Table of Contents

[1 Despliegue 1](#_Toc217389869)

[2 Características Glassfish 2](#_Toc217389870)

[2.1 Grizzly 2](#_Toc217389871)

[2.2 JavaWebStart 2](#_Toc217389872)

[2.3 Opciones de Configuración 2](#_Toc217389873)

[2.3.1 Application Server 2](#_Toc217389874)

[2.3.2 Aplicaciones Empresariales, Aplicaciones Web, Servicios Web, EJB, Clientes,… 2](#_Toc217389875)

[2.3.3 Recursos 2](#_Toc217389876)

[2.3.4 Contenedor Web 2](#_Toc217389877)

[2.3.5 Contenedor EJB 2](#_Toc217389878)

[2.3.6 Servicio JMS 3](#_Toc217389879)

[2.3.7 Seguridad 3](#_Toc217389880)

[2.3.8 Servicio HTTP 3](#_Toc217389881)

[2.3.9 ORB 3](#_Toc217389882)

[2.3.10 Subprocesos 3](#_Toc217389883)

[2.3.11 Servicio de Administración 4](#_Toc217389884)

[3 JSF y AJax 4](#_Toc217389885)

[3.1 GWT 4](#_Toc217389886)

[3.2 AdHoc 4](#_Toc217389887)

[3.3 Dojo o similares 4](#_Toc217389888)

[3.4 Ajax Blue Prints 5](#_Toc217389889)

[3.5 Dynamic Faces 5](#_Toc217389890)

[3.5.1 NOTAs: Sobre Ajax Zones 5](#_Toc217389891)

[3.6 Woodstock 7](#_Toc217389892)

[3.7 IceFaces 8](#_Toc217389893)

[4 WebServices 8](#_Toc217389894)

[5 EJBs 9](#_Toc217389895)

[5.1 Despliegue 9](#_Toc217389896)

[5.2 Ejecución (Servlet) 10](#_Toc217389897)

[5.2.1 Demo (EJB alojada en Glassfish e invocada desde Tomcat) 11](#_Toc217389898)

[5.2.2 EJB Remoto y Local 12](#_Toc217389899)

[5.2.3 Statefull y Stateless 12](#_Toc217389900)

[5.3 Ejecución (Cliente) 12](#_Toc217389901)

[6 JMS 13](#_Toc217389902)

[6.1 Infraestructura 13](#_Toc217389903)

[6.2 Demos 13](#_Toc217389904)

[6.2.1 Sesión 13](#_Toc217389905)

[6.2.2 Acknowledge 14](#_Toc217389906)

[6.2.3 Productor 15](#_Toc217389907)

[6.2.4 Consumidor 15](#_Toc217389908)

[6.2.5 Subscripciones “duraderas” 16](#_Toc217389909)

[6.2.6 Otros factores 16](#_Toc217389910)

[6.2.7 Destinos temporales 16](#_Toc217389911)

[6.2.8 Mensajes 17](#_Toc217389912)

[7 Seguridad 17](#_Toc217389913)

[7.1.1 Básica 17](#_Toc217389914)

[7.1.2 Forms 17](#_Toc217389915)

[8 Otros. DataBind 17](#_Toc217389916)

[9 Tomcat 17](#_Toc217389917)

