Componentes enterprise

Domingo Gallardo <domingo.gallardo@ua.es>

Tabla de contenidos

1. Introducción a los Componentes Enterprise JavaBeans

En este módulo vamos a presentar la tecnología Enterprise JavaBeans. Se trata de una tecnología que permite definir componentes (*enterprise beans*) que implementan lógica de negocio y que son gestionados por un servidor de aplicaciones Java EE. El servidor de aplicaciones, un entorno en tiempo de ejecución, proporciona un conjunto de servicios, como gestión de transacciones y seguridad, a los beans gestionados.

En el módulo de Componentes Web ya vimos un ejemplo de *componentes gestionados* por el servidor de aplicaciones en forma de *managed beans*. En ese caso, el servidor de aplicaciones se encargaba de la creación de instancias y de la *inyección* de estas instancias en las variables anotadas. Pero el trabajo del servidor de aplicaciones se limitaba al momento de la creación, una vez creada la instancia e inyectada en la variable que la referencia termina su trabajo. E

n el caso de los componentes enterprise el servidor de aplicaciones va a tener un trabajo mucho más intenso, ya que los servicios que proporciona duran todo el ciclo de vida del objeto: creación, invocación de sus métodos y borrado. En concreto, los servicios que proporciona son:

- Acceso remoto de múltiples clientes
- Seguridad
- Transaccionalidad distribuida
- Acceso a recursos
- Concurrencia
- Llamadas asíncronas

Veremos que el servidor de aplicaciones proporcionará estas funcionalidades en cualquier invocación a métodos de componente enterprise. De forma que, por ejemplo, comprobará que el llamador del método tiene permisos de acceso al método o incorporará la llamada al método en una transacción previamente creada. El llamador al bean nunca tendrá acceso directo a su código, todas las invocaciones de sus métodos se harán a través de los servicios proporcionados por el servidor de aplicaciones.

La especificación actual de la tecnología Enterprise JavaBeans es la 3.1, incluida en la plataforma Java EE 6. La JSR de la especificación es la JSR 318[1], aprobada en diciembre de 2009.

Para más información recomendamos el libro *EJB 3 in Action*[2] de la editorial Manning.

1.1. Acceso a la capa de negocio gestionado por el servidor de aplicaciones

Las aplicaciones y frameworks que hemos venido estudiando y utilizando hasta ahora se basan en el denominado modelo de N-capas que separa claramente la presentación, la lógica de negocio y el acceso a los datos. Este diseño facilita sobremanera el mantenimiento de la aplicación, ya que la separación en capas nos permite modificar una de las partes de la aplicación sin que el resto se vea afectado por el cambio (o, al menos, viéndose afectado de forma muy localizada).

Definimos los siguientes elementos en esta arquitectura:

[1] http://jcp.org/en/jsr/detail?id=318
[2] http://www.manning.com/panda/

- API REST que recibe peticiones GET, POST, PUT y DELETE sobre recursos y que devuelve objetos JSON. La capa de API REST se encarga de transformar objetos JSON en objetos Java con los que se comunica con las funciones proporcionadas por la capa de servicio que implementa la lógica de negocio
- Capa de servicios que recibe objetos Java y realiza peticiones transaccionales a la capa de datos, definida por DAOs implementados con JPA
- Capa de datos que trabaja con JPA y con entidades gestionadas por el *entity manager* y que implementan actualizaciones y consultas realizadas sobre la base de datos

Toda la aplicación, exceptuando el servidor de base de datos, está implementada por clases Java que se ejecutan en el servidor de aplicaciones, ejecutado a su vez por una única JVM (Máquina Virtual Java). Además de la gestión de los servlets y el API REST, el servidor de aplicaciones da soporte a una serie de servicios adicionales como la gestión de la seguridad en el acceso a los servlets, definición de recursos JNDI o el pooling de conexiones JDBC.

Esta arquitectura puede responder perfectamente a un gran número de peticiones simultáneas de clientes (navegadores que consultan nuestro sistema). El servidor utiliza las características de multiproceso de la JVM para poder responder a todas las peticiones HTTP colocando cada servlet en un hilo de ejecución.

La arquitectura EJB permite añadir un elemento más a esta arquitectura: la intermediación automática de los servicios del servidor de aplicaciones en cualquier llamada a los métodos de negocio definidos en los beans. Veamos algunos de los posibles servicios.

Seguridad en el acceso a la capa de negocio

Si diseñamos aplicaciones con requisitos de seguridad estrictos (como banca, telefonía, etc.) es necesario ser muy cuidadoso en el acceso a las operaciones de negocio. Una operación que, por ejemplo, realice una transferencia bancaria de una cuenta a otra sólo podría ser autorizada si quien la solicita tiene un rol determinado. El servidor de aplicaciones puede obligar a que los accesos a los métodos de negocio sólo se puedan realizar si el llamador tiene los permisos adecuados.

Con la tecnología vista hasta ahora no es posible garantizar esto. El acceso de seguridad lo hemos visto a nivel de aplicación web (servlet), pero podría pasar que nos equivocáramos en el código del servlet o al añadir nuevas funcionalidades y termináramos permitiendo acceder desde capas externas de la aplicación sin los permisos adecuados a métodos de negocio restringidos. Es muy conveniente establecer una barrera de seguridad en el nivel de los métodos de negocio de la capa de servicios.

La definición de restricciones de seguridad en la capa de negocio permite también automatizar su comprobación en tests.

La arquitectura EJB permite dar una solución a esta necesidad, definiendo restricciones de acceso en los enterprise beans. Un método de un enterprise bean sólo puede ser llamado si se tiene un rol determinado. Y además, esta configuración de seguridad se define de forma declarativa, simplificándose mucho su mantenimiento.

Transacciones gestionadas por el contenedor

Otro elemento interesante que nos proporciona la capa de EJB y que no tenemos hasta ahora es la posibilidad de anidar varias operaciones de negocio en una única transacción. En la definición de los métodos de negocio que hemos hecho hasta ahora cada método define una única transacción atómica siguiendo el siguiente patrón:

- Creamos un *entity manager*
- Abrimos una transacción
- Realizamos operaciones sobre la base de datos a través de los DAO
- Cerramos la transacción
- Cerramos el *entity manager*

Si queremos agrupar dos o más operaciones no hay otra forma de hacerlo que creando un nuevo método de negocio que realice todas las modificaciones sobre la base de datos.

La arquitectura EJB combinada con JPA en el servidor nos va a permitir utilizar transacciones JTA y propagación del contexto de persistencia para poder englobar más de un método de negocio en la misma transacción. Esto nos permitirá crear métodos de negocio mucho más versátiles y combinables entre si.

Además, la arquitectura EJB va a permitir la realización de transacciones distribuidas que incorporen a más de una base de datos. Por ejemplo, pensemos en una agencia de viajes que debe añadir una reserva de un hotel, una reserva de un vuelo y una entrada de un espectáculo. Supongamos que estos datos se encuentran distribuidos y son gestionados por tres bases de datos diferentes. Cada base de datos tiene su sistema de gestión de transacciones. Necesitamos entonces coordinar los sistemas de transacción de las tres bases de datos para englobarlos en una transacción global que los incluya. De esta forma, podremos realizar una operación de reserva conjunta de hotel, vuelo y espectáculo. Esta operación deberá definir una transacción global que fallará si alguna de las bases de datos falla y que hará que todas las bases de datos vuelvan a su estado anterior si esto sucede.

Al hacer que los enterprise beans sean recursos transaccionales se abstrae el uso de los recursos de datos y son los propios métodos de negocio los que se convierten en transaccionales. De esta forma, cada método de negocio de un enterprise bean proporciona un servicio transaccional que puede participar en una transacción de mayor alcance. En el ejemplo anterior de la agencia de viajes, los servicios de añadir una reserva o comprar una entrada pueden ser implementados de forma transaccional como métodos de enterprise beans (cada uno situado además en su servidor de aplicaciones correspondiente y con sus restricciones de acceso). Como ya hemos dicho previamente, esto facilita el desarrollo y (sobre todo) las pruebas del sistema.

Acceso remoto a la capa de negocio

La última funcionalidad avanzada que nos va a permitir la arquitectura EJB es la posibilidad de exportar los métodos de negocio a clientes remotos, situados en otras aplicaciones desplegadas en el servidor o incluso en otros servidores.

En una organización se suele tener un gran número de aplicaciones que proporcionan servicios y que necesitan comunicarse unas con otras. Esta arquitectura se denomina SOA (Service Oriented Arquitecture). Una de las características de SOA es que las aplicaciones de la empresa se ejecutan de forma distribuida en distintos servidores (atendiendo a necesidades de acceso a ciertos recursos o de políticas de seguridad). Estas aplicaciones necesitan comunicarse entre si, accediendo de forma remota a los servicios definidos por cada una de ellas.

Ya veremos que una forma de proporcionar estos servicios remotos es definiendo servicios REST. Pero la arquitectura EJB permite también la posibilidad de convertir en remotas las

interfaces de las operaciones de negocio, de forma que desde cualquier servidor de la empresa se puede acceder a los métodos declarados en un componente EJB, utilizando una llamada similar a la de una llamada a un procedimiento remoto, pero de forma totalmente transparente.

La utilización de los componentes EJB como componentes remotos está bastante extendida, pero no lo estudiaremos en profundidad en el curso. Sirven para definir la versión más tradicional de la arquitectura SOA, utilizando elementos como los servicios web SOAP y JAX-WS, los componentes EJB de mensajes o los buses de conexiones. Frente a esta versión tradicional de las arquitecturas distribuidas, el enfoque que presentamos en el curso se base en la utilización de servicios REST ligeros que se comunican utilizando formatos de datos abiertos y flexibles como JSON.

1.2. Un ejemplo sencillo

Vamos a empezar con un sencillo ejemplo en el que comprobaremos cómo definir un componente EJB y cómo usarlo desde otro componente gestionado, un servlet.

El código que define el enterprise bean es muy sencillo: una clase java anotada con `@Stateless`. Con esa anotación estamos definiendo un bean de sesión sin estado. Comentaremos más adelante las distintas características de cada uno de los tipos de enterprise bean.

```
import javax.ejb.Stateless;

@Stateless
public class SaludoServicio {

    private String[] misSaludos = {"Hola, que tal?",
            "Cuanto tiempo sin verte", "Que te cuentas?",
            "Me alegro de volver a verte"};

    public String saludo(String nombre) {
        int random = (int) (Math.random() * misSaludos.length);
        String saludo = nombre + ", " + misSaludos[random];
        return saludo;
    }
}
```

En el bean definimos el método `saludo` que recibe un `nombre` y devuelve un saludo aleatorio precedido por ese nombre.

Un ejemplo de la llamada al bean es el siguiente código, un servlet en el que se obtiene una referencia al bean y se invoca a su método `saludo()`:

```
@WebServlet(name="holamundo", urlPatterns="/holamundo")
public class HolaMundoServlet extends HttpServlet {

    @EJB
    SaludoServicio saludoServicio; ❶

    protected void doGet(HttpServletRequest request,
                HttpServletResponse response) throws ServletException,
 IOException {
```

```
        response.setContentType("text/html");

        int errorStatus = 0;
        String errorMsg = "";
        String nombre = request.getParameter("nombre");

        if (nombre == null) {
            errorStatus = HttpServletResponse.SC_BAD_REQUEST;
            errorMsg = "Faltan parámetros en la petición";
            response.setStatus(errorStatus);
            PrintWriter out = response.getWriter();
            out.println(errorMsg);
        } else {
            PrintWriter out = response.getWriter();
            out.println("<!DOCTYPE HTML PUBLIC \"" +
                    "-//W3C//DTD HTML 4.0 " +
                    "Transitional//EN\">");
            out.println("<HTML>");
            out.println("<BODY>");
            out.println("<h1>Stateless</h1>");
            out.println("<ul>");
            out.println("<li>" + saludoServicio.saludo(nombre) + "</
li>"); ❷
            out.println("</ul>");
            out.println("</BODY>");
            out.println("</HTML");
        }
    }
}
```

❶ Obtención del bean por inyección de dependencias. El servidor de aplicaciones inyecta en la variable `saludoServicio` una referencia al bean

❷ Invocación al método `saludo` del bean

Por completitud, la página JSP desde la que se invoca al servlet es la siguiente:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
    <head>
        <title>Start Page</title>
        <meta http-equiv="Content-Type" content="text/html;
 charset=UTF-8">
    </head>
    <body>
        <h1>Hello World!</h1>
        <form action="<%=request.getContextPath()%>/holamundo">
        <p>Introduce tu nombre (sin
 estado): <input type="text" name="nombre"></p>
        <input type="submit" value="Enviar">
        </form>
    </body>
</html>
```

Al ejecutar el servlet veremos que la llamada al método `saludo()` del bean devuelve una cadena con uno de los saludos. De esta forma podemos comprobar que la inyección ha funcionado bien.

Pero hasta aquí el funcionamiento es muy parecido al de un *managed bean* La diferencia es que en este caso el trabajo del servidor de aplicaciones no termina con la inyección, como pasa en el *managed bean*. En este caso, al ser *saludoServicio* un *enterprise bean* el servidor de aplicaciones debe gestionar **todas las llamadas a sus métodos**, en nuestro caso la invocación al método `saludo()`.

Añadimos restricción de seguridad en el acceso al método

¿Cómo podríamos comprobar que la invocación al método `saludo()` está gestionada por el servidor de aplicaciones? Podríamos hacerlo de una forma muy sencilla. Hemos dicho que una de las características del servidor de aplicaciones es que permite definir restricciones de seguridad. Vamos a comprobar que añadiendo una sencilla anotación en el método podemos conseguir provocar un error cuando se intenta acceder a él sin tener los permisos necesarios.

