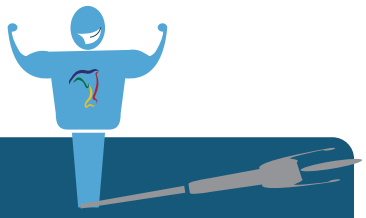
autentia

Avenida de Castilla,1 - Edificio Best Point - Oficina 21B
28830 San Fernando de Henares (Madrid)
tel./fax: +34 91 675 33 06
info@autentia.com - **www.autentia.com**

¿**Qué ofrece** Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

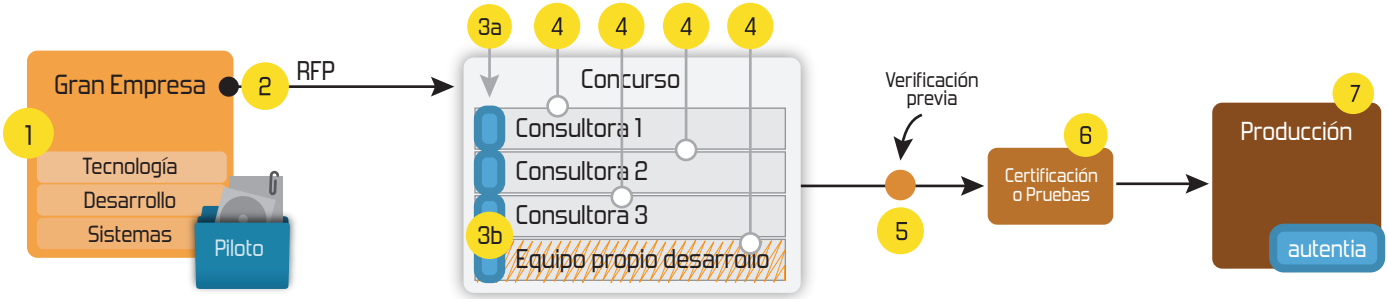
1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

Spring MVC, JSF-PrimeFaces /RichFaces, HTML5, CSS3, JavaScript-jQuery

Gestor portales (Liferay)
Gestor de contenidos (Alfresco)
Aplicaciones híbridas

Control de autenticación y acceso (Spring Security)
UDDI
Web Services
Rest Services
Social SSO
SSO (Cas)

JPA-Hibernate, MyBatis
Motor de búsqueda empresarial (Solr)
ETL (Talend)

Dirección de Proyectos Informáticos.
Metodologías ágiles
Patrones de diseño
TDD

Tareas programadas (Quartz)
Gestor documental (Alfresco)
Inversión de control (Spring)

BPM (jBPM o Bonita)
Generación de informes (JasperReport)
ESB (Open ESB)

Compartimos nuestro conocimiento en:
www.adictosaltrabajo.com

Para más información visítenos en:
www.autentia.com

Tutorial desarrollado por:

Alejandro Pérez García

Alejandro es socio fundador de Autentia y nuestro experto en J2EE, Linux y optimización de aplicaciones empresariales.

Si te gusta lo que ves, puedes contratarle para impartir **cursos presenciales** en tu empresa o para **ayudarte en proyectos** (Madrid).