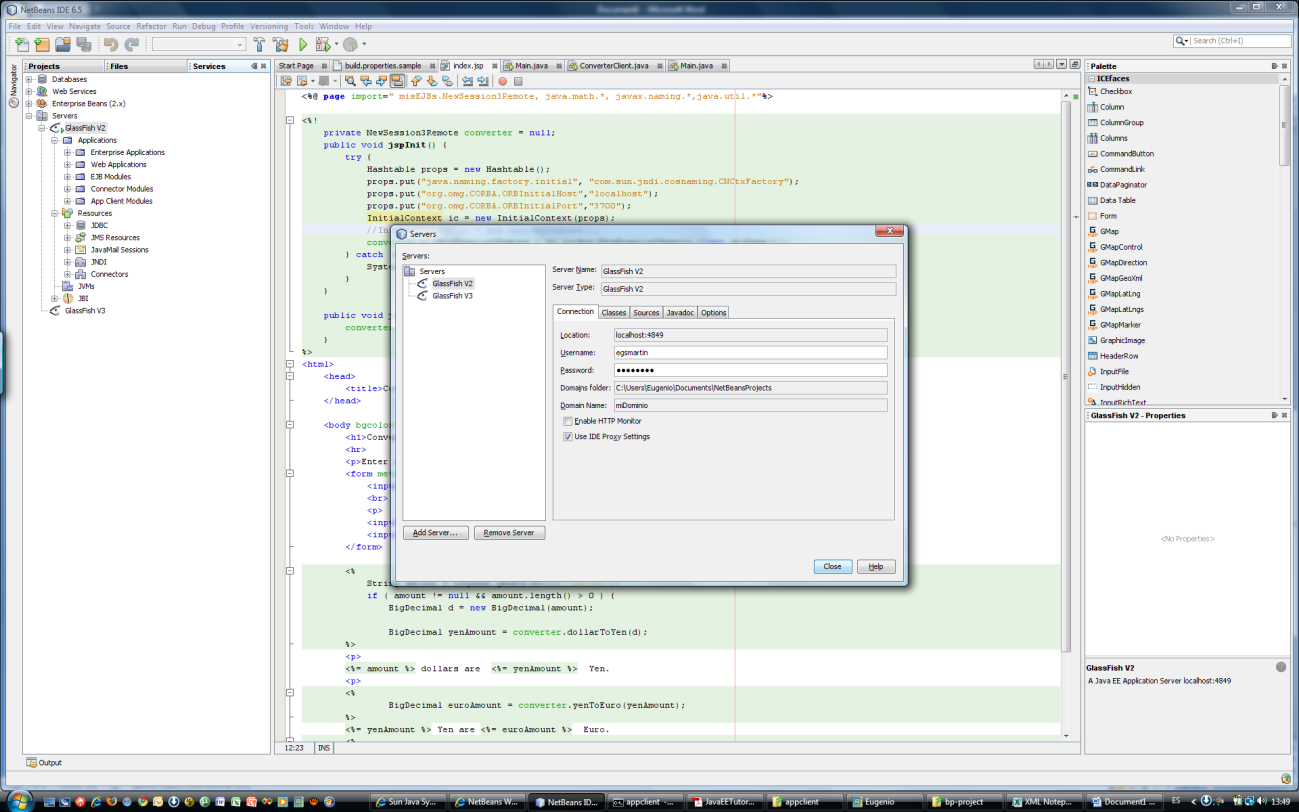
# Despliegue

Las aplicaciones pueden desplegarse de distintas formas:

1. Usando asant “target”. Esta utilidad que viene con glassfish hace uso de ant. Toma el build.xml (se debe ejecutar en el directorio donde esté el build.xml) y aplica las actuaciones que correspondan al “target”. Si no se informa target aplica el por defecto. Valores habituales son “build”, “run”, “deploy”, pero no se trata de ningún estándar, hay que fijarse en el build.xml. En el build.xml, o en algún archivo externo (que se importa en el build.xml) se tienen que definir los datos de conexión (URL y puerto) y usuario, donde debe desplegarse la aplicación (y ejecutarse). En los ejemplos del tutorial de J2EE en C:\Users\Eugenio\Desktop\javaeetutorial5\examples\bp-project\build.properties. Si esto no está bien configurado es posible que nos compile todo bien, pero que no podamos desplegar y ejecutar



1. Dejando el jar, war,… en la carpeta autodeploy
2. Desplegando desde la consola de administración de glassfish
3. Desde Netbeans (En Netbeans podemos incluir distintos plugins con diferentes servidores de aplicaciones. Glassfish es uno de ellos. En este caso nos estamos conectando con un dominio concreto, de ahí que Netbeans sepa la url en la que debe desplegar y desde la que debe arrancar las aplicaciones)



Se tienen dos plugins con las versiones 2 y 3 de Glass Fish. Se puede arrancar, parar… el servidor (con el botón derecho), y se ven todas las cosas que se han desplegado. Es una vista reducida de lo que se vería en la consola.

Obsérvese como se especifica la URL de administración del dominio miDominio, junto con las credenciales de acceso definidas para este dominio, y el directorio donde está definido el dominio.



# Características Glassfish

## Grizzly

Esta es una tecnología que permite incorporar en Glassfish IO no bloqueante. De esta forma los clientes establecen una conexión con el servidor, y en lugar de responder inmediatamente, la conexión se mantiene activa hasta que podamos necesitar informar “algo” desde el servidor al cliente. Esta técnica es la que permite el “push Ajax” que se utiliza en IceFaces (y seguramente en las versión 2.0 de JSF).



NOTA: Hay un ejemplo en C:\Users\Eugenio\Documents\NetBeansProjects\EjemploICEFAces

## JavaWebStart

Tecnología equivalente a ClickOnce.

## Opciones de Configuración

### Application Server

Acceso a diferentes datos del servidor, entre ellos podemos ver al árbol JNDI.

### Aplicaciones Empresariales, Aplicaciones Web, Servicios Web, EJB, Clientes,…

### Recursos

Se pueden definir conexiones JDBC, JMS, JNDI personalizadas…

### Contenedor Web

No hay mucho que configurar

### Contenedor EJB

Configura las propiedades del contendor, su cache, time-out, …



### Servicio JMS

Permite configurar un servicio JMS, indicar cual es el host predeterminado, y crear nuevos Host JMS, y definir destinos físicos (subscripciones y colas).

Para cada Host se define el puerto y el usuario de administración:

### Seguridad

Se permite definir los distintos realm que podemos utilizar.

Se especifica también el realm por defecto, y el usuario y password que queremos usar si necesitaramos impersonar.

### Servicio HTTP

Se permite definir Listeners HTTP y servidores virtuales. Los servidores virtuales se asocian con un nombre simbólico o una IP (bueno, uno o varios nombres simbólicos y/o direcciones de red). Hay un servidor virtual “harcodeado”, el \_\_asadmin, que es el encargado de la administración del Glassfish.

Los listeners definen puertos, y se asocian a servidores virtuales. También permiten que se especifique una dirección IP, aunque el valor por defecto, 0.0.0.0 significa cualquier dirección IP.

La convinación de Listeners y servidores virtuales determina los distintos IP:puertos por los que se escuchan peticiones HTTP (y donde se pueden publicar aplicaciones web, WS,…)

### ORB

Recibe peticiones de resolución de nombres (JNDI, IIOP). Las escuchas IIOP definen los canales por los que se pueden recibir peticiones de resolución de nombres.

Cada escucha es una IP (si el valor 0.0.0.0 significa que se escucha de cualquier IP) y un puerto.

### Subprocesos

Define el pool de threads del servidor

### Servicio de Administración

Configura el servicio de administración remota (puerto, IP, realm utilizado para la autentificación).

