

## [EJB3 e Flex 2: Interfacciare Flex con gli Ejb3 - Parte 3](#)

(, 10-04-2007 10:10)

**Finalmente ci occupiamo di una delle parti piÃ¹ interessanti di questa rassegna: il collegamento tra Flex e gli Ejb. In questa lezione vediamo come sia possibile collegarci al Ejb creato nella lezione precedente attraverso il meccanismo dei FlexFactory dei Flex Data Services.**

I Flex Data Services come avevamo accennato nella prima lezione danno la possibilitÃ  alle applicazioni Flex di collegarsi a oggetti remoti scritti in Java (POJO) utilizzando un protocollo di comunicazione proprietario molto performante (RMTP). Le chiamate agli oggetti Java vengono effettuate attraverso la classe "flex.messaging.services.remoting.adapters.JavaAdapter", anch'essa estendibile, la quale prima recupera l'istanza dell'oggetto di business e poi invoca il metodo desiderato.

Il recupero dell'istanza dell'oggetto avviene attraverso un meccanismo di factory estendibile e configurabile permettendo l'integrazione con service layer sviluppati con diverse tecnologie, quali Ejb e Spring per esempio.

Il meccanismo del factory si puÃ² estendere grazie all'interfaccia "flex.messaging.FlexFactory", alla classe "flex.messaging.FactoryInstance" e modificando i file di configurazione dei Flex Data Services.

### **Implementare il meccanismo del FlexFactory**

Estendere queste due classi significa implementarne i metodi nel modo seguente:

La classe FactoryInstance Ã¨ stata definita come inner class statica per propria scelta poichÃ© legata strettamente al EjbFactory ma ai fini pratici Ã¨ possibile anche implementarla come classe normale.

Per capire il funzionamento di queste due classi Ã¨ sufficiente visualizzarne il diagramma delle sequenze nella fase di inizializzazione e in quella di esecuzione:

Durante la fase di inizializzazione che avviene all'avvio dell'applicazione Flex (nel nostro caso l'applicazione flex.war), la configurazione procede attraverso i file services-config.xml e remoting-config.xml situati nella WEB-INF/flex. Il primo file permette di istanziare e configurare una tipologia di FlexFactory identificata

attraverso un codice univoco:

```
<factories>
  <factory id="myFactoryId" class="myPackage.MyFlexFactory">
    <properties>

<myfactoryattributename>myfactoryattributevalue</myfactoryattributename>
    </properties>
  </factory>
</factories>
```

Nell'esempio viene istanziata la classe "myPackage.MyFlexFactory" che estende FlexFactory e che attraverso il metodo initialize() viene inizializzata con il codice "myFactoryId" e con una mappa di proprietà personalizzabili.

Successivamente viene processato il file remoting-config.xml che contiene la definizione di tutte le destinazioni dei servizi utilizzati dall'applicazione con il relativo FlexFactory da utilizzare per la risoluzione dell'istanza del servizio:

```
<destination id="myDestination">
  <properties>
    <factory>myFactoryId</factory>
    <source>mypackage.MyRemoteClass</source>
    <myfactoryinstanceattribute>myfoobar2value</myfactoryinstanceattribute>
  </properties>
</destination>
```

Nell'esempio viene definita la destinazione "myDestination" che avrà associato un'istanza FactoryInstance costruita dal FlexFactory di tipo "myFactoryId". Le proprietà a seguire permettono di configurare il FactoryInstance attraverso una mappa di valori passata attraverso il metodo "createFactoryInstance()" del FlexFactory e attraverso il costruttore del FactoryInstance.

Rimandiamo all'esame del codice in seguito che chiarirà ulteriormente questi punti.

Una volta inizializzate le destinazioni con la propria istanza di FactoryInstance, esse risponderanno alle invocazioni di metodo eseguite tramite il JavaAdapter nel modo seguente:

La sequenza chiarisce come il recupero del nostro Ejb avverrà implementando il metodo lookup() del FlexFactory in modo tale che interroghi il servizio di naming utilizzando per esempio la proprietà "source" già disponibile dal FactoryInstance base, per ottenere l'indirizzo JNDI corretto.

Il metodo "operationComplete()" non verrà utilizzato nei nostri esempi ma potrebbe essere utile in caso si vogliano effettuare delle deinizializzazioni sull'oggetto recuperato dopo averne invocato il metodo di business.