```
@RolesAllowed({"usuario-saludo"}) ❶
public String saludo(String nombre) {
    int random = (int) (Math.random() * misSaludos.length);
    String saludo = misSaludos[random];
    return nombre + ", " + saludo;
}
```

❶ La anotación `@RolesAllowed` restringe el acceso al método

Si ejecutamos el ejemplo, comprobaremos que la llamada al método desde el servlet provoca una excepción de tipo `javax.ejb.EJBAccessException`. Realmente a quien estamos invocando cuando ejecutamos el método `saludo()` es al servidor de aplicaciones, que se encarga de comprobar las restricciones de seguridad y de invocar al bean sólo si se cumplen.

1.3. Arquitectura Enterprise JavaBeans

La arquitectura Enterprise JavaBeans hace posible la creación de clases gestionadas (denominados *componentes EJB o enterprise beans*) que viven en un contenedor (servidor de aplicaciones) y que proporcionan servicios (métodos) que pueden ser invocados de forma segura, transaccional, remota y concurrente. El contenedor es el encargado de proporcionar los servicios comentados en el apartado anterior (acceso remoto, seguridad y transaccionalidad) así como de gestionar su ciclo de vida.

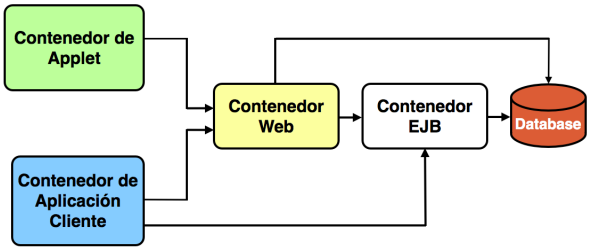
El enterprise bean vive en un contenedor

Al desplegar el *enterprise bean* en el servidor de aplicaciones se construye automáticamente un conjunto de clases que se encargarán de realizar la intercepción de las llamadas de los clientes y que son las que realmente inyecta el servidor de aplicaciones en las variables del cliente. El cliente nunca invocará directamente al código del enterprise bean, sino que todas las llamadas serán siempre a través del servidor de aplicaciones.

Cuando programamos un enterprise bean estamos proporcionando su interfaz y su implementación (cuáles son sus métodos de negocio y cómo se implementan), pero dejamos al servidor de aplicaciones la tarea engorrosa de generar las clases de soporte que proporcionan acceso remoto, seguro, concurrente y transaccional.

La intermediación del servidor de aplicaciones en las llamadas de los clientes al bean es un ejemplo más del uso de contenedores en Java EE. El uso de contenedores que gestionan objetos y les proporcionan servicios añadidos es una técnica recurrente en la arquitectura Java

EE. Los servlets también son componentes que se despliegan y viven en el contenedor web. Este contenedor da soporte a su ciclo de vida y proporciona un entorno con servicios que éstos pueden invocar. Cuando se recibe una petición HTTP que debe ser respondida por un servlet determinado es el contenedor web el que se encarga de instanciar el servlet (o recuperarlo de un *pool*) en un thread nuevo y de invocarlo para que responda a la petición.



En la figura anterior vemos los contenedores proporcionados por la arquitectura Java EE. Además de los contenedores de servlets y de enterprise beans, podemos utilizar el contenedor de aplicaciones cliente y el contenedor de applets. Las aplicaciones cliente son aplicaciones de escritorio Java que se encuentran en el servidor de aplicaciones pero que se pueden descargar y ejecutar en máquinas clientes remotas, utilizando la tecnología Java Web Start. El contenedor de aplicaciones cliente proporciona una serie de servicios, como el acceso al servicio JNDI del servidor de aplicaciones, que pueden ser utilizados desde las máquinas cliente. Los applets, por último, como es bien conocido, son aplicaciones Java que se ejecutan en los clientes web.

Cuando se está trabajando con componentes se tiene que dedicar tanta atención al despliegue (deployment) del componente como a su desarrollo. Entendemos por despliegue la incorporación del componente a nuestro contenedor EJB y a nuestro entorno de trabajo (bases de datos, arquitectura de la aplicación, etc.). El despliegue se define de forma declarativa. En la versión 2.1 de la especificación EJB, el despliegue se definía mediante un fichero XML (descriptor del despliegue, deployment descriptor) en el que se definen todas las características del enterprise bean. Desde la versión 3.0 de la especificación es posible definir la configuración de los componentes EJB mediante anotaciones en las clases Java que los implementan.

Servicios proporcionados por el contenedor EJB

En el apartado anterior hemos comentado que la diferencia fundamental entre los componentes y los objetos clásicos reside en que los componentes *viven* en un contenedor EJB que los envuelve proporcionando una capa de servicios añadidos. ¿Cuáles son estos servicios? Los más importantes son los siguientes:

Manejo de transacciones
: Apertura y cierre de transacciones asociadas a las llamadas a los métodos del enterprise bean.

Seguridad
: Comprobación de permisos de acceso a los métodos del enterprise bean.

Concurrencia
: Llamada simultánea a un mismo bean desde múltiples clientes.

Servicios de red
: Comunicación entre el cliente y el bean en máquinas distintas.

Gestión de recursos
: Gestión automática de múltiples recursos, como colas de mensajes, bases de datos o fuentes de datos en aplicaciones heredadas que no han sido traducidas a nuevos lenguajes/entornos y siguen usándose en la empresa.

Persistencia
: Sincronización entre los datos del bean y tablas de una base de datos.

Gestión de mensajes
: Manejo de Java Message Service (JMS).

Escalabilidad
: Posibilidad de constituir clusters de servidores de aplicaciones con múltiples hosts para poder dar respuesta a aumentos repentinos de carga de la aplicación con sólo añadir hosts adicionales.

Adaptación en tiempo de despliegue
: posibilidad de modificación de todas estas características en el momento del despliegue del enterprise bean.

Pensemos en lo complicado que sería programar una clase "a mano" que implementara todas estas características. Como se suele decir, la programación de EJB es sencilla si la comparamos con lo que habría que implementar de hacerlo todo por uno mismo. Evidentemente, si la aplicación que se está desarrollando no necesita estos servicios, se podría utilizar simplemente páginas JSP y JDBC.

1.4. Tipos de enterprise beans

A partir de la especificación EJB 3.0 se definen principalmente dos tipos de enterprise beans:

- Beans de sesión (con estado, sin estado y singleton)
- Beans dirigidos por mensajes dirigidos por mensajes (*message-driven beans*, MDBs)

Un **bean de sesión** representa un componente que encapsula un conjunto de métodos o acciones de negocio que pueden ser llamados de forma síncrona. Ejemplos de bean de sesión podrían ser un sistema de peticiones de reserva de libros de una biblioteca o un carrito de la compra. En general, cualquier componente que ofrezca un conjunto de servicios (métodos) a los que se necesite acceder de forma distribuida, segura y transaccional.

Los **beans dirigidos por mensajes** se diferencian de los anteriores en que la comunicación con ellos no es por medio de invocaciones, sino por el envío de mensajes que se encolan y se procesan de forma asíncrona. Un ejemplo podría ser un MDB `ListenerNuevoCliente` que se activara cada vez que se envía un mensaje comunicando que se ha dado de alta a un nuevo cliente.

En el curso vamos a ver en profundidad únicamente el primer tipo de beans.

Beans de sesión

Los beans de sesión ofrecen métodos de negocio que pueden ser utilizados de forma transaccional, remota y segura.

Existen tres tipos de beans de sesión: con estado, sin estado y singleton.

Los beans de sesión sin estado responden a peticiones de los clientes y no guardan ningún estado entre una petición y la siguiente. Los clientes del bean realizan una petición a uno de

sus métodos. El bean responde a cada petición de forma síncrona (la petición no devuelve el control hasta que el bean no termina de procesarla), el cliente obtiene la respuesta y sigue realizando su procesamiento. Cada petición es independiente. No se establece ningún tipo de sesión entre el cliente y el bean que haga necesario guardar un estado.

Los beans de sesión con estado contienen variables de instancia, campos, en los que se guarda su estado. Los clientes remotos establecen una sesión con ellos y pueden ir modificando sus datos. El estado se mantiene entre una petición y otra.

Los beans de sesión singleton se instancian una vez por aplicación y existen durante el ciclo de vida de la aplicación. Los beans de sesión singleton están pensados para circunstancias en las que un hay que compartir una única instancia de enterprise bean entre todos los clientes concurrentes.

Beans de sesión sin estado

Los beans de sesión sin estado no se modifican con las llamadas de los clientes. Los métodos que ponen a disposición de las aplicaciones clientes son llamadas que reciben datos y devuelven resultados, pero que no modifican internamente el estado del bean. Esta propiedad permite que el contenedor EJB pueda crear una reserva (*pool*) de instancias, todas ellas del mismo bean de sesión sin estado y asignar cualquier instancia a cualquier llamada de un cliente. Incluso un único bean puede estar asignado a múltiples clientes, ya que la asignación sólo dura el tiempo de invocación del método solicitado por el cliente. Y al revés, si hacemos distintas llamadas desde un cliente cada vez puede ser que nos responda una instancia distinta del bean.

Cuando un cliente invoca un método de un bean de sesión sin estado, el contenedor EJB obtiene una instancia de la reserva. Cualquier instancia servirá, ya que el bean no guarda ninguna información referida al cliente. Tan pronto como el método termina su ejecución, la instancia del bean está disponible para otros clientes. Esta propiedad hace que los beans de sesión sin estado sean muy escalables para un gran número de clientes. El contenedor EJB no tiene que mover sesiones de la memoria a un almacenamiento secundario para liberar recursos, simplemente puede obtener recursos y memoria destruyendo las instancias.

Podemos comprobar que esto es así realizando una pequeña modificación a nuestro bean de ejemplo. Podemos devolver junto con el saludo una referencia de la instancia real que ejecuta el método:

```
...
public String saludo(String nombre) {
    int random = (int) (Math.random() * misSaludos.length);
    String saludo = nombre + ", " + misSaludos[random] + " ["
 + this.toString() + "]"; ❶
    return saludo;
}
...
```

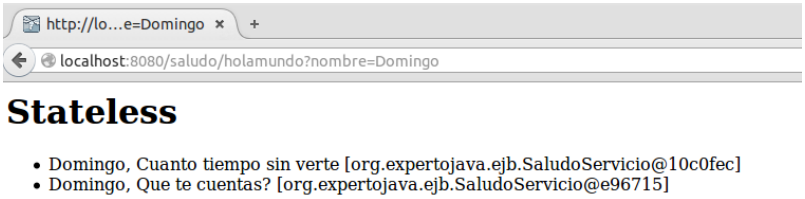
❶ Añadimos en el saludo la llamada a `this.toString()` que devuelve el identificador de la instancia del bean

Y modificamos el servlet para que realice dos llamadas al método:

```
...
```

```
out.println("<h1>Stateless</h1>");
out.println("<ul>");
out.println("<li>" + saludoServicio.saludo(nombre) + "</li>");
out.println("<li>" + saludoServicio.saludo(nombre) + "</li>");
out.println("</ul>");
...
```

Podemos comprobar en la página devuelta por el servlet que están respondiendo instancias distintas:



Los beans de sesión sin estado se usan en general para encapsular procesos de negocio. Estos beans suelen recibir nombres como `ServicioBroker` o `GestorContratos` para dejar claro que proporcionan un conjunto de procesos relacionados con un dominio específico del negocio. A veces se les suele denominar *Business Objects*, y también usar el sufijo BO para su nombre: `GestorContratosBO`.

También puede usarse un bean de sesión sin estado como un puente de acceso a una base de datos o a un conjunto de entidades JPA. En una arquitectura cliente-servidor, el bean de sesión podría proporcionar al interfaz de usuario del cliente los datos necesarios, así como modificar objetos de negocio (base de datos o entidades JPA) a petición de la interfaz. Este uso de los beans de sesión sin estado es muy frecuente y constituye el denominado patrón de diseño *session facade*.

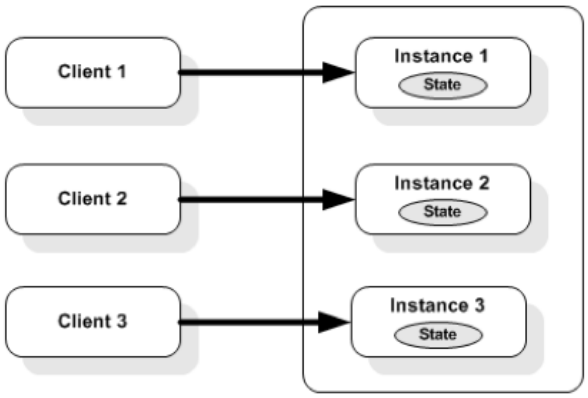
Algunos ejemplos de bean de sesión sin estado podrían ser:

- Un componente que comprueba si un símbolo de compañía está disponible en el mercado de valores y devuelve la última cotización registrada.
- Un componente que calcula la cuota del seguro de un cliente, basándose en los datos que se le pasa del cliente.

Beans de sesión con estado

En un bean de sesión con estado, el bean almacenan datos específicos obtenidos durante la conexión con el cliente en sus campos (variables de instancia). Cada bean de sesión con estado, por tanto, almacena el estado conversacional de un cliente que interactúa con el bean. Este estado conversacional se modifica conforme el cliente va realizando llamadas a los métodos de negocio del bean. El estado conversacional no se guarda cuando el cliente termina la sesión.

El hecho de que se establezca un estado entre el bean y el cliente obliga a que, una vez establecida la conexión, siempre sea el mismo bean el que atienda al cliente. El servidor de aplicaciones ya no puede hacer lo de utilizar una instancia distinta en cada llamada. Esto hace que los beans con estado sean menos escalables que los beans sin estado, ya que es necesario mantener un bean activo para cada conexión establecida.



Al igual que en los beans sin estado, la interacción del cliente con el bean se realiza mediante llamadas a sus métodos de negocio.

Los beans con estado son apropiados para aquellos casos en los que necesitamos guardar en memoria un estado que vamos a ir modificando llamada a llamada. Su uso evita tener que almacenar los estados intermedios en una base de datos.

- Un ejemplo típico es un carrito de la compra, en donde el cliente va guardando uno a uno los ítem que va comprando.
- Un enterprise bean en una biblioteca que tiene métodos para ir reservando libros y realizar un préstamo conjunto de todos los libros reservados.

Para definir un bean con estado basta con utilizar la anotación `Stateful`:

```
@Stateful
public class SaludoConEstadoServicio {
    ArrayList<String> saludos = new ArrayList<String>(); ❶