Puedes encontrarme en Autentia

Somos expertos en Java/J2EE

Descargar este documento en formato PDF rmiRemoteRegistry.pdf

Fecha de creación del tutorial: 2008-01-11

RMI y como registrar objetos en un Registry remoto

Creación: 08-01-2008

Índice de contenidos

1. Introducción

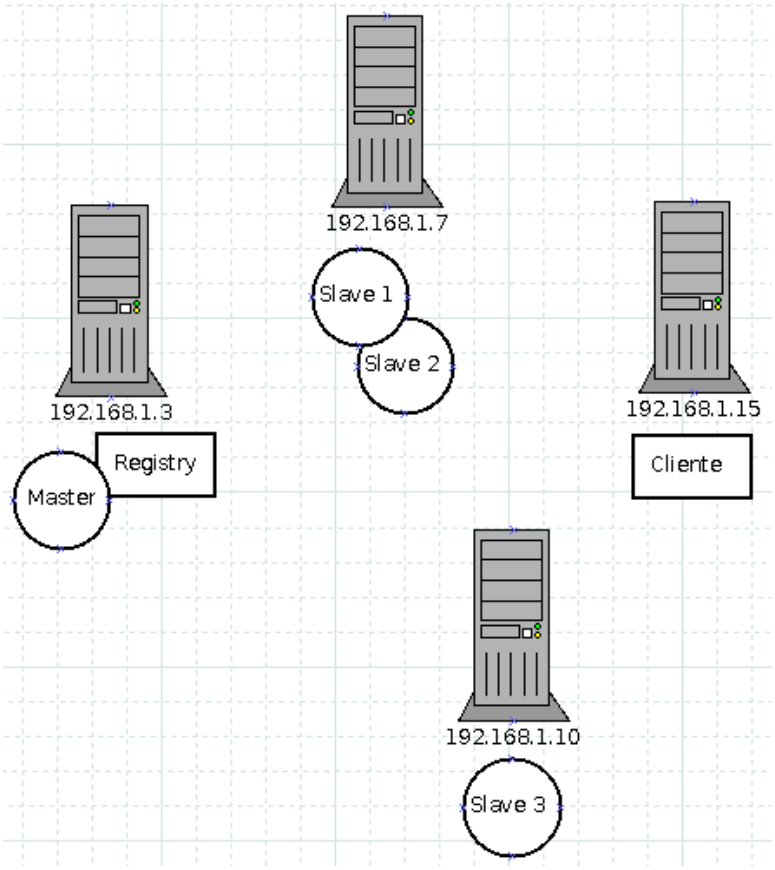
A partir de la versión 1.5 de Java es muy sencillo trabajar con objetos remotos (objetos que están distribuidos en diferentes máquinas). A esto lo llamaremos RMI: Remote Method Invocation; o lo que es lo mismo: Invocación Remota de Métodos.

En este tutorial vamos a hacer un pequeño resumen de como funciona RMI, y vamos a contestar a una de las grandes preguntas que rondan por internet ¿como puedo registrar objetos en un Registry remoto (un Registry que está en una máquina diferente en la red a donde se creó la instancia del objeto remoto)?

Para poder contestar a esta preguntas primero tenemos que explicar un poco como funciona RMI. Como ya hemos dicho antes, RMI es el mecanismo que me va a permitir invocar métodos de objetos que están en otras máquinas de la red, como si estuvieran en local. Para poder hacer esto primero tengo que conseguir una referencia a ese objeto remoto (es decir, una referencia a ese objeto que esta en otra máquina en la red). Normalmente para conseguir esta referencia usaremos un Registry (un registro), que es como unas páginas amarillas, donde yo le digo el nombre del objeto y el me devuelve la referencia al objeto remoto. Una vez tengo esa referencia puedo invocar los métodos como si el objeto estuviera en local, aunque realmente se hará una llamada por la red y el método se ejecutará en al máquina donde reside en el objeto.

Ojo porque hemos dicho que la ejecución se hará donde reside el objeto (en la máquina donde se creo la instancia del objeto), y no donde reside el Registry, que como vamos a demostrar en este tutorial pueden ser máquinas distintas.

La siguiente imagen muestra lo que pretendemos conseguir:



Vemos como tenemos 4 máquinas:

- 192.168.1.3 Tendrá el objeto remoto "Master", y también tendrá el "Registry" donde registraremos todos los objetos remotos.
- 192.168.1.7 Tendrá dos objetos remotos: "Slave 1" y "Slave 2".
- 192.168.1.10 Tendrá el objeto remoto "Slave 3"

- 192.168.1.15 No tendrá ningún objeto remoto, tendrá un programa cliente que se le preguntará al "Registry" cuantos objetos tiene registrados y luego los irá llamando uno a uno. En la consola de cada máquina veremos como la ejecución se hace en cada una de las máquinas.