### **Sviluppo del nostro EjbFactory**

Per sviluppare la nostra implementazione del factory per poter accedere agli Ejb dovremo creare un nuovo progetto Java che chiameremo "Ejb3FlexFactory", seguendo gli stessi passi svolti durante la creazione del progetto della lezione precedente.

Una volta creato il progetto aggiungiamo la cartella "lib" per raccogliere le librerie necessarie alla compilazione e assicuriamoci di avere le librerie:

- log4j.jar
- flex-messaging.jar
- flex-messaging-common.jar

Le librerie di flex si possono copiare dalla "WEB-INF/lib" della nostra applicazione web flex.war (ricordatevi dopo averle copiate di aggiungerle al build path del progetto).

Creiamo ora il package "com.aminoweb.flexfactory" e la classe "com.aminoweb.flexfactory.EjbFactory" implementata come segue:

```
package com.aminoweb.flexfactory;

import javax.naming.InitialContext;

import javax.naming.NameNotFoundException;

import javax.naming.NamingException;

import org.apache.log4j.Logger;

import flex.messaging.FactoryInstance;

import flex.messaging.FlexFactory;

import flex.messaging.config.ConfigMap;

import flex.messaging.services.ServiceException;

/**
 * Questa classe fornisce una semplice implementazione dell'interfaccia
 * FlexFactory che consente di collegare un RemoteObject ad un Ejb di sessione
 * stateless. Estendendo il metodo lookup() del FlexFactory è possibile infatti
 * interrogare il servizio di naming tramite JNDI e recuperare l'Ejb registrato
 * col nome definito dalla proprietà "source" nella configurazione del
 * remoting-config.xml.
 *
 * @author Roberto Tassi (roberto.tassi@aminoweb.com)
 *

```

```
*/

public class EjbFactory implements FlexFactory {

/**
 *
 */

private static Logger LOGGER = Logger.getLogger(EjbFactory.class

.getName());

/**
 * Nome attributo di configurazione che specifica il nome con cui è stato
registrato l'Ejb nel servizio di naming.
 */

private static final String SOURCE = "source";

/**
 * Codice di errore per Ejb non trovato.
 */

private static final String EJB_NOT_FOUND_ERROR_CODE = "EJB.EjbNotFound";

/**
 * Codice di errore generico prodotto dal servizio di naming.
 */

private static final String NAMING_ERROR_CODE = "EJB.Naming";
```

```
/**
 *
 *
 */
public EjbFactory() {
    LOGGER.info("EjbFactory()");
}

/**
 * Inizializza il factory attraverso un codice univoco e una
 * mappa di proprietà di configurazione.
 *
 * @see flex.messaging.FlexConfigurable#initialize(java.lang.String,
 *         flex.messaging.config.ConfigMap)
 */
public void initialize(String pId, ConfigMap pProperties) {
    LOGGER.info("EjbFactory.initialize()");
}

/**
 * Crea un'istanza di collegamento ad uno specifico Ejb.
 *
 * @see flex.messaging.FlexFactory#createFactoryInstance(java.lang.String,
```

```
*      flex.messaging.config.ConfigMap)
*/

public FactoryInstance createFactoryInstance(String pId,
ConfigMap pProperties) {
    LOGGER.info("EjbFactory.createFactoryInstance()");
    FactoryInstance instance = new EjbFactoryInstance(this, pId,
pProperties);

    /* inizializza la proprietà source sull'istanza a partire dal valore letto dalla
configurazione */

    instance.setSource(pProperties.getPropertyAsString(SOURCE, instance
.getId()));

    return instance;
}

/**
 * Recupera l'Ejb relativo all'istanza specificata.
 *
 * @see flex.messaging.FlexFactory#lookup(flex.messaging.FactoryInstance)
 */

public Object lookup(FactoryInstance pFactoryInstance) {
    LOGGER.info("EjbFactory.lookup()");

    try {

        InitialContext context = new InitialContext();
```

```
return context.lookup(pFactoryInstance.getSource());

} catch (NameNotFoundException e) {

throw createServiceException(EJB_NOT_FOUND_ERROR_CODE,

"EJB not found: " + pFactoryInstance.getSource(), e);

} catch (NamingException e) {

throw createServiceException(NAMING_ERROR_CODE,

"Naming exception: " + pFactoryInstance.getSource(), e);

}

}

/**

* Crea un'eccezione del servizio con un codice e messaggio parlante.

*

* @param pCode

* @param pMessage

* @param pDetails

* @param pCause

* @return

*/

private ServiceException createServiceException(String pCode,

String pMessage, Throwable pCause) {

ServiceException e = new ServiceException();

e.setCode(pCode);
```

```
e.setMessage(pMessage);

e.setDetails(pMessage);

e.setRootCause(pCause);

return e;

}

/**
 * Estende il FactoryInstance base semplicemente per loggarne i metodi chiave.
 *
 * @author Roberto Tassi
 *
 */

static class EjbFactoryInstance extends FactoryInstance {

/**
 *
 */

private static Logger LOGGER = Logger

.getLogger(EjbFactoryInstance.class.getName());

/**
 * Crea un'istanza utilizzando una mappa di proprietà di configurazione.
 */
```

```
EjbFactoryInstance(FlexFactory pEjbFactory, String pId,
ConfigMap pProperties) {
    super(pEjbFactory, pId, pProperties);
    LOGGER.info("EjbFactoryInstance()");
}

/**
 * Recupera l'Ejb associato a questa istanza.
 *
 * @see flex.messaging.FactoryInstance#lookup()
 */
public Object lookup() {
    LOGGER.info("EjbFactoryInstance.lookup()");
    return super.lookup();
}

/**
 * Completa l'operazione di lookup.
 *
 * @see flex.messaging.FactoryInstance#operationComplete(java.lang.Object)
 */
public void operationComplete(Object obj) {
    LOGGER.info("EjbFactoryInstance.operationComplete()");
}
```

```
super.operationComplete(obj);  
  
}  
  
}  
  
}
```