NOTA: Hay documentación sobre Glassfish (guias de uso, administración, desarrollo, tunning,…) en C:\Users\Eugenio\Documents\Redbooks\Java\GlassFish.

## Archivos de Configuración

|  |  |
| --- | --- |
| C:\glassfish\config\asenv | Variables de entorno |
| C:\glassfish\setup | Variables que configuran las propiedades generales del servidor |
| C:\glassfish\registry.xml | Información de registro del producto |
| C:\glassfish\domains\domain1\config\domain.xml | Configuración del Dominio. |

# JSF y AJax

Hay varias técnicas para incorporar Ajax en páginas JSF:

* GWT (Google Web Toolkit)
* AdHoc
* Dojo o similares
* Ajax Blue Prints
* Dynamic Faces
* Woodstock
* IceFaces

## GWT

Se trata de un framework de desarrollo de Google que permite trabajar en Java como si se tratara de una aplicación Cliente normal, incluso a efectos de depurarla. Finalmente el framework es capaz de generar javascript que “ajaxifica” el cliente.

En ejemploGWT se observa un ejemplo de uso del GWT, incluyendo la invocación de servicios desde el cliente. Hay un plugin de GWT que se tiene que instalar en el Netbeans para poder trabajar desde Netbeans.

## AdHoc

Desarrollar desde el ajax la conexión HTTP, serialización y el callback con la respuesta. En el servidor crear el servlet (o el handler .Net) que se encargue de recibir la petición y devolver la contestación al callback. Es un mecanismo muy básico y poco productivo.



## Dojo o similares

Utilizar librerías de componentes javascript que incorporar además de todo lo relativo a la presentación, librerías que permiten realizar peticiones Ajax. En lo que se refiere a Ajax estamos en un punto muy similar al anterior. La mejora esta en los controles que podemos utilizar con toda la codificación javascript.

## Ajax Blue Prints

Lo primero, indicar que es una tecnología obsoleta, y que la última versión de Netbeans (6.5) no lo soporta, porque las librerías que se precisan para los BP son antiguas.

Basada en Dojo, incorpora la novedad de proporcionar componentes JSF que pueden incorporarse en el canvas desde la paleta de diseño.

C:\Users\Eugenio\Documents\NetBeansProjects\bpcatalog-ee5-ea-0.8

## Dynamic Faces

Introduce dos mecanismos para Ajaxizar páginas JSF. Las transacciones y las Ajaz Zones.

En el ejemplo que se adjunta (C:\Users\Eugenio\Documents\NetBeansProjects\AutoComplete) se muestra el uso de transacciones Ajax:



### NOTAs: Sobre Ajax Zones

Definitivamente el Ajax Zone no funciona con webuijsf, esto es, solo funciona con ui y h[[1]](#footnote-1). Esto significa que hay que trabajar con la librería obsoleta de Sun ui si queremos utilizar los AjaxZones. Es posible que con el JSF 2.0 que se incluye en la EE6 se resuelva.

Si no nos creemos, aquí está la prueba. El mismo software, uno bajo con ui funciona:

C:\Users\Eugenio\Documents\NetBeansProjects\CurrencyTrader

Y el otro con webui no funciona (paleta Basic):

C:\Users\Eugenio\Documents\NetBeansProjects\CurrencyTrader JSF12

Las librerías que están incluidos en ambos proyectos son las mismas, la diferencia reside en la configuración de los proyectos netbean. Esto no se puede configurar con el IDE, y hace que las paletas de componentes sean unas u otras (Básica o Woodstock):

En el project.xml, se puede ver este extracto. Con esto la paleta que se usa es la Woodstock (este es el que genera por defecto el Netbeans 6.5):

<web-module-libraries>

<library>

<file>${libs.jsf12-support.classpath}</file>

<path-in-war>WEB-INF/lib</path-in-war>

</library>

<library>

<file>${libs.woodstock-components.classpath}</file>

<path-in-war>WEB-INF/lib</path-in-war>

</library>

<library>

<file>${libs.woodstock-theme-default.classpath}</file>

<path-in-war>WEB-INF/lib</path-in-war>

</library>

<library>

<file>${libs.dynamicfaces\_58593.classpath}</file>

<path-in-war>WEB-INF/lib</path-in-war>

</library>

<library>

<file>${libs.rowset-ri.classpath}</file>

<path-in-war>WEB-INF/lib</path-in-war>

</library>

</web-module-libraries>

Estos son los módulos que se usan el CurrencyTrader donde los AjaxZones funciona (este es el que genera por defecto el Netbeans 6.1):

<web-module-libraries>

<library files="4">

<file>${libs.jsfsupport-designtime.classpath}</file>

</library>

<library files="2">

<file>${libs.webui-designtime.classpath}</file>

</library>

<library files="1">

<file>${libs.jdbcsupport-designtime.classpath}</file>

</library>

<library files="1">

<file>${libs.rowset-ri.classpath}</file>

<path-in-war>WEB-INF/lib</path-in-war>

</library>

<library files="5">

<file>${libs.dynamicfaces\_58593.classpath}</file>

<path-in-war>WEB-INF/lib</path-in-war>

</library>

<library files="1">

<file>${libs.theme-default-green.classpath}</file>

<path-in-war>WEB-INF/lib</path-in-war>

</library>

</web-module-libraries>

Si abrimos el proyecto generado con el Netbeans 6.1 en el Netbeans 6.5 todo funciona. Esto significa que podríamos manipular el Project.xml generado en la 6.5 si queremos usar los AjaxZones.



## Woodstock

En los nuevos componentes JSF se introducen ciertas propiedades Ajax (en los textbox se permite la validación Ajax, de modo que sin hacer un post podemos validar el contenido del textbox).