1.1. ¿Como registrar objetos en un Registry creado en otra máquina?

`Registry` es una interfaz proporcionada por la máquina virtual (`java.rmi.Registry`). La implementación de esta interfaz que proporciona la maquina virtual de Sun tiene una restricción los métodos `bind` (para añadir un objeto al `Registry`), `rebind` (para volver a añadir un objeto con el mismo nombre que la vez anterior), y `unbind` (para eliminar un objeto del `Registry`). La restricción consiste en que, por motivos de seguridad, estos métodos van a lanzar la excepción `java.rmi.AccessException` si se intentan invocar en una máquina que no es donde "reside" realmente el `Registry`.

Es decir, el `Registry` es también un objeto remoto, y como tal, podemos invocar sus métodos desde máquinas en la red distintas a la máquina donde se creó la instancia del `Registry`. Si es el caso, los métodos `bind`, `rebind` y `unbind`, nos darán un una excepción diciendo que se está intentando registrar un objeto desde una máquina que no es "localhost".

Da igual como configuremos la seguridad de la máquina virtual, siempre obtendremos el mismo error. En Internet dicen muchas cosas al respecto, las he probado todas y ninguna funciona, así que por favor, si alguien sabe alguna fórmula mágica que tenga la seguridad de que funciona, por favor que me la diga ;)

Entonces, ¿como vamos a poder registrar un objeto en un `Registry` que no está en local? Fácil, dejaremos que el registro lo haga otro que si esté en la misma máquina que el `Registry`.

En nuestro ejemplo será el `Master`, que si está físicamente en la misma máquina que el `Registry`, el que se encargará de registrar a los `Slave`s. Para esto crearemos en el `Master` un método donde le pasemos como parámetro la instancia del `Slave`. Este método del `Master` se encargará de registrar esa instancia del `Slave` que le han pasado como parámetro.

Cuando llamamos a métodos de un objeto remoto las parámetros del método se serializan para viajar por la red. Pero si en el método de un objeto remoto estamos pasando la instancia de otro objeto remoto, no os preocupéis, porque lo que se serializa no es todo el objeto, sino un proxy que se encarga de representarlo y redirigir las llamadas a la máquina apropiada. Con esto quiero decir que podéis tener objetos remotos que no implementen la interfaz `Serializable` o que tengan atributos que no lo sean (como `Thread` o `Connection`), ya que el objeto nunca va a llegar a viajar por la red (lo que viaja es su proxy).

Ahora que ya tenemos claro que es lo quermos hacer, vamos a ver como hacerlo con un sencillo ejemplo ;)

2. Entorno

El tutorial está escrito usando el siguiente entorno:

- Hardware: Portátil Asus G1 (Core 2 Duo a 2.1 GHz, 2048 MB RAM, 120 GB HD).
- Sistema Operativo: GNU / Linux, Debian (unstable), Kernel 2.6.23, KDE 3.5
- JDK 1.5.0_13 (instalada con el paquete sun-java5-jdk de Debian)
- Eclipse Europa 3.3, con soporte para desarrollo JEE (WTP 2.0)