Come si può vedere dal metodo `EjbFactory.lookup()` l'indirizzo da utilizzare nell'interrogazione al servizio di naming viene recuperata attraverso la proprietà "source" già definita nella classe `FactoryInstance`. L'implementazione del nostro `EjbFactoryInstance` non sarebbe necessaria poichè ne deleghiamo il comportamento alla classe madre `FactoryInstance` ma ci torna utile per attivare il log sulle chiamate dei metodi per capirne meglio il funzionamento.

Poichè la nostra classe deve essere istanziabile da Flex Data Services, essa deve essere archiviata in un jar e copiata nella cartella `WEB-INF/lib` della nostra applicazione Flex.

Come abbiamo visto dall'articolo precedente è molto facile utilizzare Ant per questo tipo di operazioni, per cui creiamo un file "build.xml" che sia in grado pacchettizzare la nostra libreria:

```
<project name="ejb3-flex-factory" basedir=".." default="dist">  
  
  <property name="eclipse.classes.path" location="bin" />  
  
  <property name="out.classes.path" location="out/classes" />  
  <property name="out.dist.path" location="out/dist" />  
  <property name="out.dist.jar"  
location="${out.dist.path}/${ant.project.name}.jar" />  
  
  <!-- =====  
    target: dist  
    ===== -->  
  <target name="dist" depends="clean,init,compile" description="Distribute the  
application">  
    <jar destfile="${out.dist.jar}" basedir="${out.classes.path}" />  
  </target>  
  
  <!-- - - - - -  
    target: clean  
    - - - - - -->  
  <target name="clean">
```

```
<delete dir="${out.classes.path}" />
<delete dir="${out.dist.path}" />
</target>
<!-- - - - - -
      target: init
      - - - - - -->
<target name="init">
  <mkdir dir="${out.classes.path}" />
  <mkdir dir="${out.dist.path}" />
</target>
<!-- - - - - -
      target: compile
      - - - - - -->
<target name="compile">

  <!-- PER ADESSO UTILIZZO IL COMPILATO DI ECLIPSE -->
  <copy todir="${out.classes.path}">
    <fileset dir="${eclipse.classes.path}" />
  </copy>
</target>

</project>
```

Una volta lanciato il task Ant, il jar con la nostra libreria sarà disponibile nella cartella "out/dist" per cui sarà sufficiente copiarla nella cartella "WEB-INF/lib" dell'applicazione flex.war affinché il nostro factory sia istanziabile.

Ricordatevi di riavviare JBoss o ricaricare l'applicazione flex.war affinché le modifiche al jar siano attive.