    @EJB
    SaludoServicio saludoServicio; ❷

    public String saludo(String nombre) {
        String saludo = saludoServicio.saludo(nombre); ❸
        saludos.add(saludo);
        return saludo;
    }

    public ArrayList<String> saludos() {
        return saludos;
    }
}
```

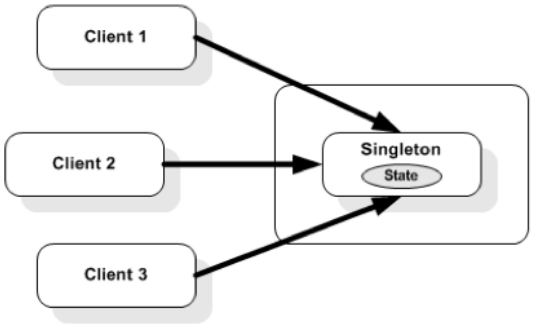
❶ Estado local definido por un `ArrayList` donde se guardan saludos
❷ Se obtiene una referencia al bean `SaludoServicio` para obtener el saludo
❸ Se obtiene un saludo invocando a `SaludoServicio` y se guarda en el array de saludos

En el ejemplo definimos el enterprise bean `SaludoConEstadoServicio` que guarda los saludos en un array local antes de devolverlo. También tiene un método `saludos()` que devuelve el array de todos los saludos emitidos.

Beans de sesión singleton

Un bean de sesión *singleton* es instanciado una vez por aplicación y proporciona un acceso fácil a estado compartido. Si el contenedor está distribuido en muchas máquinas, cada aplicación tendrá una única instancia del singleton en cada JVM.

Múltiples clientes van a acceder de forma concurrente a la misma instancia del bean:



Para definir un bean de sesión singleton se debe utilizar la anotación `@Singleton`:

```
@Startup
@Singleton
public class SaludoSingletonServicio {
    ArrayList<String> saludos = new ArrayList<String>();

    @EJB
    SaludoServicio saludoServicio;

    @Lock(LockType.WRITE)
    public String saludo(String nombre) {
        String saludo = saludoServicio.saludo(nombre);
        saludos.add(saludo);
        return saludo;
    }

    @Lock(LockType.READ)
    public ArrayList<String> saludos() {
        return saludos;
    }
}
```

La anotación `@Lock(LockType.READ)` permite accesos concurrentes a los métodos. La anotación `@Lock(LockType.WRITE)` obliga a un acceso secuencial al método.

El contenedor decide cuándo inicializar la instancia del singleton. Es posible obligar a que la inicialización se realice al arrancar la aplicación usando la anotación `@Startup`. De esta forma, el contenedor inicializa todos los singleton de arranque y ejecuta sus métodos marcados con `@PostConstruct` antes de que la aplicación esté disponible y de servir las peticiones de los clientes.

Es posible determinar el orden en el que el contenedor debe inicializar los beans singleton utilizando la anotación `@DependsOn`:

```
@Singleton
public class Foo {
 //...
}

@DependsOn("Foo")
@Singleton
public class Bar {
 //...
}
```

El contenedor se asegura de que el bean `Foo` se ha inicializado antes de inicializar el bean `Bar`.

Beans dirigidos por mensajes

Son el segundo tipo de beans propuestos por la especificación de la arquitectura EJB. Estos beans permiten que las aplicaciones J2EE reciban mensajes JMS de forma asíncrona. Así, el hilo de ejecución de un cliente no se bloquea cuando está esperando que se complete algún método de negocio de otro enterprise bean. Los mensajes pueden enviarse desde cualquier componente J2EE (una aplicación cliente, otro enterprise bean, o un componente Web) o por una aplicación o sistema JMS que no use la tecnología J2EE.

La diferencia más visible con los beans de sesión es que los clientes no acceden a los beans dirigidos por mensajes mediante interfaces (explicaremos esto con más detalle más adelante), sino que un bean dirigido por mensajes sólo tienen una clase de implementación.

En muchos aspectos, un bean dirigido por mensajes es parecido a un bean de sesión sin estado.

- Las instancias de un bean dirigido por mensajes no almacenan ningún estado conversacional ni datos de clientes.
- Todas las instancias de los beans dirigidos por mensajes son equivalentes, lo que permite al contenedor EJB asignar un mensaje a cualquier instancia. El contenedor puede almacenar estas instancias para permitir que los streams de mensajes sean procesados de forma concurrente.
- Un único bean dirigido por mensajes puede procesar mensajes de múltiples clientes.

Las variables de instancia de estos beans pueden contener algún estado referido al manejo de los mensajes de los clientes. Por ejemplo, pueden contener una conexión JMS, una conexión de base de datos o una referencia a un objeto enterprise bean.

Cuando llega un mensaje, el contenedor llama al método `onMessage` del bean. El método `onMessage` suele realizar un casting del mensaje a uno de los cinco tipos de mensajes de JMS y manejarlo de forma acorde con la lógica de negocio de la aplicación. El método onMessage puede llamar a métodos auxiliares, o puede invocar a un bean de sesión o de entidad para procesar la información del mensaje o para almacenarlo en una base de datos.

Un mensaje puede enviarse a un bean dirigido por mensajes dentro de un contexto de transacción, por lo que todas las operaciones dentro del método onMessage son parten de un única transacción.

1.5. Acceso al enterprise bean

Existen dos formas de obtener una referencia a un bean: usando inyección de dependencias o nombres JNDI. En ningún caso se puede hacer un `new` de un enterprise bean, siempre es el servidor de aplicaciones el que realiza su construcción y devuelve el objeto con el que interactúa el cliente.

Acceso por inyección de dependencias

La primera forma de obtener una referencia a un enterprise bean es usando inyección de dependencias. Debemos usar la anotación `@EJB` en una variable para indicar que el servidor debe inyectar un enterprise bean del tipo definido por el tipo de la variable:

```
@EJB
SaludoServicio saludoServicio;
```

Podemos realizar inyección de dependencias en componentes gestionados desplegados en la misma aplicación en la que se define el enterprise bean: servlets y otros enterprise beans.

El servidor inyectará en la variable un bean del tipo definido en el momento de creación del componente, inyectando la referencia en la instancia recién creada. Es importante por ello conocer el ciclo de vida del componente.

Por ejemplo, en el caso de un servlet, la creación del servlet sólo se hace una vez, en el momento de la puesta en marcha de la aplicación. Recordemos que las peticiones al servlets se implementan como hilos de ejecución concurrente y que las variables definidas en el servlet son compartidas por todos los hilos.

En un servlet podemos inyectar un enterprise bean de sesión sin estado o un bean singleton. No podemos inyectar un bean de sesión con estado, porque el estado sería compartido por todas las peticiones. Si queremos que cada petición utilice un bean de sesión con estado debemos definir la variable que referencia el bean en el método `deGet` y obtener la referencia al bean accediendo a su nombre JNDI.

Acceso por nombre JNDI

Se definen tres espacios de nombres JNDI para identificar los componentes enterprise y acceder a ellos utilizando JNDI: `java:global`, `java:module` y `java:app`.

El espacio de nombres `java:global` permite acceder a enterprise beans remotos y su sintaxis es la siguiente:

```
java:global[/nombre aplicación]/nombre módulo/nombre enterprise bean
```

El nombre de la aplicación se requiere si el enterprise bean está empaquetado en un EAR. No es nuestro caso, porque estamos empaquetando los beans en WARs.

El espacio de nombres `java:module` se utiliza para acceder a beans local dentro del mismo módulo. Por ejemplo para acceder desde un servlet a un bean desplegado en el mismo WAR. Su sintaxis es la siguiente:

```
java:module/nombre enterprise bean
```

El espacio de nombres `java:app` se utiliza para acceder a enterprise beans en otra aplicación dentro del mismo servidor. Su sintaxis es

```
java:app[/nombre módulo]/nombre enterprise bean
```

Por ejemplo, si el enterprise bean `MiBean` se empaqueta dentro del archivo de aplicación web `miApp.war`, el nombre del módulo es `miApp`. Sus nombres JNDI son `java:module/MiBean`, `java:app/miApp/MiBean` y `java:global/miApp/MiBean`.

Cuando se despliega la aplicación el servidor WildFly muestra por la salida estándar los distintos nombres JNDI de los beans desplegados. Por ejemplo, suponiendo que hayamos definido un enterprise bean `SaludoConEstadoServicio` en el WAR `saludo.war`:

```
java:global/saludo/SaludoConEstadoServicio
java:app/saludo/SaludoConEstadoServicio
java:module/SaludoConEstadoServicio
```

Desde cualquier componente gestionado podemos obtener una referencia a un enterprise bean llamando al método `lookup` de un objeto `InitialContext`. Hay que pasar el nombre JNDI del bean que queremos obtener. Hay que capturar la excepción checked que puede lanzar el método.

Por ejemplo, si queremos acceder al bean `SaludoConEstadoServicio` desde un servlet podríamos hacerlo de la siguiente forma:

```
@WebServlet(name="holamundoestado", urlPatterns="/holamundoestado")
public class HolaMundoConEstadoServlet extends HttpServlet {

    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
                             throws ServletException, IOException { }

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
                                throws ServletException, IOException {

        response.setContentType("text/html");

        int errorStatus = 0;
        String errorMsg = "";
        String nombre = request.getParameter("nombre");

        if (nombre == null) {
            errorStatus = HttpServletResponse.SC_BAD_REQUEST;
            errorMsg = "Faltan parámetros en la petición";
            response.setStatus(errorStatus);
            PrintWriter out = response.getWriter();
            out.println(errorMsg);
        } else {
            try {
                SaludoConEstadoServicio saludoServicio =
 (SaludoConEstadoServicio)
```

```
                    new InitialContext().lookup("java:module/
SaludoConEstadoServicio"); ❶

                PrintWriter out = response.getWriter();
                out.println("<!DOCTYPE HTML PUBLIC \"" +
                        "-//W3C//DTD HTML 4.0 " +
                        "Transitional//EN\">");
                out.println("<HTML>");
                out.println("<BODY>");
                out.println("<h1>Stateful</h1>");
                out.println("<p>Saludos: </p>");
                out.println("<ul>");
                out.println("<li>" + saludoServicio.saludo(nombre) + "</
li>"); ❷
                out.println("<li>" + saludoServicio.saludo("Juan") + "</
li>");
                out.println("<li>" + saludoServicio.saludo("Isabel") + "</
li>");
                out.println("<li>" + saludoServicio.saludo("Antonio")
 + "</li>");
                out.println("</ul>");
                out.println("<p> Recuperamos la lista de saludos guardada:
 </p>");
                ArrayList<String> saludos = saludoServicio.saludos(); ❸
                out.println("<ul>");
                for (String str : saludos) {
                    out.println("<li>" + str + "</li>");
                }
                out.println("<ul>");
                out.println("</BODY>");
                out.println("</HTML");

            } catch (NamingException e) {
                e.printStackTrace();
            }
        }
    }
```

❶ Obtención de la referencia al bean mediante su nombre JNDI
❷ Cambiamos el estado del bean, enviándole distintos nombres
❸ Recuperamos todos los saludos almacenados

1.6. Pruebas de enterprise beans con Arquillian

El framework de JBoss Arquillian[3] permite realizar testing de componentes desplegándolos en el servidor de aplicaciones y lanzando los tests, que se ejecutan como si se invocaran dentro del servidor.

Por ejemplo, el siguiente código prueba el método `saludo` del bean `SaludoServicio`:

```
package org.expertojava.ejb;

import static org.junit.Assert.assertTrue;
```

[3] http://arquillian.org

```
@RunWith(Arquillian.class)
public class SaludoServicioTest {

    @EJB
    private SaludoServicio saludoServicio;

    @Deployment
    public static Archive<?> deployment() {
        return ShrinkWrap.create(JavaArchive.class)
                .addClass(SaludoServicio.class);
    }

    @Test
    public void deberiaDevolverUnoDeLosSaludos() {
        String[] misSaludos = {"Hola, que tal?",
                "Cuanto tiempo sin verte", "Que te cuentas?",
                "Me alegro de volver a verte"};
        String nombre = "Pepito";
        String saludo = saludoServicio.saludo(nombre);
        assertTrue(saludo.startsWith(nombre + ", " + misSaludos[0]) ||
                   saludo.startsWith(nombre + ", " + misSaludos[1]) ||
                   saludo.startsWith(nombre + ", " + misSaludos[2]) ||
                   saludo.startsWith(nombre + ", " + misSaludos[3]));
    }
}
```

Podemos utilizar Arquillian de dos formas: con un servidor de aplicaciones *gestionado* o remoto. En el primer caso Arquillian se encarga de poner en marcha el servidor de aplicaciones, desplegar los componentes y los tests, lanzarlos y cerrar el servidor de aplicaciones. En el segundo caso debemos tener un servidor de aplicaciones en marcha y Arquillian se comunica con él para desplegar los componentes y los tests. Esta forma es más eficiente.

Asegúrate que WildFly esté funcionando antes de lanzar los tests de Arquillian. La configuración de Arquillian definida en el POM es la de acceso remoto al servidor de aplicaciones.

Se pueden comprobar las dependencias necesarias en el siguiente POM completo:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.expertojava.ejb</groupId>
    <artifactId>saludo</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>
    <name>saludo</name>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
```

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.jboss.arquillian</groupId>
            <artifactId>arquillian-bom</artifactId>
            <version>1.1.10.Final</version>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <dependency>
        <groupId>javax</groupId>
        <artifactId>javaee-api</artifactId>
        <version>7.0</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.jboss.arquillian.junit</groupId>
        <artifactId>arquillian-junit-container</artifactId>
        <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>org.wildfly</groupId>
        <artifactId>wildfly-arquillian-container-remote</artifactId>
        <version>8.1.0.Final</version>
        <scope>test</scope>
    </dependency>

</dependencies>

<build>
    <finalName>${project.name}</finalName>

    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.1</version>
            <configuration>
                <source>1.7</source>
                <target>1.7</target>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-war-plugin</artifactId>
```

```
                <version>2.3</version>
                <configuration>
                    <failOnMissingWebXml>false</failOnMissingWebXml>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.wildfly.plugins</groupId>
                <artifactId>wildfly-maven-plugin</artifactId>
                <version>1.0.2.Final</version>
                <configuration>
                    <hostname>localhost</hostname>
                    <port>9990</port>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

1.7. Ejercicios

(1,5 puntos) Construir el módulo `saludo`

En este ejercicio deberás construir el módulo `saludo` con los ejemplos vistos en la teoría de todos los tipos de enterprise beans:

- `SaludoServicio`: bean de sesión sin estado
- `SaludoRestringidoServicio`: bean de sesión sin estado con autorización
- `SaludoConEstadoServicio`: bean de sesión con estado
- `SaludoSingletonServicio`: bean de sesión singleton

Empezamos por crear el proyecto `ejercicios-ejb-expertojava`:

1. Haz un fork del proyecto vacío IntelliJ `ejercicios-ejb-expertojava` en la cuenta de `java_ua` que contiene un repositorio git inicial en el que iremos añadiendo los módulos IntelliJ que vamos a crear como ejercicios de
2. Descárgalo en tu máquina con el comando `git clone`. Tendrás un directorio `ejercicios-ejb-expertojava` que contiene un fichero `.idea` que podrás abrir con IntelliJ

```
$ git clone https://<usuario>@bitbucket.org/java_ua/ejercicios-ejb-
expertojava.git
```

Creamos dentro del proyecto el módulo IntelliJ `saludo` con un proyecto web inicial en el que vamos a añadir los distintos enterprise beans y los servlets que los usan. Podemos hacerlo creando un nuevo módulo Maven en el proyecto o a partir de la plantilla `webapp-expertojava` que tenemos en Bitbucket. A continuación se indican los pasos para hacerlo de la segunda forma:

1. Descarga en el directorio recién creado la plantilla `webapp-expertojava`, cambia su nombre a `saludo` y borra los datos de git (va a estar controlado por el proyecto git en el directorio padre):

```
$ cd ejercicios-ejb-expertojava
$ git clone https://<usuario>@bitbucket.org/java_ua/webapp-
expertojava.git
$ mv webapp-expertojava saludo
$ cd saludo
$ rm -rf .git
$ rm -f .gitignore
```

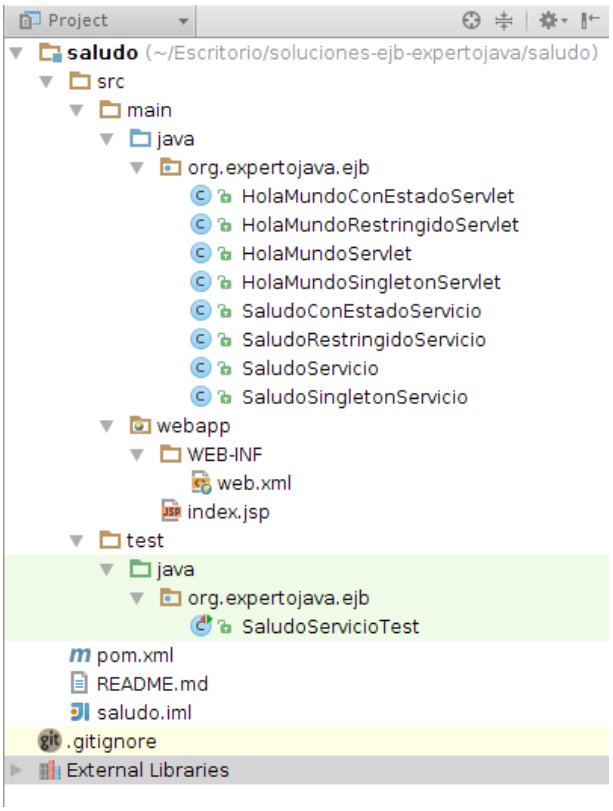
2. Edita el fichero `pom.xml` para actualizar las coordenadas del proyecto Maven, modificando los atributos como sigue:

```
<groupId>org.expertojava.ejb</groupId>
<artifactId>saludo<artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>war</packaging>

<name>saludo</name>
```

3. Por último, abre en IntelliJ el proyecto del directorio padre `ejercicios-ejb-expertojava` y procede a importar el módulo `saludo` como un módulo Maven. Para ello, escogemos la opción *Import Module* desde el panel de *Project Sctructure* o *New > Import Module > Import module from externa model > Maven*. Una vez importado el módulo podemos añadir una configuración de despliegue/ejecución y nos aseguramos de que la aplicación funciona correctamente.

4. Añade los cambios al control de versiones, abriendo el panel Changes y pulsando el botón derecho sobre *Unversioned Files* y seleccionando *Add to VCS*.

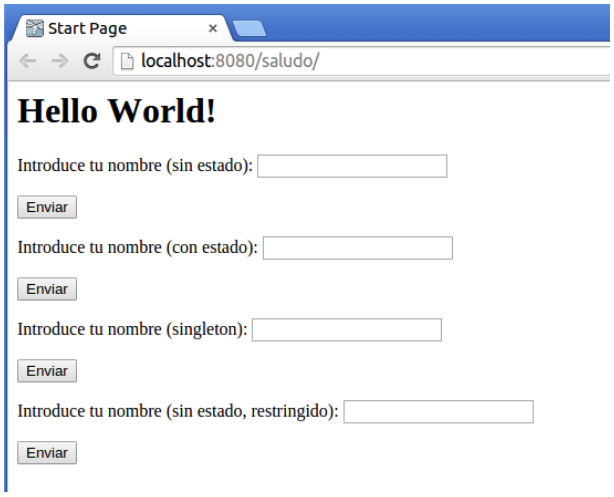
Una vez tengas el abierto en Intellij el proyecto con el primer submódulo, y tengas correctamente configurado git, debes añadir todos los distintos tipos de componentes enterprise que hemos visto en el tema, junto con servlets en los que se invocan.



El fichero `index.jsp` contiene una sencilla página HTML para invocar a los distintos servlets:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
    <head>
        <title>Start Page</title>
        <meta http-equiv="Content-Type" content="text/html;
 charset=UTF-8">
    </head>
    <body>
        <h1>Hello World!</h1>
        <form action="<%=request.getContextPath()%>/holamundo">
        <p>Introduce tu nombre (sin
 estado): <input type="text" name="nombre"></p>
        <input type="submit" value="Enviar">
        </form>
        <form action="<%=request.getContextPath()%>/holamundoestado">
            <p>Introduce tu nombre (con
 estado): <input type="text" name="nombre"></p>
            <input type="submit" value="Enviar">
        </form>
        <form action="<%=request.getContextPath()%>/holamundosingleton">
            <p>Introduce tu nombre
 (singleton): <input type="text" name="nombre"></p>
            <input type="submit" value="Enviar">
        </form>
        <form action="<%=request.getContextPath()%>/holamundorestringido">
            <p>Introduce tu nombre (sin estado,
 restringido): <input type="text" name="nombre"></p>
            <input type="submit" value="Enviar">
        </form>
```

```
    </body>
</html>
```



Por último, añade la clase de test `SaludoServicioTest` con la prueba Arquillian que hemos visto en teoría que comprueba el funcionamiento del bean de sesión sin estado. Deberás añadir las dependencias necesarias en el `pom.xml`.

(2 puntos) Construir el módulo `calculadora`

1. Crea dentro de `ejercicios-ejb-expertojava` un nuevo módulo IntelliJ llamado `calculadora` con las siguientes características
   - Módulo Maven de tipo WAR con el groupId `org.expertojava.ejb` y el artifactID `calculadora`
   - Contiene un enterprise bean de sesión sin estado `org.expertojava.ejb.Calculadora` con los siguientes métodos:
     # `public double mult(double num1, double num2)`
     # `public double div(double num1, double num2)`
2. Escribe la página `index.jsp` y un servlet para comprobar su funcionamiento.
3. Escribe un fichero de test de Arquillian con dos tests que comprueben el funcionamiento de ambos métodos.

2. Ciclo de vida de los enterprise beans y seguridad de acceso

Vamos a incluir en esta sesión dos apartados que no tiene mucha relación, pero que tienen una extensión adecuada para cubrirlos ambos en una única sesión. Son el ciclo de vida y las restricciones de seguridad en el uso de los enterprise beans.

2.1. Ciclo de vida

Hemos visto que los enterprise beans son objetos gestionados por el servidor de aplicaciones. Esto significa que nosotros no vamos a poder llamar a `new()` para sólo vamos a poder obtener referencias para su uso. Es el servidor de aplicaciones se encarga de crear las instancias de los enterprise beans, de inyectar sus referencias en las variables anotadas y de controlar sus distintas fases del ciclo de vida.

Empezamos con el ciclo de vida de los beans de sesión con estado, el más elaborado, y seguimos con el de los beans sin estado.

Ciclo de vida de los beans de sesión con estado

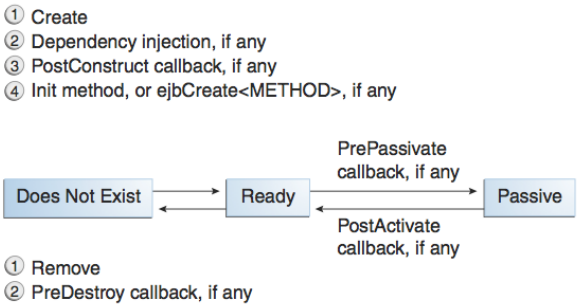
Recordemos un ejemplo de enterprise bean con estado:

```
@Stateful
public class SaludoConEstadoServicio {
    ArrayList<String> saludos = new ArrayList<String>(); ❶

    @EJB
    SaludoServicio saludoServicio; ❷

    public String saludo(String nombre) {
        String saludo = saludoServicio.saludo(nombre); ❸
        saludos.add(saludo);
        return saludo;
    }

    public ArrayList<String> saludos() {
        return saludos;
    }
}
```

❶ Estado local definido por un `ArrayList` donde se guardan saludos
❷ Se obtiene una referencia al bean `SaludoServicio` para obtener el saludo
❸ Se obtiene un saludo invocando a `SaludoServicio` y se guarda en el array de saludos

En los beans con estado el servidor debe garantizar que el cliente se comunica siempre con la misma instancia del bean, ya que el cliente va modificando su estado. Todas las invocaciones a métodos del bean de un mismo cliente deben ser respondidas por la misma instancia, que es la que guarda el estado modificado. Cada cliente se conectará con una instancia distinta.

El ciclo de vida de los beans de sesión con estado debe por tanto garantizar esta característica. Se muestra en la siguiente figura:



1. El contenedor crea una instancia usando el constructor por defecto cuando comienza una nueva sesión con un cliente.
2. Después de que el constructor se ha completado, el contenedor inyecta los recursos tales como contextos JPA, fuentes de datos y otros beans.
3. La instancia se almacena en memoria, esperando la invocación de alguno de sus métodos.
4. El cliente ejecuta un método de negocio y el contenedor lo invoca en el bean
5. Espera hasta que los siguientes métodos son invocados y los ejecuta.
6. Si el cliente permanece ocioso por un periodo de tiempo, el contenedor *pasiva* el bean, serializándolo y guardándolo en disco.
7. Si el cliente vuelve a invocar a un bean pasivado, éste se activa (el objeto es leído del disco) y se ejecuta el método
8. Si el cliente no invoca un método sobre el bean durante un cierto periodo de tiempo, el bean es destruido.
9. Si el cliente invoca algún método con el atributo `@Remove` el bean es destruido.

Es posible definir métodos de callback del ciclo de vida, a los que el contenedor llamará antes o después de cierto momento del ciclo de vida, con las siguientes anotaciones. Son muy útiles para gestionar la apertura y cierre de recursos inyectados y usados por el bean.

- `@PostConstruct`: el método con esta anotación se invoca justo después de que se ha ejecutado el constructor por defecto y de que se han inyectado los recursos.
- `@PrePassivate`: invocado antes de que un bean sea pasivado.
- `@PostActivate`: invocado después de que el bean haya sido traído a memoria por el contenedor y antes de que se ejecute cualquier método de negocio invocado por el cliente
- `@PreDestroy`: invocado después de que haya expirado el tiempo de conexión o el cliente haya invocado a un método anotado con `@Remove`. Despés de invocar al método, la instancia del bean se elimina y se pasa al recolector de basura.

Las anotaciones `@PrePassivate` y `@PostActivate` tienen que ver con la actualización de recursos que no pueden ser serializados. Por ejemplo, un objeto `java.sql.Connection` no puede ser serializado y debe ser reiniciado cuando el bean se reactiva.

Ciclo de vida de los beans de sesión sin estado

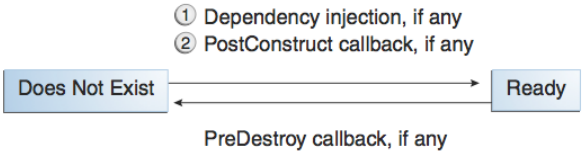
Los beans de sesión sin estado tienen un ciclo de vida muy sencillo. O existen o no existen. Una vez que el bean es creado, se coloca en un *pool* para servir las peticiones de los clientes.

En algún momento el bean se destruye, o bien cuando se reduce la carga del servidor o cuando la aplicación se cierra. El contenedor hace lo siguiente:

1. Crea una instancia del bean usando el constructor por defecto.
2. Inyecta recursos como proveedores de JPA o conexiones de bases de datos y llama a los métoods etiquetados con `@PostConstruct`.
3. Coloca instancias del bean en un *pool*.
4. Saca del pool una instancia ociosa cuando recibe una petición de ejecución de un método por el cliente.
5. Ejecuta el método de negocio solicitado por el cliente.
6. Cuando el método se termina de ejecutar, el bean se vuelve a colocar en el *pool*.
7. Cuando el contenedor lo considera necesario elimina el bean del *pool* y llama a los métodos `@PreDestroy` antes de colocarlo en el recolector de basura.

Los beans de sesión sin estado también soportan las anotaciones `@PostConstruct` y `@PreDestroy`.

La siguiente figura refleja este ciclo de vida:



2.2. Acceso concurrente a los beans singleton

Es posible gestionar el acceso concurrente de los clientes usando la *concurrencia gestionada por el contenedor* o la *concurrencia gestionada por el bean*. Se definen para ello las anotaciones `@ConcurrencyManagement(ConcurrencyManagementType.BEAN)` y `@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)` que hay que definir a nivel de clase.

Esta última es la menos frecuente y deja en manos del desarrollador la gestión de la concurrencia usando construcciones de Java como `synchronized` o `volatile`.

En la concurrencia gestionada por el contenedor, el contenedor se encarga de gestionar el acceso concurrente de los clientes a los métodos del bean. El contenedor realiza un bloque a nivel de métodos, asignádoles un bloqueo de tipo `Read` (compartido) o `Write` (exclusivo). Un bloqueo `Read` en un método permite las invocaciones concurrentes del método. Un bloqueo `Write` espera al procesamiento de una invocación antes de comenzar la siguiente. Si no especificamos nada en un método, por defecto tiene el bloque `Write`.

Estos tipos de bloqueos se especifican con las anotaciones `@Lock(LockType.READ)` y `@Lock(LockType.WRITE)` que se pueden realizar a nivel de clase o de método de negocio. El valor especificado en un método sobreescribe el de la clase.

En el caso del bloqueo exclusivo es posible definir un time-out para que se genere una excepción si el bloqueo dura más de un determinado tiempo usando la anotación `AccessTimeout`:

```
@Singleton
public class Foo {
 @Lock(LockType.WRITE)
 @AccessTimeout(value=1,unit=TimeUnit.MINUTES)
 public void TestMethod() {
    //...
 }
}
```

Es posible especificar las unidades (por defecto son MILLISECONDS):

- NANOSECONDS
- MICROSECONDS
- MILLISECONDS
- SECONDS
- MINUTES
- HOURS

2.3. Seguridad en la arquitectura EJB

Introducción a la seguridad en EJB

La seguridad es un aspecto fundamental de las aplicaciones empresariales. Cualquier aplicación interna o accesible via web va a intentar ser hackeada por extraños. Podemos analizar tres elementos fundamentales en la seguridad de una aplicación:

Autentificación
: Dicho sencillamente, la autentificación valida la identidad del usuario. La forma más común de autentificación es una simple ventana de login que pide un nombre de usuario y una contraseña. Una vez que los usuarios han pasado a través del sistema de autentificación, pueden usar el sistema libremente, hasta el nivel que les permita el control de acceso. La autentificación se puede basar también en tarjetas de identificación, certificados y en otros tipos de identificación.

Control de acceso
: El control de acceso (también conocido como autorización) aplica políticas de seguridad que regulan lo que un usuario específico puede y no puede hacer en el sistema. El control de acceso asegura que los usuarios accedan sólo a aquellos recursos y operaciones a los que se les ha dado permiso. El control de acceso puede restringir el acceso de un usuario a subsistemas, datos, y objetos de negocio. Por ejemplo, a algunos usuarios se les puede dar permiso de modificar información, mientras que otros sólo tienen permiso de visualizarla.

Comunicación segura
: Los canales de comunicación entre un cliente y un servidor son un elemento muy importante en la seguridad del sistema. Un canal de comunicación puede hacerse seguro mediante aislamiento físico (por ejemplo, via una conexión de red dedicada) o por medio de la encriptación de la comunicación entre el cliente y el servidor. El aislamiento físico es caro, limita las posibilidades del sistema y es casi imposible en Internet, por lo que lo más usual es la encriptación. Cuando la comunicación se asegura mediante la encriptación, los mensajes se codifican de forma que no puedan ser leídos ni manipulados por individuos no

autorizados. Esto se suele consigue mediante el intercambio de claves criptográficas entre el cliente y el servidor. Las claves permiten al receptor del mensaje decodificarlo y leerlo.

La seguridad debe aplicarse tanto a la vista como a la capa de negocio. El hecho de que un usuario no pueda acceder a una página no significa que un la lógica de negocio no pueda invocarse. Un hacker puede conseguir acceder a los servicios web REST y hacer peticiones no autorizadas. O un programador de la capa web puede cometer un error y llamar desde una página sin el nivel de autorización necesaria a una operación restringida de la lógica de negocio. La arquitectura EJB proporciona un modelo de seguridad flexible y elegante que permite garantizar la seguridad en el acceso a los métodos de negocio de la aplicación enterprise.

La mayoría de los servidores EJB soportan la comunicación segura a través del protocolo SSL (Secure Socket Layer) y proporcionan algún mecanismo de autentificación, pero la especificación Enterprise JavaBeans sólo especifica el control de acceso a los enterprise beans.

En este apartado veremos qué mecanismos define la especificación EJB para el control de acceso a los enterprise beans. Veremos que es posible definir un control de acceso declarativo y programativo a los métodos de los beans, de forma que sólo aquellos usuarios autorizados puedan ejecutar el código restringido. Antes de eso, revisemos brevemente cómo realizar la autentificación del usuario.

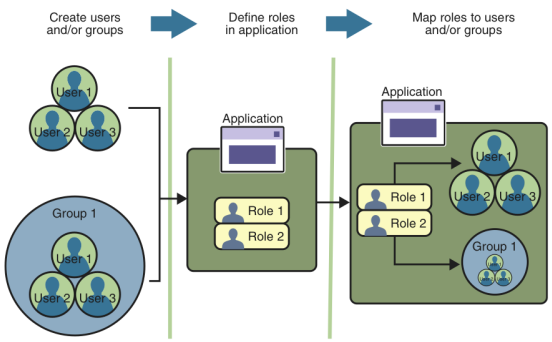
Realms, Users, Groups y Roles

Uuarios, grupos y roles son tres conceptos relacionados que forman la base de la seguridad EJB.

El usuario es cualquier persona autentificada en el sistema. Normalmente es un nombre de usuario autentifcado mediante una contraseña.

El concepto de grupo es similar al de sistema de ficheros UNIX. Un grupo se define en el servidor de aplicaciones como un conjunto de usuarios. Por ejemplo, podemos definir el grupo de "administrador" al que pertenecerán todos los usuarios que sean administradores de las aplicaciones desplegadas.

Los roles se definen en las aplicaciones y es la base de la autorización del acceso a los métodos de negocio. Sólo aquellos usuarios o grupos que tengan un determinado rol podrán acceder a determinados métodos del EJB. Se deberá mapear usuario y grupos con roles. La separación entre usuarios/grupos y roles permite que la aplicación se codifique de forma indenpendiente al entorno en el que se va a desplegar. Una aplicación puede tener un rol "Administrador" y en la compañía en la que se despliega podemos tener en el directorio LDAP un grupo "Gestor Aplicaciones" que sea equivalente. El mapeo entre grupos/usuarios y roles lo realiza el servidor de aplicaciones.



Un *realm* se define en el servidor de aplicaciones como un conjunto de usuarios y grupos definidos mediante un determinado mecanismo.

Los dos realms que tenemos por defecto en WildFly son `ManagementRealm` y `ApplicationRealm`. `ManagementRealm` se utiliza para la aplicación de administración del servidor, por lo que sólo nos permite controlar la autentificación (no se indican roles porque el único rol que tienen los usuarios de este conjunto es el de administrar el servidor). Por otro lado, `ApplicationRealm` nos permite además controlar la autorización, mediante la asignación de roles a usuarios.

Como ya vimos en el módulo de Componentes Web, es posible añadir nuevos usuarios en estos realms con el script de WidlFly `addUser.sh` situado en `$WILDFLY_HOME/bin`:

```
$ ./addUser.sh
```

Nos preguntará en primer lugar en cuál de los dos realms por defecto queremos introducir el usuario, y a continuación nos irá pidiendo los datos del nuevo usuario. El último de los datos que pide es la lista de roles permitidos separados por coma. En las aplicaciones que incorporen seguridad declarativa se nos permitirá entrar con cualquiera de los usuarios del `ApplicationRealm`. Las operaciones que nos permita hacer dependerán de los roles asignados.

JAAS: Servicio Java de Autentificación y Autorización

La seguridad en Java EE está basada en el Servicio Java de Autentificación y Autorización (JAAS). Esta arquitectura se introdujo en la versión 1.4 de Java EE y desde entonces ha sido adoptado ampliamente por la industria Java comercial y open source. La versión más reciente es la JSR 196[4] (Java Authentication Service Provider Interface for Containers), definida en Java EE 6. Las APIs más importantes son las definidas por los paquetes:

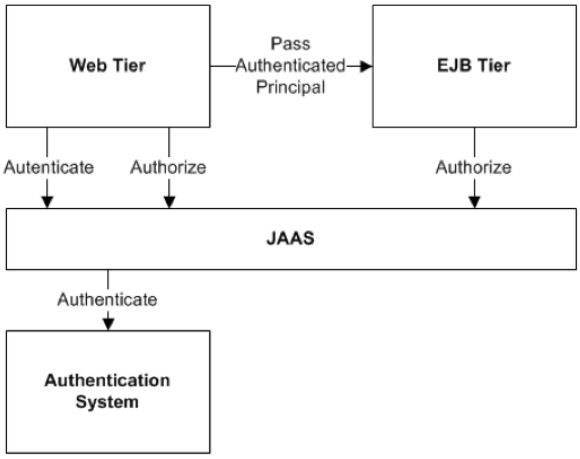
- javax.security.auth[5]
- java.security[6]
- javax.annotation.security[7]

JAAS separa el sistema de autentificación de la aplicación Jaava EE mediante el uso de un API bien definida que es implementada por el servidor de aplicaciones. La aplicación Java EE no se debe preocupar de detalles de bajo nivel como trabajar con los algoritmos de encriptación de contraseñas o comunicarse con servicios externos de autentificación como el Active Directory de Microsoft o el LDAP. El servidor de aplicaciones que contiene la aplicación Java EE se encarga de ello.

JAAS está diseñado para que la autentificación y autorización puede realizarse en cualquier capa Java EE, incluyendo la capa web y la capa EJB. En la realidad, sin embargo, la mayoría de aplicaciones Java EE son accesibles via web y comparten un único sistema de autentificación a lo largo de todas las capas. JAAS permite propagar la autentificación realizada en la capa web a cualquier otra capa de Java EE. Una vez autentificado el usuario, el contexto de autentificación se pasa a todas las capas siempre que sea posible, en lugar de repetir los

[4] https://www.jcp.org/en/jsr/detail?id=196
[5] https://docs.oracle.com/javase/7/docs/api/javax/security/auth/package-summary.html
[6] https://docs.oracle.com/javase/7/docs/api/java/security/package-summary.html
[7] https://docs.oracle.com/javaee/7/api/javax/annotation/security/package-summary.html

pasos de autentificación en cada una de ellas. El objeto Java `Principal` representa este usuario autentificado compartible entre capas. La siguiente figura representa este mecanismo:



Como se muestra en la figura, un usuario se logea en la aplicación a través de la capa web. La capa web obtiene la información de autenficiación del usuario (login y password normalmente) y los autentifica usando JAAS contra el *realm* definido en el servidor de aplicaciones. Si se valida la autentificación se obtiene un `Principal` que se asocia con uno o más roles de la aplicación. Para cada recurso restringido de la capa web o la capa EJB, el servidor de aplicaciones chequea si el principal/role esta autorizado para acceder al recurso. El `Principal` se pasa a de la capa web a la capa EJB cuando se realiza una invocación de un método de algún bean.

Autentificación en la capa web

En la asignatura *Componentes Web* ya estudiamos con detalle la autentificación en la capa web. Recordemos que hay que realizar la configuración de seguridad en el fichero `web.xml`, y que podemos escoger entre tres tipos de autentificación: FORM, BASIC y CLIENT-CERT. Con el método FORM podemos especificar una página HTML en la que se define un formulario que el usuario utilizará para logearse. En el formulario se utilizarán los nombres estándar de parámetros `j_username` y `j_password` para guardar en ellos el usuario y su contraseña y se llamará a la acción estándar `j_security_check`.

El método BASIC proporciona un mecanismo de autentificación básico, basado en cabeceras de autentificación para solicitar datos del usuario (el servidor) y para enviar los datos del usuario (el cliente). Para la comunicación se debería utilizar el protocolo SSL y la contraseña se envía codificada con el método Base64. Será el que utilicemos en los ejemplos por ser el más sencillo.

Con el método CLIENT-CERT no es necesario pedir el usuario/contraseña sino que el cliente envía al servidor de aplicaciones un certificado de clave pública almacenado en el navegador utilizando SSL y el servidor autentifica el contenido del certificado. El proveedor JAAS valida a continuación las credenciales.

En el mismo fichero `web.xml` se definen los roles y las restricciones de acceso a recursos (URLs) definidos en los servlets o páginas JSP. Por ejemplo, el siguiente código declara una restricción de acceso a la URL `/holamundorestringido` a los usuarios con el rol `User`. Como el método de autentificación es BASIC, el navegador pedirá en una ventana de diálogo el usuario y la contraseña. Si utilizamos una ventana normal del navegador éste guardará la

configuración de usuario y contraseña una vez introducida por primera vez. Para probar más de una vez podemos abrir una ventana de incógnito.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://
xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1">
    <security-constraint> ❶
        <web-resource-collection>
            <web-resource-name>holamundorestringido</web-resource-name>
            <url-pattern>/holamundorestringido</url-pattern> ❷
        </web-resource-collection>
        <auth-constraint>
            <role-name>usuario-saludo</role-name> ❸
        </auth-constraint>
    </security-constraint>
    <login-config>
        <auth-method>BASIC</auth-method> ❹
    </login-config>
    <security-role> ❺
        <role-name>usuario-saludo</role-name>
    </security-role>
</web-app>
```

❶ Declaración de restricción de seguridad
❷ URL del recurso que se restringe
❸ Rol/grupo del usuario autorizado
❹ Tipo de autentificación `BASIC`
❺ Declaración de todos los roles que se utilizan en las restricciones (en el caso en que haya más de un recurso autorizado a otros roles, aquí se deben listar todos

La autentificación `BASIC` hace que el navegador pregunta al usuario por un login y una contraseña. El navegador envía esa identidad al servidor de aplicaciones, que la valida contra la lista de usuarios. Si el usuario y la contraseña coinciden con la registrada en el servidor, la petición adquiere la identidad y sus roles. Esta identidad se propaga en las llamadas que se realizan desde el servlet a otros componentes.

En este caso, al acceder a la URL `/holamundorestringido` se lanza la autenficación y se comprueba que el usuario tiene el rol `usuario-saludo` para permitirle el acceso al servlet.

Control de acceso basado en roles

En la arquitectura Enterprise JavaBeans, la identidad de seguridad se representa por un objeto `java.security.Principal`. Este objeto actúa como representante de usuarios, grupos, organizaciones, tarjetas inteligentes, etc. frente a la arquitectura de control de acceso de los enterprise beans.