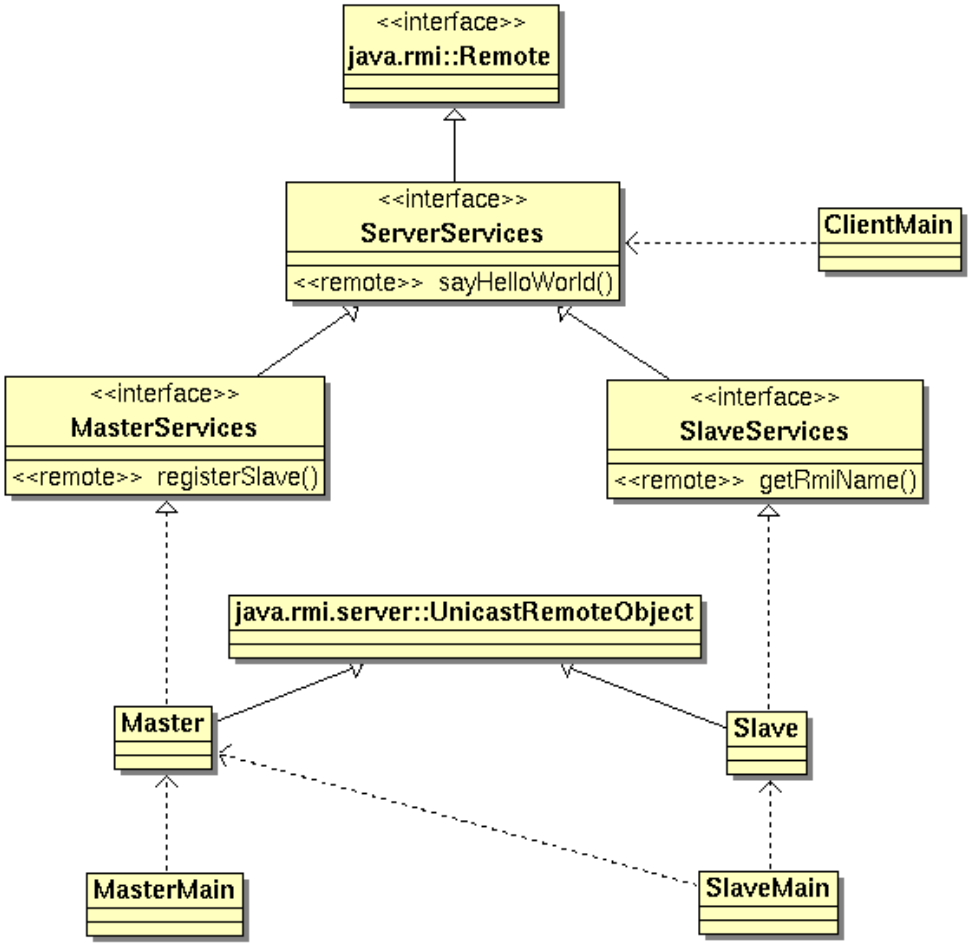
3. Creando los objetos remotos

Para crear objetos remotos que podamos usar desde otras máquinas de la red tendremos que cumplir las siguientes condiciones:

1. El objeto debe implementar una interfaz que extienda directa o indirectamente de la interfaz `java.rmi.Remote`. Esta interfaz será la que defina los método que se pueden llamar remotamente. La interfaz `java.rmi.Remote` es una interfaz marcadora. Es decir no nos obliga a implementar ningún método.
2. Los métodos de la interfaz que van a ser llamados remotamente tendrán que lanzar la excepción `java.rmi.RemoteException`. Además de esta excepción pueden lanzar también cualquier otra excepción que consideremos necesaria.
3. Es aconsejable, aunque no obligatorio, que el objeto remoto extienda de `java.rmi.server.UnicastRemoteObject`. Esto no es obligatorio, pero muy interesante, ya que este objeto nos da implementaciones especiales de algunos métodos como `equals` y `hashCode`. Si nuestra clase ya extiende otra.

   > **Nota**: En versiones anteriores a la 1.5 además de estas condiciones era necesario usar la herramienta rmic para generar las clases adicionales (los proxy, los stub, ...). A partir de la versión 1.5 esto ya no es necesario, aunque si que tendremos que hacerlo si queremos tener clientes que estén escritos con versiones anteriores a la 1.5.