### Utilizzo del nostro EjbFactory da Flex

A questo punto per utilizzare il factory da noi implementato sarà necessario configurare Flex in modo opportuno e creare un'applicazione di esempio Flex per testarne il funzionamento.

Per attivare il nostro factory è necessario aggiungere questa configurazione al services-config.xml:

```
<factories>
```

```
<factory id="ejb3" class="com.aminoweb.flexfactory.EjbFactory" />
</factories>
```

Mentre per definire la destinazione per collegarsi al nostro Ejb è necessario aggiungere questa configurazione al file remoting-config.xml:

```
<destination id="LocalService">
  <properties>
    <factory>ejb3</factory>
    <source>ServiceBean/local</source>
  </properties>
</destination>
```

Editare il file FlexSample.mxml nel modo seguente:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute">

  <mx:Script>
    <![CDATA[
      import mx.messaging.events.MessageEvent;
      import mx.messaging.messages.AsyncMessage;
      import mx.controls.Alert;

      import mx.rpc.events.FaultEvent;
      import mx.rpc.events.ResultEvent;

      private function onResult(event:ResultEvent):void {
        var total:Number = event.result as Number;
        Alert.show("Total: " + total);
      }
    ]]>
  </mx:Script>
</mx:Application>
```

```
private function onFault(event:FaultEvent):void {
    Alert.show("ERROR: " + event.message);
}

private function callService():void {
    service.multiply(2, 5);
}
]]>
</mx:Script>

<mx:RemoteObject id="service" destination="LocalService"
result="onResult(event)" fault="onFault(event)" />

<mx:Button label="Call Service" click="callService()" />

</mx:Application>
```

L'applicazione non fa altro che definire un servizio remoto legato alla destinazione definita in precedenza e a visualizzare un bottone la cui azione di pressione è l'esecuzione del metodo remoto di moltiplicazione presente nel nostro servizio Ejb.

Eseguendo la nostra applicazione e premendo il bottone "Call Service" verrà visualizzata la schermata:

Sulla console di JBoss invece comparirà la sequenza di log attivata dalle nostre chiamate sui metodi delle classi implementate:

Si può notare ad una successiva pressione del bottone come l'istanza del ServiceBean sia stata messa in un pool dal container JBoss:

~~Questo esempio di FlexFactory è molto semplice ma già fa capire come sia flessibile il meccanismo dei FlexFactory per integrarsi ad un qualsiasi service layer. L'esempio utilizzato funziona però solamente con gli Ejb stateless mentre per quelli stateful sarebbe necessario implementare una cache di sessione, in modo che una volta effettuato il lookup del Ejb, per quella sessione venga riutilizzato lo stesso ad ogni invocazione. Rimandando a lezioni successive un'approfondimento del nostro FlexFactory per trattare quest'ultimo tema, anticipo che l'argomento della prossima lezione sarà rivolto al supporto ai Web Service e ai Message Driven Beans offerto dagli Ejb3 e ovviamente alla loro integrazione con Flex.~~

## Materiale

Progetto Ejb3FlexFactory completo: [download](#)

## Riferimenti

Ejb3 e Flex2: Creare un progetto Ejb3 - Parte 2:

<http://www.augitaly.com/flexgala/index.php?cmd=newsreader&id=77>

Ejb3 e Flex2: Installare il server JBoss - Parte 1:

<http://www.augitaly.com/flexgala/index.php?cmd=newsreader&id=76>

EJB and Flex Integration: [http://weblogs.macromedia.com/pmartin/archives/2006/08/ejb\\_and\\_flex\\_in.cfm](http://weblogs.macromedia.com/pmartin/archives/2006/08/ejb_and_flex_in.cfm)

Using the factory mechanism:

[http://livedocs.adobe.com/flex/201/html/ent\\_services\\_config\\_097\\_26.html#282856](http://livedocs.adobe.com/flex/201/html/ent_services_config_097_26.html#282856)

FlexFactory API Javadoc: <http://livedocs.adobe.com/flex/2/fds2javadoc/flex/messaging/FlexFactory.html>

## Note sull'autore

*Roberto Tassi (roberto.tassi@aminoweb.com) è un consulente Sun Certificated Java Programmer & Developer, che da 8 anni si occupa di sviluppo di applicazioni web-oriented su piattaforma Java. La sua esperienza web è partita da Cold Fusion, per poi consolidarsi su Java fino ad approdare al fantastico mondo di Flex.*