Los descriptores del despliegue y las anotaciones en Java EE 5.0 incluyen elementos que declaran a qué roles lógicos se permite el acceso a los métodos de los enterprise beans. Estos roles se consideran lógicos porque no reflejan directamente usuarios, grupos o cualquier otra identidad de seguridad en un entorno operacional específico. Los roles se hacen corresponder con grupos de usuarios o usuarios del mundo real cuando el bean se despliega. Esto permite

que el bean sea portable ya que cada vez que el bean se despliega en un nuevo sistema, los roles se asignan a usuarios y grupos específicos de ese entorno operacional.

Definición de roles con anotaciones

Para declarar los roles con permiso de acceso a los métodos de un bean primero se utiliza la anotación `@DeclareRoles` en la que se declaran todos los roles que accederán a distintos métodos del bean. Después podemos especificar para cada uno de los métodos qué roles están autorizados con la anotación `@RolesAllowed`. Si ponemos esta anotación a nivel de clase, se aplica a todos los métodos de la clase.

Por ejemplo, en el siguiente código se declaran los roles `Admin`, `Bibliotecario` y `Socio` y se definen los permisos de acceso a los métodos.

```
@Stateless
    @DeclareRoles({"Admin","Bibliotecario","Socio"})
    public class OperacionBOBean implements OperacionBOLocal {
    // ...

    @RolesAllowed("Admin")
    public String borraOperacion(String idOperacion) {
    // ...
    }

    @RolesAllowed({"Admin","Bibliotecario"})
    public String realizaReserva(String idUsuario, String idLibro) {
    // ...
    }

    @RolesAllowed({"Admin","Bibliotecario"})
    public String realizaPrestamo(String login, String isbn) {   }

    // ...

    @PermitAll
    public List<OperacionTO> listadoTodosLibros() {   }

}
```

La anotación `@PermitAll` declara que todos los roles tienen acceso al método.

La identidad de seguridad runAs

Mientras que las anotaciones `@RolesAllowed` y los elementos `method-permission` especifican qué roles tienen acceso a qué métodos del bean, la anotación `@RunAs` y el elemento `security-identity` especifica bajo qué rol se ejecuta el bean, usando el elemento `runAs`. En otras palabras, el objeto rol que se define en `runAs` se usa como la identidad del enterprise bean cuando éste intenta invocar métodos en otros beans. Esta identidad no tiene por qué ser la misma que la identidad que accede al bean por primera vez.

Por ejemplo, la siguiente anotación declara que todos los métodos del bean `EmployeeService` siempre se van a ejecutar con la identidad `admin`.

```
@RunAs("admin")
```

```
@Stateless public class EmployeeServiceBean implements EmployeeService {
...
}
```

Esta clase de configuración es útil cuando el enterprise bean o los recursos accedidos en el cuerpo del método requieren un rol distinto del que ha sido usado para obtener acceso al método. Por ejemplo, el método `create()` podría llamar a un método en el enterprise bean X que requiera la identidad de seguridad de `Administrador`.

Podemos entender la secuencia de cambio de identidades de la siguiente forma:

1. El cliente invoca un método del bean con una identidad `Id1`.
2. El bean comprueba si la identidad `Id1` tiene permiso para ejecutar el método. La tiene.
3. El bean consulta el elemento `security-identity` y cambia la identidad a la que indica ese elemento. Supongamos que es la identidad `Id2`.
4. El bean realiza las llamadas dentro del método con la identidad `Id2`.

Es obligado resaltar que hay que usar con precaución esta funcionalidad, ya que con ella podemos atribuir cualquier rol a cualquier usuario.

Una limitación de la funcionalidad es que es obligado definir un único rol para todos los métodos.

Gestión de seguridad programativa en el enterprise bean

En el código del enterprise bean es posible comprobar qué usuario o grupo ha realizado la llamada al bean y si tiene un determinado rol asociado. Para ello se usan los siguientes métodos del `SessionContext`:

- `Principal getCallerPrincipal()`: devuelve el objeto `Principal` asociado al usuario o grupo que ha llamado al método.
- `Boolean isCallerInRole(String rol)`: devuelve `true` o `false` dependiendo de si el usuario o grupo que ha llamado al método pertenece al rol que se le pasa como parámetro.

El objeto `SessionContext` se obtiene declarando el método `setSessionContext(SessionContext ctx)` del ciclo de vida del bean y guardando el contexto en una variable de instancia del bean. El contenedor EJB llamará a este método en el momento de creación del enterprise bean.

Un ejemplo de código en el que se utilizan estos métodos:

```
@Stateless
public class MiBean implements SessionBean {

   @Resource
   SessionContext ctx;

   ...

   public void miMetodo() {
      System.out.println(ctx.getCallerPrincipal().getName());
```

```
        if (ctx.isCallerInRole("administrador")) {
            //código ejecutado por administradores
            System.out.println("Me llama un administrador");
        }
        if (ctx.isCallerInRole("bibliotecario")){
            //código ejecutado por bibliotecarios
            System.out.println("Me llama un bibliotecario");
        }
    }
    ...
```

2.4. Ejercicios

(0,75 puntos) Ciclo de vidad de bean con estado

1. Comprueba el funcionamiento de un bean con estado creando una versión con estado del bean Calculadora llamada `CalculadoraConEstado`. Escribe un servlet que compruebe su uso accesible desde la página web principal y un test Arquillian.
2. Comprueba el funcionamiento del ciclo de vida del bean con estado añadiendo métodos con las anotaciones del ciclo de vida que escriban algún mensaje por la salida estándar. Comprueba que el funcionamiento del ciclo de vida es el indicado en los apuntes. Escribe la explicación en un fichero `respuestas.txt` en la raíz del repositorio.

(0,75 puntos) Acceso seguro a un método

En esta sesión debes probar el registro de usuarios para que puedan acceder al método seguro `SaludaRestringido` del enterprise bean `SaludoRestringidoServicio`.

1. Usando el comando `add-user.sh` añade en el servidor un usuario con el rol (grupo) `usuario-servicio`.
2. Configura correctamente la seguridad en el `web.xml` y en el bean para que el acceso al servlet `HolaMundoRestringidoServlet` y al bean esté limitado sólo a un usuario con este rol.
3. Escribe otro bean llamado `SaludoRestringidoProgServicio` en el que pruebes la seguridad programativa.
4. Comprueba el correcto funcionamiento y escribe los resultados en el fichero `respuestas.txt`.

3. Enterprise beans y JPA

Veremos en este apartado cómo se gestionan las transacciones y el contexto de persistencia (entity manager) con beans de sesión.

3.1. Proyecto básico JPA

Vamos a retomar el ejemplo básico con el que terminamos la sesión de JPA, en el que definimos una aplicación web con las entidades `Autor` y `Mensaje` y una relación uno-a-muchos entre ellas. Nos vamos a olvidar por el momento de los DAOs y vamos a trabajar directamente con las entidades.

Por simplificar, vamos a colocar todas las clases de entidad en el paquete `org.expertojava.ejb`:

Autor.java

```
package org.expertojava.ejb;

import javax.persistence.*;
import java.util.HashSet;
import java.util.Set;

@Entity
public class Autor {
    @Id
    @GeneratedValue
    @Column(name = "autor_id")
    Long id;
    @Column(name="email", nullable = false)
    private String correo;
    private String nombre;
    @OneToMany(mappedBy = "autor", cascade = CascadeType.ALL)
    private Set<Mensaje> mensajes = new HashSet<Mensaje>();

    public Long getId() { return id; }

    public String getCorreo() { return correo; }
    public void setCorreo(String correo) { this.correo = correo; }

    public String getNombre() { return nombre; }
    public void setNombre(String nombre) { this.nombre = nombre; }

    public Set<Mensaje> getMensajes() { return mensajes; }
    public void setMensajes(Set<Mensaje> mensajes) { this.mensajes =
 mensajes; }

    public Autor() {
    }

    public Autor(String nombre, String correo) {
        this.nombre = nombre;
        this.correo = correo;
    }
```

```
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Autor autor = (Autor) o;

        if (!correo.equals(autor.correo)) return false;
        if (!nombre.equals(autor.nombre)) return false;

        return true;
    }

    @Override
    public int hashCode() {
        int result = correo.hashCode();
        result = 31 * result + nombre.hashCode();
        return result;
    }

    @Override
    public String toString() {
        return "Autor{" +
                "id=" + id +
                ", correo='" + correo + '\'' +
                ", nombre='" + nombre + '\'' +
                '}';
    }
}
```

Mensaje.java

```
package org.expertojava.ejb;

import javax.persistence.*;
import java.util.Date;

@Entity
public class Mensaje {

    @Id
    @GeneratedValue
    @Column(name = "mensaje_id")
    private Long id;
    @Column(nullable = false)
    private String texto;
    private Date fecha;
    @ManyToOne
    @JoinColumn(name = "autor", nullable = false)
    private Autor autor;

    public Long getId() { return id; }

    public String getTexto() { return texto; }
    public void setTexto(String texto) { this.texto = texto; }

    public Date getFecha() { return fecha; }
```

```
    public void setFecha(Date fecha) { this.fecha = fecha; }

    public Autor getAutor() { return autor; }
    public void setAutor(Autor autor) { this.autor = autor; }

    public Mensaje() {}

    public Mensaje(String texto, Autor autor) {
        this.texto = texto;
        this.autor = autor;
     }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Mensaje mensaje = (Mensaje) o;

        if (!autor.equals(mensaje.autor)) return false;
        if (!texto.equals(mensaje.texto)) return false;

        return true;
    }

    @Override
    public int hashCode() {
        int result = texto.hashCode();
        result = 31 * result + autor.hashCode();
        return result;
    }

    @Override
    public String toString() {
        return "Mensaje{" +
                "id=" + id +
                ", texto='" + texto + '\'' +
                ", fecha=" + fecha +
                '}';
    }
}
```

El fichero `persistence.xml` es el siguiente. suponemos que en el servidor de aplicaciones ya está creada la fuente de datos `java:/datasources/MensajesDS`.

META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
             xmlns="http://java.sun.com/xml/ns/
persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="
        http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
    <persistence-unit name="mensajes">
        <jta-data-source>java:/datasources/MensajesDS</jta-data-source>
        <properties>
```

```
            <property name="hibernate.hbm2ddl.auto" value="create" />
            <property name="hibernate.show_sql" value="true" />
        </properties>
    </persistence-unit>
</persistence>
```

3.2. Gestión de transacciones con beans de sesión

Los beans de sesión permiten dos tipos de gestión de transacciones, denominados CMP (*Container Managed Persistence*) y BMP (*Bean Managed Persistence*).

Cuando se utiliza CMP, los beans gestionan de forma automática las transacciones. Toda llamada a un método del bean es interceptada por el contenedor, que crea un contexto transaccional que dura hasta que el método ha terminado (con éxito o con fracaso, al lanzar una excepción). Si el método termina con éxito el contenedor hace un commit de la transacción. Si el método lanza una excepción el contenedor hace un rollback.

Cuando se utiliza BMP es el programador el que debe abrir y cerrar las transacciones utilizando JTA de forma explícita el código de los métodos del bean.

Ambos enfoques se basan en JTA, que es utilizado por el servidor de aplicaciones para gestionar las transacciones distribuidas utilizando el algoritmo *two phase commit*. Los recursos que participan en estas transacciones deben ser fuentes de datos XA.

Un uso muy común de los beans de sesión es la implementación de la capa de negocio de la aplicación, definiendo métodos de servicio transaccionales. En cada método de servicio se realizan llamadas a la capa de persistencia definida por los DAO que gestionan las entidades del modelo. Los DAO utilizan entity managers inyectados por el contenedor. El contenedor se encarga automáticamente de inyectar el mismo entity manager en todos los DAO que participan en la misma transacción, de forma que todos los DAO van a compartir el mismo contexto de persistencia. Se libera a los DAO de la gestión de transacciones y su implementación se hace más sencilla y flexible.

Al utilizar JTA es posible llamar desde un método de servicio a otros servicios de la capa de negocio y englobar todo ello en una única transacción.

Gestión de transacciones BMT

En la gestión de transacciones denominada BMT (*Bean Managed Transactions*) el programador se hace cargo de la gestión de transacciones dentro de los métodos de los beans utilizando JTA. Ya hemos visto anteriormente un ejemplo cuando describimos JTA:

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN) ❶
public class PedidoService {
   @Resource
   private UserTransaction userTransaction; ❷

   public void hacerPedido(Item item, Cliente cliente) {
      try {
         userTransaction.begin(); ❸
         if (itemService.disponible(item)) {
            pedidoService.add(cliente,item);
            clienteService.anotaCargo(cliente,item);
```

```
            itemService.empaqueta(cliente,item);
         }
         userTransaction.commit();
      } catch (Exception e) { ❹
         userTransaction.rollback();
         e.printStackTrace();
      }
   }
}
```

❶ Se declara el tipo de gestión de transacción como BMT, para lo que se define el valor `TransactionManagementType.BEAN` como tipo de la anotación `@TransactionManagement`

❷ Se declara el recurso `UserTransaction` que vamos a utilizar para controlar la transacción mediante JTA. Se define utilizando inyección de dependencias.

❸ Se inicia la transacción. Las siguientes líneas de código definen las llamadas a los métodos transaccionales de otros beans de sesión. En estos métodos se utilizan conexiones XA que participan en la transacción definida en el bean.

❹ Si alguna de estas llamadas genera una excepción, ésta se recoge y se llama a `userTransaction.rollback()`, con lo que automáticamente se deshace la transacción y todos los recursos implicados en la transacción vuelven a su estado original.

En el caso en que todas las llamadas terminen correctamente, se realiza la llamada `userTransaction.commit();` que la transacción y final se propaga a todos los recursos que intervienen en la transacción.

Vemos que el funcionamiento es idéntico al que hemos definido cuando hemos hablado de JTA. Todas las llamadas a los beans de sesión se incorporan a la misma transacción, *sin modificar el código de los métodos*. De esta forma no perdemos flexibilidad en los métodos de negocio, siguen siendo atómicos, se pueden utilizar de forma independiente y, además, pueden incorporarse distintas llamadas a la misma transacción.

Un problema de BMT es que no se puede unir una transacción definida en un método a una transacción ya abierta por el llamador. De esta forma, si el método anterior es llamado desde un cliente que tiene una transacción JTA abierta, esta transacción del llamador se suspende. JTA no permite transacciones anidadas. De la misma forma, los métodos dentro de la transacción definida en el bean (`pedidoBean.add()` y demás) no pueden a su vez abrir una nueva transacción con BMT, ya que estaríamos en el mismo caso que el que hemos mencionado, abriendo una transacción anidad. En el siguiente apartado veremos que la gestión de transacciones por el contenedor (CMT, *Containter Managed Transaction*) proporciona más flexibilidad, ya que, por ejemplo, permite que los métodos de los beans se incorporen a transacciones ya abiertas.