Teniendo en cuenta estas condiciones vamos a crear la siguiente estructura de clases:



En el diagrama se puede ver como hemos creado tres interfaces:

- `ServerServices`: define el comportamiento común que de cualquier de los nodos de nuestro pequeño "servidor".
- `MasterServices`: añade el comportamiento que sólo sabe hacer el nodo maestro. En esta caso el maestro es capaz de registrar a los esclavos.
- `SlaveServices`: Añade el comportamiento que sólo saben hacer los nodos esclavos. En este caso los esclavos saben decir cual debe ser su nombre RMI para poder registrarlos en el `Registry`.

Todos los métodos de estas interfaces lanzan la excepción `java.rmi.RemoteException`, por eso los hemos marcado con el estereotipo "<<remote>>". Con esto cumplimos con el punto 2.

Vemos como tenemos la clase `Master` y `Slave`. Ambas extienden de `java.rmi.server.UnicastReoteObject` e implementan interfaces que indirectamente extienden de `java.rmi.Remote`, así que también cumplimos con los puntos 1. y 3.

Finalmente vemos otras tres clases:

- `MasterMain`: El programa principal que crea la instancia de `Master` y lo registra en el `Registry` creado en la misma máquina.
- `SlaveMain`: El programa principal que crea la instancia del `Slave`, busca al `Master` en el `Registry`, y lo usa para registrar la instancia del `Slave` qua acaba de crear.
- `ClientMain`: Este se limita a buscar en el `Registry` todos los objetos que sean de tipo `ServerServices` y llama al método remoto `helloWorl()`.

4. El código

He quitado los detalles que no son imprescindibles. Podéis acceder al código completo aquí.

4.1. ServerServices.java

```
public interface ServerServices extends Remote {
   public void sayHelloWorld() throws RemoteException;
}
```

4.2. MasterServices.java

```
public interface MasterServices extends ServerServices {
   public void registerSlave(SlaveServices slave) throws RemoteException;
}
```

4.3. SlaveServices.java

```
public interface SlaveServices extends ServerServices {
   public String getRmiName() throws RemoteException;
}
```

4.4. Master.java

```
public class Master extends UnicastRemoteObject implements MasterServices {

     ...

   public void registerSlave(SlaveServices slave) throws RemoteException {
      String rmiName;
      try {
         rmiName = slave.getRmiName();
         registry.rebind(rmiName, slave);
      } catch (RemoteException e) {
         e.printStackTrace();
         throw e;
      }
      System.out.println("Registered: " + rmiName);
   }

   public void sayHelloWorld() throws RemoteException {
      System.out.println("I'm de Master, and I'm unique !!!");
   }
}
```

Aquí tenemos el quid de la cuestión. Vemos como en el método `registerSlave` toma como parámetro un `SlaveServices`. Esto es muy importante: el parámetro tiene que ser del tipo de la interfaz, y no de la clase. En este método vemos como se llama al `registry` (es un atributo, ver el código completo) para hacer un `rebind`. Como este `rebind` lo está ejecutando el `Master`, que sí está en la misma máquina que el `Registry`, no hay ningún problema.

4.5. Slave.java

```
public class Slave extends UnicastRemoteObject implements SlaveServices {

     ...

   public void sayHelloWorld() throws RemoteException {
      System.out.println("I'm the slave " + getRmiName() + ", and I say: Hellow world !!!");
   }

   public String getRmiName() throws RemoteException {
      return rmiName;
   }
}
```

4.6. MasterMain.java

```
public class MasterMain {

   public static void main(String[] args) throws Exception {
      final Registry registry = LocateRegistry.getRegistry(Registry.REGISTRY_PORT);
      Master master = new Master(registry);
      registry.rebind(Master.RMI_NAME, master);
      ...
   }
}
```

Podemos ver como se localiza el `Registry`, se crea la instancia del `Master`, y se hace `rebind` de esta instancia en el `Registry`. A partir de este momento ya podemos localizar esta instancia a través del `Registry`.