C:\Users\Eugenio\Documents\NetBeansProjects\Woodstock



## IceFaces

Estos componentes son sin duda ninguna los mejores. Son Ajax nativos y además introducen una nueva capacidad, el Ajax push.



NOTA: Hay un ejemplo en C:\Users\Eugenio\Documents\NetBeansProjects\EjemploICEFAces

# WebServices

El web service se crea haciendo un uso extensivo de anotaciones:

//Declara el Web Service

@WebService()

public class CalculatorWS {

//Declara un web method. La operación es la que figurará en el wsdl

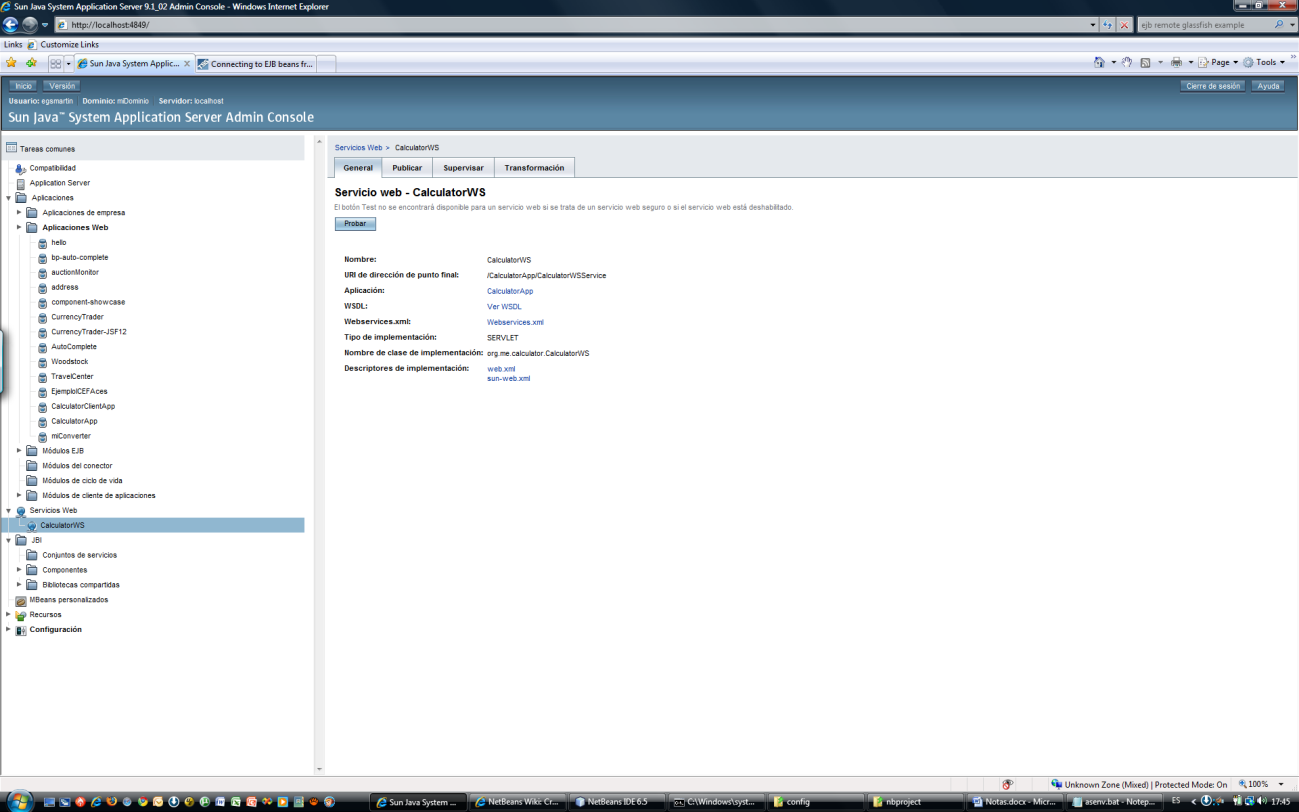
@WebMethod(operationName = "add")

public int add(@WebParam(name = "i") int i, @WebParam(name = "j") int j) {

return i + j;

}

Nótese que a diferencia de en .Net, la clase del web service es una clase base.



Se da acceso al descriptor del WS, el WSDL

Los WS se muestran en este nodo.

El WDSL está en <http://localhost:8081/CalculatorApp/CalculatorWSService?wsdl>

Para invocar al web service, primero establecemos la referencia al ws:

@WebServiceRef(wsdlLocation = "http://localhost:8081/CalculatorApp/CalculatorWSService?wsdl")

public CalculatorWSService service;

Luego, hay que crear una instancia a el Puerto del ws, para desde ahí poder invocar a las distintas operaciones:

//Obtiene el puerto del servicio

org.me.calculator.client.CalculatorWS port = service.getCalculatorWSPort();

int result = port.add(i, j);