Gestión de transacciones CMT

Cuando verdaderamente aprovechamos todas las funcionalidades del contenedor EJB es cuando utilizamos el otro enfoque de la gestión de las transacciones en los enterprise beans, el llamado CMT (*Container Managed Transactions*). Aquí, el programador no tiene que escribir explícitamente ninguna instrucción para definir el alcance de la transacción, sino que es el contenedor EJB el que la maneja automáticamente. En este enfoque, los métodos de negocio de los beans se anotan y podemos configurar con anotaciones cómo se van a comportar. Veamos cómo definiríamos con CMT el mismo ejemplo anterior.

```
@Stateless
```

```
@TransactionManagement(TransactionManagementType.CONTAINER)
public class PedidoService {

   @TransactionAttribute(TransactionAttributeType.REQUIRED)
   public void hacerPedido(Item item, Cliente cliente) {
      if (itemService.disponible(item)) {
         pedidoService.add(cliente,item);
         clienteService.anotaCargo(cliente,item);
         itemService.empaqueta(cliente,item);
      }
   }
}
```

Definimos en la clase el tipo de gestión de transacciones como `TransactionManagementType.CONTAINER`. El contenedor se encarga de la gestión de las transacciones del bean. Nosotros sólo debemos indicar en cada método qué tipo de gestión de transacción se debe hacer. En el método `hacerPedido` lo hemos definido como `REQUIRED` usando la anotación `@TransactionAttribute`.

La anotación `@TransactionAttribute` se puede realizar a nivel de método o a nivel de clase. En este último caso todos los métodos de la clase se comportan de la forma especificada.

El tipo de atributo de transacción REQUIRED obliga a que el método se ejecute dentro de una transacción. En el caso en que el llamador del método haya abierto ya una transacción, el contenedor ejecuta este método en el ámbito de esa transacción. En el caso en que el llamador no haya definido ninguna transacción, el contenedor inicia automáticamente una transacción nueva.

El tipo de atributo de transacción REQUIRED es el que tienen por defecto todos los métodos de los componentes EJB. Por eso se dice que son componentes transaccionales.

Todas las llamadas a otros métodos desde el bean se ejecutan en una transacción. Estas llamadas pueden ser a la capa de persistencia o a otros métodos de negocio.

Si el método no termina normalmente, sino que lanza una excepción de tipo `RuntimeException` (porque la lanza él mismo o alguna de las llamadas realizadas) se realiza automáticamente un rollback de la transacción. El contenedor captura la excepción provocada por el método y llama a JTA para realizar un rollback.

Si se lanza una excepción chequeada que no es de tipo `RuntimeException` el rollback no se realiza automáticamente. En este caso hay que llamar al método `setRollbackOnly` del `EJBContext` que se puede obtener por inyección de dependencias:

```
@Stateless
public class PeliculaService {
   @Resource
   private EJBContext context;

   public void metodoTransaccional() throws AlgunaExcepcionChequeada {

      // Llamadas a otros métodos

      if (seHaProducidoError) {
         context.setRollbackOnly();
```

```
            throw new AlgunaExcepcionChequeada("Se ha producido un error");
        }
    }
}
```

Si el método del bean termina normalmente, el contenedor hace commit de la transacción.

El código resultante es muy sencillo ya toda la gestión de la transacción se realiza de forma implícita. En el código sólo aparece la lógica de negocio.

La utilización de las transacciones CMT tiene la ventaja añadida de que los métodos pueden participar en transacciones abiertas en otros métodos.

Propagación de transacciones

Existen seis posibles formas de propagar una transacción, definidas por los posibles tipos de atributos de transacción:

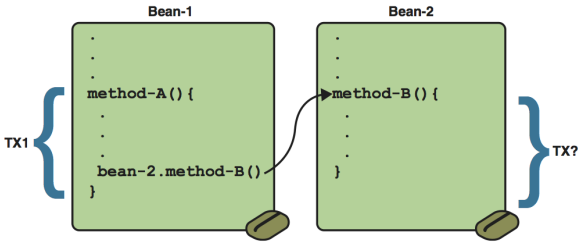
- `TransactionAttributeType.REQUIRED`
- `TransactionAttributeType.REQUIRES_NEW`
- `TransactionAttributeType.SUPPPORTS`
- `TransactionAttributeType.MANDATORY`
- `TransactionAttributeType.NOT_SUPPORTED`
- `TransactionAttributeType.NEVER`

El funcionamiento de cada tipo de gestión depende de si el llamador del método hace la llamada con una transacción ya abierta o no. Vamos a ver un ejemplo. Supongamos que los métodos a los que se llama en `hacerPedido()` no son métodos de un DAO (un POJO), sino que se tratan de métodos de negocio de otros EJBs:

```
@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
public class PedidoEJB implements PedidoLocal {
   @Resource
   private SessionContext context;

   @TransactionAttribute(TransactionAttributeType.REQUIRED)
   public void hacerPedido(Item item, Cliente cliente) {
      try {
         if (itemEJB.disponible(item)) {
            pedidoEJB.add(cliente,item);
            clienteEJB.anotaCargo(cliente,item);
            itemEJB.empaqueta(cliente,item);
         } catch (Exception e) {
            context.setRollbackOnly();
         }
      }
   }
}
```

En este caso tenemos una situación como la que se muestra en la siguiente figura, en la que un método de negocio de un EJB llama a otro método de negocio. ¿Cómo

van a gestionar la transacción los métodos llamados? Va a depender de qué tipo de `@TransactionAttribute` se hayan definido en los métodos.



`REQUIRED`
: se trata del tipo de gestión de transacciones por defecto. Si no declaramos el tipo en el método (ni en el bean) éste es el que se aplica. El código del método siempre debe estar en una transacción. Si el llamador realiza la llamada al método dentro de una transacción, el método participa en ella y se une a la transacción. En el caso de que el llamador no haya definido ninguna transacción, se crea una nueva que finaliza al terminar el método.

  Si la transacción falla en el método y éste participa en una transacción definida en el cliente, el contenedor deshará la transacción en el método y devolverá al cliente la excepción `javax.transaction.RollbackException`, para que el cliente obre en consecuencia.

  Si el método termina sin que se genere ninguna excepción y la transacción es nueva (el llamador no había definido ninguna), el contenedor hace un commit de la transacción. En el caso en que el método participe en una transacción abierta en el cliente, será éste el que deberá hacer el commit.

`REQUIRES_NEW`
: indica que el contenedor debe crear siempre una transacción nueva al comienzo del método, independientemente de si el cliente ha llamado al método con una transacción creada o no. La transacción en el llamador (si existe) queda suspendida hasta que la llamada termina. Además, el método llamado no participa en la transacción del llamador. El método llamado tiene su propia transacción y realiza un commit cuando termina. Si la transacción en el cliente falla después, el rollback ya no afecta al método terminado.

  Un ejemplo de uso de este atributo es una llamada a un método para escribir logs. Si hay un error en la escritura del log, no queremos que se propague al llamador. Por otro lado, si el llamador falla queremos que quede constancia de ello.

`SUPPORTS`
: el método hereda el estado transaccional del llamador. Si el llamador define una transacción, el método participa en ella. Si no, el método no se ejecuta en ninguna transacción. Suele utilizarse cuando el bean realiza operaciones de lectura que no modifican el estado de sus recursos transaccionales.

`MANDATORY`
: el cliente debe obligatoriamente crear una transacción antes de llamar a este método. Si se llama al método desde un entorno en el que no se ha abierto una transacción, se genera una excepción. Se usa muy raramente.

`NOT_SUPPORTED`
: el método se ejecuta sin crear una transacción. En el caso en que el llamador haya creado una, ésta se suspende, se ejecuta el método y después continua. Para el cliente el comportamiento es igual que `REQUIRES_NEW`. La diferencia es que en el bean no se ha creado ninguna transacción.

Se suele usar sólo por MDB soportando un proveedor JMS no transaccional

`NEVER`
: se genera una excepción si el cliente invoca el método habiendo creado una transacción. No se usa casi nunca.

Gestión de la transacción con la interfaz UserTransaction

Para gestionar una transacción JTA de forma programativa deberemos en primer lugar obtener del servidor de aplicaciones el objeto `javax.transacion.UserTransaction`. La forma más sencilla de hacerlo es mediante inyección de dependencias, declarándolo con la anotación:

```
@Resource
UserTransaction utx;
```

Una vez obtenido el `UserTransaction` podemos llamar a los métodos de la interfaz para demarcar la transacción:

- `public void begin()`
- `public void commit()`
- `public void rollback()`
- `public int getStatus()`
- `public void setRollbackOnly()`
- `public void setTransactionTimeout(int segundos)`

Para comenzar una transacción la aplicación se debe llamar a `begin()`. Para finalizarla, la transacción debe llamar o bien a `commit()` o bien a `rollback()`.

El método `setRollbackOnly()` modifica la transacción actual de forma que el único resultado posible de la misma sea de roll back (falllo).

El método `setTransactionTimeout(int segundos)` permite modificar el timeout asociado con la transacción actual. El parámetro determina la duración del timeout en segundos. Si se le pasa como valor cero, el timeout de la transacción es el definido por defecto.

El siguiente ejemplo simplificado muestra un fragmento de código que utiliza transacciones JTA. En el ejemplo mostramos un fragmento de código en el que un usuario de una biblioteca devuelve tarde un préstamo y se le anota una multa

```
UserTransaction tx;

//...
// Obtenemos el UserTransaction
// y lo guardamos en tx
//...

tx.begin();
try {
   operacionService.devolverEjemplar(ejemplar);
   usuarioService.multar(login, fechaDevolucionPrevista,
                         fechaDevolucionReal);
   tx.commit();
```

```
    }
} catch (Exception e) {
    tx.rollback();
    throw new BusinessException("Error al hacer la devolución", e);
}
```

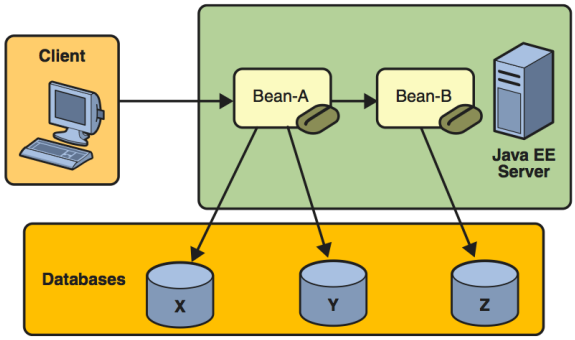
Los objetos `operacionService` y `usuarioService` son beans de sesión sin estado y los métodos `devolverEjemplar` y `multar` son métodos atómicos. Cada uno de ellos crea su propia transacción. JTA nos permite incorporar ambas llamadas en una única transacción, de forma que si uno de los métodos devuelve algún error se deshace toda la operación.

El código anterior puede ejecutarse en cualquier componente gestionado por el contenedor, como por ejemplo un servlet. También puede incluirse en un bean que define un servicio componiendo llamadas a otros servicios.

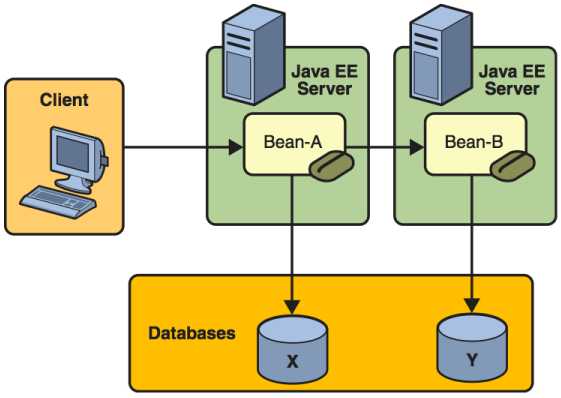
La otra forma de gestionar transacciones es gestionándolas de forma declarativa con anotaciones en los EJBs. Lo veremos más adelante.

Transacciones a través de múltiples bases de datos

Una de las características interesantes de JTA es que permite gestionar transacciones distribuidas que afectan a distintas bases de datos. Podemos utilizar distintas unidades de persistencia en los beans y agruparlas en una misma transacción. Las siguientes figuran muestran dos posibles escenarios:



En la figura anterior, el cliente invoca un método de negocio en el Bean-A. El método de negocio inicia una transacción, actualiza la base de datos X, actualiza la base de datos Y e invoca un método de negocio en el Bean-B. El segundo método de negocio actualiza la base de datos Z y devuelve el control al método de negocio en el Bean-A, que finaliza con éxito la transacción. Las tres actualizaciones de la base de datos ocurren en la misma transacción.



En la figura anterior, el cliente llama a un método de negocio en el Bean-A, que comienza una transacción y actualiza la base de datos X. Luego el Bean-A invoca un segundo método de negocio en el Bean-B, que reside en un servidor de aplicaciones remoto. El método en el Bean-B actualiza la base de datos Y. La gestión de transacciones de ambos servidores de aplicaciones se asegura de que ambas actualizaciones se realizan en la misma transacción. Es un ejemplo de una transacción distribuida a nivel de servidores de aplicaciones.

Para este segundo ejemplo necesitamos un servidor de aplicaciones compatible con la especificación completa de Java EE 6. No basta con un servidor compatible con el perfil Web.

3.3. Entity manager y contexto de persistencia en los beans

Todas las acciones efectivas sobre las base de datos se realizan a través del *entity manager*. Es el objeto que se encarga de realizar las consultas y actualizaciones en las tablas y de mantener sincronizados con la base de datos los objetos entidad que residen en su contexto de persistencia.

Recordemos que el contexto de persistencia de un *entity manager* está formado por todos los objetos entidad gestionadas por el *entity manager* y que representan una caché de primer nivel de los datos.

El *entity manager* se encarga de propagar a la base de datos los cambios realizados en los objetos entidad haciendo un *flush* de forma automática cuando la transacción termina o antes de ejecutar una query.