Si la clase `Master` no extendiera de `java.rmi.server.UnicastRemoteObject`, entonces tendríamos que hacer lo siguiente para crear y registrar la instancia:

```
      ...
      Master master = new Master(registry);
      Remote remote = UnicastRemoteObject.exportObject(master, 0);
      registry.rebind(Master.RMI_NAME, remote);
```

```
...
```

Fijaros como el la instancia del `Master` la creamos igual, pero tenemos que hacer el `exportObject` a mano.

4.7. SlaveMain.java

```
public class SlaveMain {

   public static void main(String[] args) throws Exception {
      final String remoteHost = args[0];
      final String slaveRmiName = args[1];

      final Registry registry = LocateRegistry.getRegistry(remoteHost, Registry.REGISTRY_PORT);
      final Slave slave = new Slave(slaveRmiName);

      ...

      System.out.println("Register " + slave.getRmiName() + " through the Master.");
      final MasterServices master = (MasterServices)registry.lookup(Master.RMI_NAME);
      master.registerSlave(slave);
   }
}
```

Igual que con el `Master`, se crea la instancia del `Slave`, pero en este caso no se registra en el `Registry`, sino que se localiza al `Master` y se llama al método `registerSlave()` para que sea el `Master` el que se encargue de registrar la instancia del `Slave`.

En el código completo os he puesto el intento de registrar el `Slave` directamente para que veáis como lanza la excepción (acordaros de que lo tenéis que ejecutar en otra máquina).

4.8. ClientMain.java

```
public class ClientMain {

   public static void main(String[] args) throws Exception {
      final String remoteHost = args[0];

      final Registry registry = LocateRegistry.getRegistry(remoteHost, Registry.REGISTRY_PORT);
      final String[] remoteObjNames = registry.list();

      for (String remoteObjName : remoteObjNames) {
         Object obj = registry.lookup(remoteObjName);
         if (obj instanceof ServerServices) {
            System.out.println("Calling remote object: " + remoteObjName);
            final ServerServices server = (ServerServices)obj;
            server.sayHelloWorld();
         }
      }
   }
}
```

El cliente se recorre todos los objetos remotos registrados en el `Registry`, y si son de tipo de `ServerServices`, entonces invoca el método `sayHelloWorld()`, independientemente de si se trata del `Master` o de un `Slave`.

Al ejecutar el cliente veremos como en la consola de cada máquina sale el mensaje correspondiente. Con esto vemos claramente como cada objeto se ejecuta en la máquina en la que "nació".

5. Ejecutando el ejemplo

Copiaremos las clases compiladas en cada una de las máquina y nos situaremos en el directorio donde hemos copiado las clases (al hacer `ls` deberíamos ver el directorio `com`). Y ejecutaremos las siguientes instrucciones en la línea de comandos.

5.1. Ejecutando el Master

```
$ rmiregistry &
$ java -cp . -Djava.rmi.server.hostname=192.168.1.3 com.autentia.rmi.MasterMain
```

Primero arrancamos el registro con el comando `rmiregistry` (el registro también se puede crear programáticamente, pero en tal caso no he sido capaz de conseguir registrar los objetos remotos, así que como siempre os digo, si alguien sabe como, que me lo cuente ;)

Con `-Djava.rmi.server.hostname` estamos especificando el nombre o ip de la máquina donde estamos ejecutando esta máquina virtual. Esto es necesario para que los objetos remotos sepan en que máquina se tienen que ejecutar. Si no lo especificamos veremos que se pone por defecto 127.0.0.1, con lo que luego no será capaz de encontrar la máquina donde tiene que hacer la ejecución. Recomiendo su utilización.

5.2. Ejecutando el Slave

```
$ java -cp . -Djava.rmi.server.hostname=192.168.1.7 com.autentia.rmi.SlaveMain 192.168.1.3 Slave-1
```

Para ejecutar el `Slave`, le pasamos como primer parámetros la ip donde reside el `Master` y el `Registry`, y como segundo parámetro el nombre con el que se va a registrar la instancia del `Slave` en el `Registry`.

