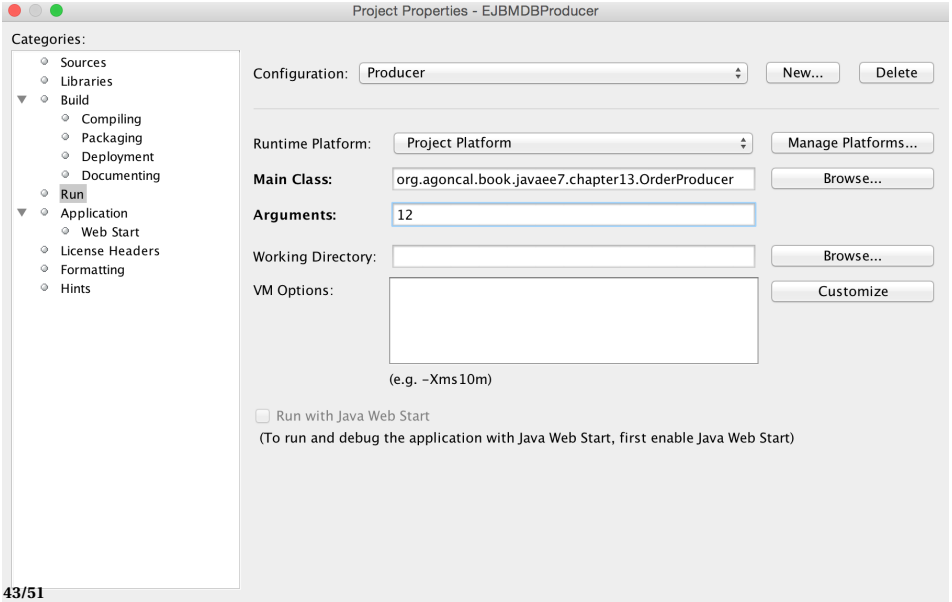
ESECUZIONE - 02: IL CONSUMATORE



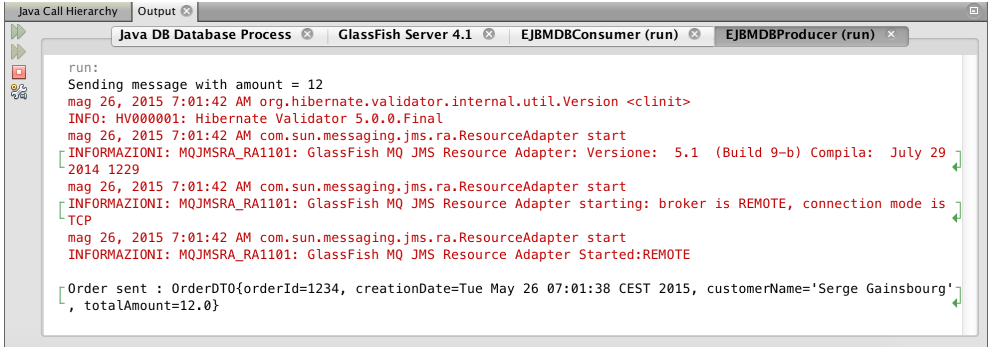
ESECUZIONE - 03: IL MDB SU CONTAINER



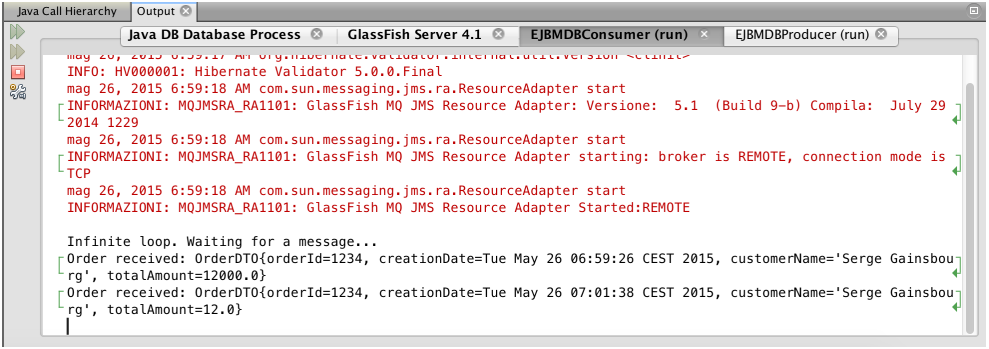
ESECUZIONE - 04: CAMBIO PARAMETRI A 12



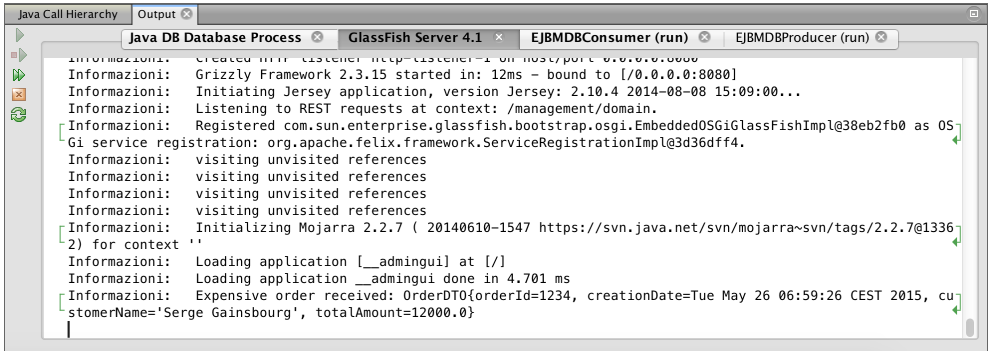
ESECUZIONE - 06: INVIO CON AMOUNT=12



ESECUZIONE - 07: IL CONSUMER INVECE SI



ESECUZIONE - 08: IL MDB NON RICEVE NIENTE



PLAN

ESEMPIO DI ESERCIZIO

- Scrivere un insieme di EJB e client che rappresentino un archivio di libri (con titolo, id, autore, prezzo)
- La organizzazione deve essere fatta in distinti progetti NetBeans
- Devono essere previste delle query per id (chiave primaria) e una query che restituisce tutti i libri
- Deve esserci un client basato su messaggi, che invia un messaggio con la id e il prezzo nuovo, che prevede l'aggiornamento del prezzo
- Deve esserci un client basato su invocazione di un bean stateless, che prevede la stampa della lista dei libri
- Deve essere previsto un bean singleton che inizializzi l'archivio

COMMENTI GENERALI

- Poco codice da scrivere, ma si richiede precisione
- Importante la struttura dei file e dei progetti (sviluppati su carta)
- Non viene richiesta la scrittura di XML (si assume verrà fatto nel progetto consegnato successivamente)
- Si richiede l'uso congiunto di EJB Stateless/stateful, di JPA, di JMS, di Web Services (da fare)

PLAN

QUELLO CHE ABBIAMO FATTO...