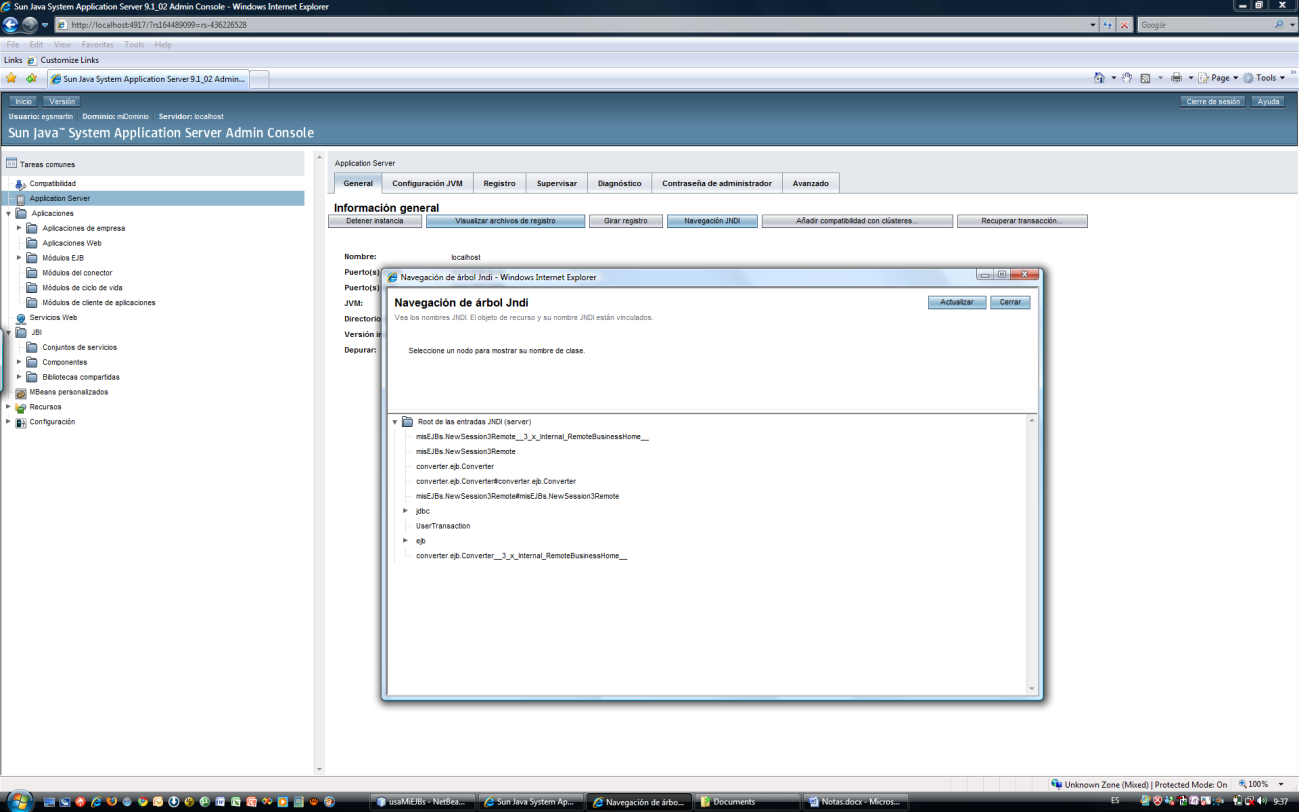
Viendo el WDSL se ve claramente esta estructura de WS->Puerto->Direccion y Binding. Con el Binding->operaciones->mensajes

Este ejemplo está en C:\Users\Eugenio\Documents\NetBeansProjects\Calculator

# EJBs

## Despliegue

Creamos un proyecto EJB. Hemos definido una stateless remota. Una vez concluido el despliegue, además de ver el EJB bajo el nodo de EJBs, en el árbol JNDI debe aparecer el componente, de modo que pueda luego ser buscado:



El EJB que hemos desplegado se muestra en el nodo (tanto el interface como la propia bean)

Selecciona la navegación JNDI

NOTA: En el caso de una aplicación empresarial, el modulo EJB no se muestra en el nodo EJBs, pero si se ven los detalles de la aplicación Empresarial, se verán como submodulo el EJB. En el árbol JNDI si se mostraría el EJB, y es más, si hubiera otro EJB desplegado con el mismo nombre que un EJB de la aplicación Empresarial, al ejecutar dará una excepción de ambigüedad de nombres (en una palabra, que los EJBs de las aplicaciones empresariales no se muestran en el nodo EJBs, sino que se muestran bajo el paraguas de la aplicación Empresarial, de modo que cuando esta se desinstala, también se desinstala el EJB. Al resto de efectos, estos EJBs son iguales a los desplegados directamente como EJBs).

## Ejecución (Servlet)

Se muestra un ejemplo en C:\Users\Eugenio\Documents\NetBeansProjects\usaMiEJBs.

En este caso ay que hacer uso del JNDI para “encontrar” el EJB, y hacer uso de él.

Se muestran en este servlet dos opciones de buscar. En una de ellas especificamos la dirección concreta del servidor donde está alojado el EJB. Esto nos permite que el Servlet se pueda ejecutar en un servidor diferente del contenedor de EJBs.

InitialContext ic = new InitialContext();

//Si no añadimos estas dos propiedades, buscará el contexto JNDI en el propio servidor donde se está ejecutando la

//petición. Si especificamos estos valores, hacemos posible que el servlet y el contenedor EJB esten en máquinas distintas.