5.3. Ejecutando el Client

```
$ java -cp . com.autentia.rmi.ClientMain 192.168.1.3
```

Si os fijáis, en este caso no hemos especificado la variable `-Djava.rmi.server.hostname`. En esta caso no es necesario ya que esta máquina no va exportar objetos.

6. Conclusiones

Si queremos hacer aplicaciones distribuidas y no tenemos o no podemos usar un servidor de aplicaciones (no todo tiene porque ser aplicaciones web), RMI se presenta como una de las mejores opciones, ya que nos aisla de todo el tejemaneje que habría que hacer con los sockets.

Además hemos visto como, a partir de la versión 1.5 de la máquina virtual, es muy sencillo declarar objetos remotos ya que no tenemos que hacer uso de la herramienta `rmic`.

7. Sobre el autor

Alejandro Pérez García, Ingeniero en Informática (especialidad de Ingeniería del Software)

Socio fundador de Autentia (Formación, Consultoría, Desarrollo de sistemas transaccionales)

mailto:alejandropg@autentia.com

Autentia Real Business Solutions S.L. - "Soporte a Desarrollo"

http://www.autentia.com

Servicio de notificaciones:

Si deseas que te enviemos un correo electrónico cuando introduzcamos nuevos tutoriales.

Formulario de subcripción a novedades:

E-mail [ ] Aceptar

Otros Tutoriales Recomendados.

Nombre corto	Descripción
Despliegue gráfico de EJBs	Os mostramos como crear y desplegar de un modo gráfico un EJB de sesión el el servidor de aplicaciones de referencia de Sun
Dali JPA Tools: O/R mappings para EJB 3.0	En este tutorial aprenderemos a usar Dali JPA tools. Una herramienta que nos facilita el mapeo de objetos con la fuente de datos (O/R mappings) para EJB 3.0
Comparativa entre EJB3 y Spring	En este tutorial os mostramos una comparativa entre EJB3 y Spring esperando que os ayude a decidir qué tecnología utilizar.
EJB 3.0 y pruebas unitarias con Maven, JUnit y Embedded JBoss	En este tutorial Alejandro Pérez nos enseña como realizar test unitarios sobre EJB 3.0. Para ello se usará Maven, JUnit y Embedded JBoss
EJB 3.0: Resurrection	Este turorial nos va a presentar las nuevas funcionalidades que nos aportan los EJB 3.0.
EJB 3.0, un ejemplo práctico con Maven y JBoss	Este tutorial presenta un ejemplo sencillo donde se verá como desarrollar EJBs de sesión y de entidad, inyección de dependencias, llamar a los EJBs desde una aplicación Web, definición de un DataSource, y como configurarlo y hacerlo funcionar en JBoss, y
Activación automatica de servicios RMI	En este tutorial veremos cómo utilizar el servicio de activación de objetos RMI directamente desde el cliente de forma automática
EJB´s y Orion	Recreación de la guía paso a paso de como crear una aplicación Web con EJB´s y Servlets y su despliegue con ANT sobre Orion
Aplicación básica con RMI	Gracias a este tutorial, podreis aprender paso a paso como crear una aplicación cliente-servidor con RMI
Interceptando un EJB en JBoss	En este tutorial os vamos a enseñar la arquitectura de EJBs en JBoss y a como modificarla, insertando un interceptor propio dentro de la cadena de interceptores del Proxy Cliente.

Ver todos los tutoriales

Nota:

Los tutoriales mostrados en este Web tienen como objetivo la difusión del conocimiento.

Los contenidos y comentarios de los tutoriales son responsabilidad de sus respectivos autores.

En algún caso se puede hacer referencia a marcas o nombres cuya propiedad y derechos es de sus respectivos dueños. Si algún afectado desea que incorporemos alguna reseña específica, no tiene más que solicitarlo.

Si alguien encuentra algún problema con la información publicada en este Web, rogamos que informe al administrador `rcanales@adictosaltrabajo.com` para su resolución.