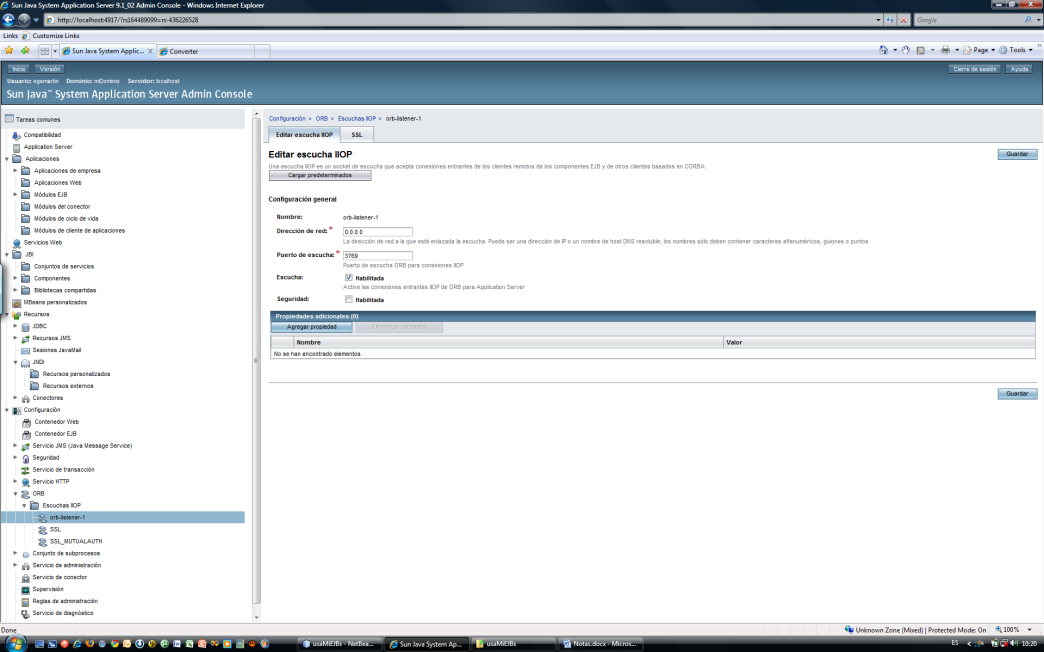
ic.addToEnvironment("org.omg.CORBA.ORBInitialPort","3769");

ic.addToEnvironment("org.omg.CORBA.ORBInitialHost","localhost");

//Nótese que lo que buscamos es el interfaz

converter = (NewSession3Remote ) ic.lookup(NewSession3Remote.class.getName());

El Puerto lo podemos ver en las propiedades del Glassfish:



### Demo (EJB alojada en Glassfish e invocada desde Tomcat)

El ejemplo puede verse en C:\Users\Eugenio\Documents\NetBeansProjects\usaMiEJBs y C:\Users\Eugenio\Documents\NetBeansProjects\misEJBs.

En el documento siguiente se indican los pasos a seguir.



La clave está en crear un contexto JNDI que apunte al contexto donde se alojan las EJBs. En el documento se indican las propiedades que deben especificarse al crear el contexto.

La clave, como indica el documento, es que en el classpath del TomCat se tiene que hacer referencia al appserv-rt.jar. En el documento que sigue (FAQs) se indica lo que se debe hacer para realizar la llamada y la referencia a incluir:



Los jar a los que se hace referencia incluyen la clase com.sun.enterprise.naming.SerialInitContextFactory. Si no se incluyen dará la tercera excepción que se indica en este documento:



Tal y como se indica en el documento con los pasos (el primero), la forma de que desde Tomcat se haga referencia a los jar es la más compleja.

Lo que hacemos es copiar en el directorio

C:\Program Files (x86)\Apache Software Foundation\Tomcat 5.5\shared\lib

Las cuatro librerías:

appserv-deployment-client.jar  appserv-ext.jar  appserv-rt.jar  javaee.jar

que podemos encontrar en C:\Program Files (x86)\glassfish-v2ur2\lib.

### EJB Remoto y Local

Si tratamos de ejecutar una EJB definida como local en remoto, fallará porque la EJB definida como local no se publica en el árbol JNDI. Ni siquiera llegamos a ejecutar, simplemente al buscarla con lookup no la encontraremos.

### Statefull y Stateless

La diferencia en este caso está en el ciclo de vida. En el caso de que el EJB sea Statefull hay dos eventos más en el ciclo de vida, pre-pasivado, y post-activado. Además de estos eventos, hay otros dos que son comunes con las Stateless, el post-creado y el pre-destrucción.

En la NewSessionBean4 mostramos el ciclo de vida.

## Ejecución (Cliente)

Hay dos formas de ejecutar desde una Aplicación Cliente, ejecutando contra la VM del propio servidor de aplicaciones (ej. Glassfish), o ejecutando desde una VM propia.

En el primero de los casos la aplicación se despliega en el servidor (Aplicaciones de Cliente), y se permite la inyección de recursos (basta un @EJB para que se busque en el árbol JDNI del servidor en el que se ejecuta la aplicación.

En el segundo de los casos precisará que el lookup se especifiquen todos los datos del contexto JNDI.

A nivel de Netbeans la primera aplicación es una aplicación cliente EJB (disponible dentro de la opción J2EE), y en el segundo es una aplicación cliente estándar.

Para la ejecución del primer caso se usará el script clientapp que se incluye en la distribución del Glassfish (naturalmente, siempre se puede ejecutar desde el propio Netbeans).

Los ejemplos que tenemos de uno y otro tipo están en:

C:\Users\Eugenio\Documents\NetBeansProjects\clienteUsaMiEJBS

C:\Users\Eugenio\Documents\NetBeansProjects\clienteUsaMiEJBs\_bis

# JMS

## Infraestructura

Glassfish incluye un bundle con el servidor JMS de Sun. Este bundle se instala en:

C:\Program Files (x86)\glassfish-v2ur2\imq

El servicio puede adminsitrarse desde la propia consola de administración de Glassfish, pero desde esta consola se ofrecen menos opciones. La otra posibilidad es utilizar los servicios que encontramos en C:\Program Files (x86)\glassfish-v2ur2\imq\bin

Imqbrokerd permite arrancar el servicio JMS. Los datos de la configuración del servicio los tenemos en C:\Program Files (x86)\glassfish-v2ur2\imq\etc (con la opción –port xxxx, donde xxx es el número del puerto donde se atienden las peticiones).



## Demos

El proyecto Empresarial clientsessionmdb muestra un ejemplo de utilización de un cliente para publicar datos de una subscripción, y un Messaging EJB. Otro ejemplo a revisar es durablesubscriberexample.



En el esquema de funcionamiento de la API es el indicado en la figura.

### Sesión

En primer lugar se debe acceder a una fábrica de conexiones. Este es un recurso publicado por el servidor JMS. Este recurso se puede referenciar así (o con el más clásico lookup JNDI):

@Resource(mappedName = "jms/ConnectionFactory")

private static ConnectionFactory connectionFactory;

Una vez tenemos este recurso, habrá que crear una sesión:

session = connection.createSession(false, Session.AUTO\_ACKNOWLEDGE);

Al crear la session se debe indicar si está bajo una transacción, y el modo de acknowledge. Hasta que un mensaje no es “acknowledge”, no se considera consumido.

### Acknowledge

In transacted sessions (see “Using JMS API Local Transactions” on page 949), acknowledgment happens automatically when a transaction is committed. If a transaction is rolled back, all consumed messages are redelivered.

In nontransacted sessions, when and how a message is acknowledged depend on the value

specified as the second argument of the createSession method. The three possible argument

values are as follows:

* Session.AUTO\_ACKNOWLEDGE: The session automatically acknowledges a client’s receipt of a message either when the client has successfully returned from a call to receive or when the MessageListener it has called to process the message returns successfully. A synchronous receive in an AUTO\_ACKNOWLEDGE session is the one exception to the rule that message consumption is a three-stage process as described earlier. In this case, the receipt and acknowledgment take place in one step, followed by the processing of the message.
* Session.CLIENT\_ACKNOWLEDGE: A client acknowledges a message by calling the message’s acknowledge method. In this mode, acknowledgment takes place on the session level: Acknowledging a consumed message automatically acknowledges the receipt of *all* messages that have been consumed by its session. For example, if a message consumer consumes ten messages and then acknowledges the fifth message delivered, all ten messages are acknowledged.
* Session.DUPS\_OK\_ACKNOWLEDGE: This option instructs the session to lazily acknowledge the delivery of messages. This is likely to result in the delivery of some duplicate messages if the JMS provider fails, so it should be used only by consumers that can tolerate duplicate messages. (If the JMS provider redelivers a message, it must set the value of the JMSRedelivered message header to true.) This option can reduce session overhead by minimizing the work the session does to prevent duplicates.

En el caso de colas (P2P), si un mensaje no se “acknowledgea” cuando la sesión se cierra, el proveedor JMS mantiene el mensaje y lo enviará al próximo consumidor que se conecte a la cola.

Si se trata de una susbcripción “duradera”, el proveedor JMS retiene el mensaje sino se acknowledgea. Si la subscripción no es “duradera”, los mensajes se pierden cuando la sesión se cierra.

#### “Rollback” de los mensajes

If you use a queue or a durable subscription, you can use the Session.recover method to stop a

nontransacted session and restart it with its first unacknowledged message. In effect, the

session’s series of delivered messages is reset to the point after its last acknowledged message.

#### Ejemplo

The **AckEquivExample**.java program shows how both of the following two scenarios ensure

that a message will not be acknowledged until processing of it is complete:

* Using an asynchronous message consumer (a message listener) in an AUTO\_ACKNOWLEDGE session
* Using a synchronous receiver in a CLIENT\_ACKNOWLEDGE session

With a message listener, the automatic acknowledgment happens when the onMessage method returns (that is, after message processing has finished).

With a synchronous receiver, the client acknowledges the message after processing is complete:

receiver = session.createConsumer(queue);

connection.start();

message = (TextMessage) receiver.receive();

//Se hace el acknowledge del mensaje

message.acknowledge();

If you use AUTO\_ACKNOWLEDGE with a synchronous receive, the acknowledgment happens immediately after the receive call; if any subsequent processing steps fail, the message cannot be redelivered.

### Productor

Se crea el productor de mensajes especificando un destino, indicando si se trata de una cola (P2P) o de una subscripción (topic). En el caso de un P2P el mensaje se mantiene en la cola hasta que es consumido por (una sola vez). En el caso de la subscripción, hay que distinguir dos casos, la subscripción “durable” y la normal. La durable se describe más adelante.

Creamos un productor especificando el destino (una cola o un tópico)

MessageProducer producer = session.createProducer(dest);

MessageProducer producer = session.createProducer(queue);

MessageProducer producer = session.createProducer(topic);

Si el productor se crea sin especificar el destino, al enviar el mensaje habrá que indicarlo:

MessageProducer anon\_prod = session.createProducer(null);

anon\_prod.send(dest, message);

### Consumidor

El consumidor “normal” se crea de forma similar al productor.

MessageConsumer consumer = session.createConsumer(dest);

MessageConsumer consumer = session.createConsumer(queue);

MessageConsumer consumer = session.createConsumer(topic);

Destacar que la escucha de mensajes es un procedimiento bloqueante. Se puede especificar el time-out máximo de espera:

connection.start();

Message m = consumer.receive();

connection.start();

Message m = consumer.receive(1000); // time out after a second

#### Listener

En el caso de que queramos escuchar los mensajes de forma no bloqueante, existe la posibilidad de utilizar un listener (es el mecanismo utilizado en las MessageBean).

Listener myListener = new Listener();

consumer.setMessageListener(myListener);

El handler de este listener define un método onMessage: When message delivery begins, the JMS provider automatically calls the message listener’s onMessage method whenever a message is delivered. The onMessage method takes one argument of type Message, which your implementation of the method can cast to any of the other message types

### Subscripciones “duraderas”

En una subscripción normal, se notifica a todos los subscriptores que están conectados al tópico en el momento de que se publica el mensaje. Si algún subscriptor se conecta después de que se haya publicado el mensaje, no se va a enterar. Con mlas susbcripciones durables esto se soluciona. A continuación se crea una subscripción “durable” llamada MySub:

String subName = "MySub";

MessageConsumer topicSubscriber =

session.createDurableSubscriber(myTopic, subName);

La subscripción al tópico puede cerrarse, pero a diferencia de lo que sucede con una subscripción normal, los mensajes que se publiquen en el tópico desde este momento, no se perderán, los recibiremos la siguiente vez que nos conectemos al tópico:

//Se cierra la subscripción, pero sigue latente

topicSubscriber.close();

Si queremos terminar definitivamente la subscripción “duradera”, hay que cerrarla y llamar a “unsubscribe”.

topicSubscriber.close();

session.unsubscribe("MySub");

If the program or another application calls createDurableSubscriber using the same connection factory and its client ID, the same topic, and the same subscription name, the subscription is reactivated, and the JMS provider delivers the messages that were published while the subscriber was inactive.

### Otros factores

Se puede determinar si el mensaje debe persistirse o no (por defecto se persisten, de modo que aun si el proveedor JMS fallara, el mensaje no se pierda), el tiempo de vigencia del mensaje, la prioridad del mensaje

### Destinos temporales

Es posible crear colas y subscripciones temporales. Se crean con la sesión, y se pierden cuando la sesión se cierra

### Mensajes

Los mensajes se crean con la API, pudiendo definirse distintos tipos de mensajes y añadirse propiedades a los mensajes. Adicionalmente en la cabecera del mensaje se incluyen una serie de propiedades estándar (destino, timestamp,…):

# Seguridad

En el Glassfish (o en el Tocat) se determinan los realms, y cual será el utilizado por defecto. En el caso del Glassfish se muestra en el nodo Seguridad la selección de realm por defecto, y en nodos hijos la definición de todos los realms disponibles. Es posible desde aquí configurar los usuarios y roles de cada real.

En el web.xml de cada aplicación se configura la seguridad (recursos protegidos, regla de acceso a cada recurso – rol o usuario permitidos -, y el mecanismo de login (anónimo, básica, certificados, formularios,…).

En el sun-web.xml se establece el mapeo entre el usuario y/o rol con el grupo definido en el web.xml y con el que se ha definido la política de seguridad.

### Básica

Los recursos /r y /r2 del C:\Users\Eugenio\Documents\NetBeansProjects\usaMiEJBs están configurados así.

NOTA: En alguna ocasión me ha sucedido que es preciso necesario crear un nuevo navegador para que se tomen las nuevas configuraciones (no basta con crear una nueva pestaña).

### Forms

No es equivalente al mecanismo homónimo de .Net. Aquí lo que estamos haciendo es escribir en un formulario el usuario y password, en unos text-box con nombres específicos, para que de aquí se tomen y validen contra el realm.

# Otros. DataBind



# Tomcat

Las aplicaciones (con textos) que están instaladas se configuran en:

C:\Program Files (x86)\Apache Software Foundation\Tomcat 5.5\conf\Catalina\localhost

Por ejemplo, la siguiente aplicación está configurada (C:\Program Files (x86)\Apache Software Foundation\Tomcat 5.5\conf\Catalina\localhost\admin.xml):

<Context docBase="${catalina.home}/server/webapps/admin" privileged="true" antiResourceLocking="false" antiJARLocking="false">

<!--

Uncomment this Valve to limit access to the Admin app to localhost

for obvious security reasons. Allow may be a comma-separated list of

hosts (or even regular expressions).

<Valve className="org.apache.catalina.valves.RemoteAddrValve"

allow="127.0.0.1"/>

-->

</Context>

A nivel de servidor, servlets y sus mapeos, mapas MIME, archivos de bienvenida, se configuran en C:\Program Files (x86)\Apache Software Foundation\Tomcat 5.5\conf\web.xml

A nivel de servidor, los listeners http, recursos, los realms, y el directorio raíz se configuran en C:\Program Files (x86)\Apache Software Foundation\Tomcat 5.5\conf\server.xml. Por ejemplo el siguiente recursos y realm figuran por defecto:

<Resource name="UserDatabase" auth="Container" type="org.apache.catalina.UserDatabase" description="User database that can be updated and saved" factory="org.apache.catalina.users.MemoryUserDatabaseFactory" pathname="conf/tomcat-users.xml"/>

<Realm className="org.apache.catalina.realm.UserDatabaseRealm" resourceName="UserDatabase"/>

También se define el servidor virtual:

<Host name="localhost" appBase="webapps" unpackWARs="true" autoDeploy="true" xmlValidation="false" xmlNamespaceAware="false">

En cada aplicación web, en el ..\ WEB-INF\web.xml se configura la aplicación.

1. Cuando hablamos de ui, h o webuijsf, me refiero al xmlns que se especifica en la definición de la página: <webuijsf:form…. [↑](#footnote-ref-1)