En JPA gestionado por el contenedor la forma normal de obtener el *entity manager* es usando la inyección de dependencias:

```
@PersistenceContext(unitName = "mensajes-ejbPU")
EntityManager em;
```

El atributo `unitName` indica la unidad de persistencia con la que se conecta el *entity manager*.

Esta inyección de dependencias hay que hacerla en un objeto gestionado al que el contenedor tenga acceso. Lo habitual es hacerla en el bean de sesión que define la interfaz de negocio.

Vamos a empezar definiendo un método de negocio muy sencillo. El método `nuevoAutor` que añade un autor a la base de datos.

AutorService.java

```
package org.expertojava.ejb;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import java.util.Date;
import java.util.List;
import java.util.Set;

@Stateless
public class AutorServicio {

    @PersistenceContext(unitName = "mensajes")
```

```
    EntityManager em;

    public Autor nuevoAutor(String correo, String nombre) {
        Autor autor = new Autor(nombre, correo);
        em.persist(autor);
        return autor;
    }
}
```

El contenedor es el encargado de inyectar el *entity manager*. El contenedor decidirá si debe crear un entity manager nuevo o si debe utilizar uno ya existente dependiendo del atributo `type` de la anotación `@PersistenceContext`.

El valor por defecto del atributo `type` es `PersistenceContextType.TRANSACTION`:

```
@PersistenceContext(type=PersistenceContextType.TRANSACTION)
EntityManager em;
```

En los entity managers de tipo `TRANSACTION` se inyecta el mismo *entity manager* en todos los DAO que comparten la transacción. Esto significa que los distintos DAOs estarán compartiendo también el mismo contexto de persistencia y aprovechando la caché de entidades gestionadas.

El otro tipo de gestión de la vida del entity manager se utiliza con los beans de sesión con estado. El contexto de persistencia se mantiene abierto mientras que existe el bean con estado y las entidades permanecen gestionadas a lo largo de múltiples transacciones. Es lo que se denomina contexto de persistencia extendido:

```
@PersistenceContext(type=PersistenceContextType.EXTENDED)
EntityManager em;
```

3.4. Ejemplo completo

Los beans de sesión sin estado son, por sus características de transaccionalidad y seguridad, los componentes idóneos para implementar la capa de negocio de una aplicación.

Podemos comprobar su funcionamiento en el siguiente ejemplo, en el que completamos la aplicación web `mensajes` con la que hemos comenzado esta sesión.

El bean sin estado `AutorServicio` define los métodos de negocio. Obtiene un entity manager por inyección de dependencias.

AutorServicio.java

```
package org.expertojava.ejb;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import java.util.Date;
import java.util.List;
import java.util.Set;
```

```
@Stateless
public class AutorServicio {

    @PersistenceContext(unitName = "mensajes")
    EntityManager em;

    private static final String all_autores = "SELECT a FROM Autor a";

    public Autor nuevoAutor(String correo, String nombre) {
        Autor autor = new Autor(nombre, correo);
        em.persist(autor);
        return autor;
    }

    public Mensaje nuevoMensaje(String texto, Long idAutor) {
        Autor autor = em.find(Autor.class, idAutor);
        if (autor == null) {
           throw new RuntimeException("No existe autor");
        } else {
            Mensaje mens = new Mensaje(texto, autor);
            mens.setFecha(new Date());
            mens.setAutor(autor);
            em.persist(mens);
            return mens;
        }
    }

    public Set<Mensaje> listaMensajes(Long idAutor) {
        Autor autor = em.find(Autor.class, idAutor);
        if (autor == null) {
            throw new RuntimeException("No existe autor");
        } else {
            Set<Mensaje> listaMensajes = autor.getMensajes();
            listaMensajes.size();
            return listaMensajes;
        }
    }

    public List<Autor> listaAutores() {
        return em.createQuery(all_autores).getResultList();
    }
}
```

Los siguientes servlets muestran cómo definir la capa de *controlador* de la aplicación que recibe peticiones del usuario y las convierte en llamadas a los métodos de negocio. Más adelante veremos que es posible sustituir esta capa por los controladores de un API RESTful.

Los servlets son:

- `NuevoMensajeServlet`: añade un nuevo mensaje a un autor
- `NuevoAutorMensajeServlet`: añade un autor y un mensaje en una única transacción. Es un ejemplo que muestra cómo es posible utilizar la propagación de transacciones en los métodos de negocio.
- `ListaMensajesServlet`: devuelve los mensajes de un autor

- `ListaAutoresServlet`: devuelve todos los autores

NuevoMensajeServlet.java

```
@WebServlet(name = "nuevomensaje", urlPatterns = "/nuevomensaje")
public class NuevoMensajesServlet extends HttpServlet {

    @EJB
    AutorServicio autorServicio;

    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
            throws ServletException, IOException {

    }

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html");

        String texto = request.getParameter("texto");
        Integer id = Integer.parseInt(request.getParameter("idAutor"));

        PrintWriter out = response.getWriter();
        Mensaje mens = autorServicio.nuevoMensaje(texto, id.longValue());
        out.println("<!DOCTYPE HTML PUBLIC \"" +
                "-//W3C//DTD HTML 4.0 " +
                "Transitional//EN\">");
        out.println("<HTML>");
        out.println("<BODY>");
        out.println("<h1>Mensaje</h1>");
        out.println("<p>" + mens.toString() + "</p>");
        out.println("</BODY>");
        out.println("</HTML");
    }
}
```

NuevoAutorMensaje.java

```
@WebServlet(name = "nuevoautor", urlPatterns = "/nuevoautor")
public class NuevoAutorMensajeServlet extends HttpServlet {

    @EJB
    AutorServicio autorServicio;
    @Resource
    UserTransaction tx;

    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
            throws ServletException, IOException {

    }

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
            throws ServletException, IOException {
```

```
        response.setContentType("text/html");

        String nombre = request.getParameter("nombre");
        String correo = request.getParameter("correo");
        String texto = request.getParameter("texto");

        PrintWriter out = response.getWriter();
        try {
            tx.begin();
            Autor autor = autorServicio.nuevoAutor(correo, nombre);
            autorServicio.nuevoMensaje(texto, autor.getId());
            out.println("<!DOCTYPE HTML PUBLIC \"" +
                    "-//W3C//DTD HTML 4.0 " +
                    "Transitional//EN\">");
            out.println("<HTML>");
            out.println("<BODY>");
            out.println("<h1>Añadido autor y mensaje</h1>");
            out.println("<p>" + autor.toString() + "<p>");
            out.println("</BODY>");
            out.println("</HTML");
            tx.commit();
        } catch (Exception e) {
            try {
                tx.rollback();
            } catch (SystemException e1) {
                e1.printStackTrace();
                throw new RuntimeException(e1);
            }
            throw new RuntimeException(e);
        }
    }
}
```

ListaMensajesServlet.java

```
@WebServlet(name = "listamensajes", urlPatterns = "/listamensajes")
public class ListaMensajesServlet extends HttpServlet {

    @EJB
    AutorServicio autorServicio;

    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
            throws ServletException, IOException {

    }

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html");

        Integer id = Integer.parseInt(request.getParameter("idAutor"));

        PrintWriter out = response.getWriter();
        Set<Mensaje> mensajes =
```

```
                autorServicio.listaMensajes(id.longValue());
        out.println("<!DOCTYPE HTML PUBLIC \"" +
                "-//W3C//DTD HTML 4.0 " +
                "Transitional//EN\">");
        out.println("<HTML>");
        out.println("<BODY>");
        out.println("<h1>Mensajes</h1>");
        out.println("<ul>");
        for (Mensaje m : mensajes) {
            out.println("<li>" + m.toString() + "</li>");
        }
        out.println("</ul>");
        out.println("</BODY>");
        out.println("</HTML");
    }
}
```

ListaAutoresServlet.java

```
@WebServlet(name="listaautores", urlPatterns="/listaautores")
public class ListaAutoresServlet extends HttpServlet {

    @EJB
    AutorServicio autorServicio;

    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
            throws ServletException, IOException {

    }

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html");

        PrintWriter out = response.getWriter();
        List<Autor> autores = autorServicio.listaAutores();
        out.println("<!DOCTYPE HTML PUBLIC \"" +
                "-//W3C//DTD HTML 4.0 " +
                "Transitional//EN\">");
        out.println("<HTML>");
        out.println("<BODY>");
        out.println("<h1>Mensajes</h1>");
        out.println("<ul>");
        for (Autor a : autores) {
            out.println("<li>" + a.toString() + "</li>");
        }
        out.println("</ul>");
        out.println("</BODY>");
        out.println("</HTML");
    }
}
```

Por último, el siguiente JSP es el que recoge los datos mediante formularios y realiza las invocaciones a los servlets correspondientes.

index.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <title>Start Page</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>

<h1>¡Mensajes!</h1>

<h3>Nuevo autor y mensaje</h3>

<form action="<%=request.getContextPath()%>/nuevoautor">
    <p>Correo: <input type="text" name="correo"></p>
    <p>Nombre: <input type="text" name="nombre"></p>
    <p>Texto: <input type="text" name="texto"></p>
    <input type="submit" value="Enviar">
</form>

<h3>Nuevo mensaje</h3>

<form action="<%=request.getContextPath()%>/nuevomensaje">
    <p>Texto: <input type="text" name="texto"></p>
    <p>Autor id: <input type="text" name="idAutor"></p>
    <input type="submit" value="Enviar">
</form>

<h3>Lista mensajes</h3>

<form action="<%=request.getContextPath()%>/listamensajes">
    <p>Autor id: <input type="text" name="idAutor"></p>
    <input type="submit" value="Enviar">
</form>

<h3>Lista autores</h3>


<form action="<%=request.getContextPath()%>/listaautores">
    <input type="submit" value="Enviar">
</form>


</body>
</html>
```

Un ejemplo de test Arquillian de la operación de negocio que crea un nuevo autor:

SaludoServicioTest.java

```
@RunWith(Arquillian.class)
public class SaludoServicioTest {

    @EJB
    private AutorServicio autorServicio;
```

```
    @Deployment
    public static Archive<?> deployment() {
        return ShrinkWrap.create(WebArchive.class)
                .addPackage(Autor.class.getPackage())
                .addAsResource("META-INF/persistence.xml");
    }

    @Test
    public void deberiaDevolverNuevoAutor() {
        String correo = "pedro.picapiedra@gmail.com";
        String nombre = "Pedro Picapiedra";
        Autor autor = autorServicio.nuevoAutor(correo, nombre);
        assertTrue(autor.getCorreo().equals(correo) &&
                   autor.getNombre().equals(nombre));
    }
}
```

3.5. Ejercicios

(3,5 puntos) Aplicación web filmoteca

Crea la aplicación web `filmoteca` con todas las entidades y DAOs que hayas definido en los ejercicios de JPA. Convierte las clases `PeliculaServicio` y `ActorServicio` en beans de sesión y crea un mínimo de 4 servlets que prueben su funcionamiento. Define también un test Arquilian como mínimo.

4. Beans asíncronos, temporizadores e interceptores

4.1. Acceso asíncrono a los beans

Por defecto, todas las llamadas a los beans son síncronas. El cliente realiza la petición y se queda en espera hasta que el bean le manda la respuesta. La especificación 3.1 de EJB introduce la novedad de permitir llamadas *asíncronas* a los beans de sesión, en las que el cliente lanza la petición al bean y continua haciendo otros trabajos hasta que el bean tiene disponible la respuesta.

Para marcar un método o un bean como asíncrono se utiliza la anotación `@Asynchronous`. El método tiene que ser `void` o devolver un objeto del tipo `Future<V>`.

Si el método asíncrono no devuelve nada, se está utilizando un patrón denominado *fire and forget*, en el que el cliente lanza la llamada al bean y continua su ejecución sin necesitar los resultados.

La interfaz `Future`, parametrizada con el tipo de dato que se devolverá cuando el método asíncrono devuelva un resultado, es una interfaz del API de concurrencia de Java. Define los siguientes métodos:

`boolean cancel(boolean mayInterruptIfRunning)`
: Intenta cancelar la ejecución de la tarea.

`V get()`
: Espera si es necesario a que la computación se complete y luego devuelve el resultado.

`V get(long timeout, TimeUnit unit)`
: Igual que el método anterior, pero con un timeout.

`boolean isCancelled()`
: Devuelve `true` si la tarea ha sido cancelada antes de ser completada normalmente.

` boolean isDone()`
: Devuelve `true` si la tarea ha terminado.

El cliente que llama a un método asíncrono y espera un resultado debe realizar un *poolig*, preguntando al método `isDone` si se ha terminado la computación. Cuando se devuelva `true` el resultado estará disponible en el propio objeto `Future` y se podrá obtener inmediatamente con el método `get`.

Por su parte, el cliente deberá devolver un objeto en el método creando un nuevo objeto de tipo `AsyncResult`:

```
@Stateless
@Asynchronous
public class MyAsyncBean {
 public Future<Integer> addNumbers(int n1, int n2) {
  Integer result;
  result = n1 + n2;
  // simulamos una consulta muy larga ...
  return new AsyncResult(result);
 }
}
```

La clase `AsyncResult` se introduce en la especificación EJB 3.1 para envolver el resultado del método en un objeto `Future`.

Como cualquier otro bean enterprise, el bean puede inyectarse en cualquier componente Java EE:

```
@EJB MyAsyncBean asyncBean;
Future<Integer> future = asyncBean.addNumbers(10, 20);
```

Después se usan los métodos del API `Future` para consultar si está disponible el resultado con `isDone` o se cancela su ejecución con el método `cancel`.

El contexto de transacción del cliente no se propaga al método de negocio asíncrono. El método se comporta como si tuviera la semántica del `REQUIRES_NEW`.

El rol de seguridad del objeto principal se propaga al método de negocio asíncrono. En este aspecto el método asíncrono se comporta de la misma forma que el síncrono.

4.2. Temporizadores

Las aplicaciones que implementan flujos de trabajo de negocio tienen que tratar frecuentemente con notificaciones periódicas. El servicio temporizador del contenedor EJB nos permite planificar notificaciones para todos los tipos de enterprise beans excepto para los beans de sesión con estado. Podemos planificar una notificación para una hora específica, para después de un lapso de tiempo o a intervalos temporales. Por ejemplo, podríamos planificar los temporizadores para que lancen una notificación a las 10:30 AM del 23 de Mayo, en 30 días o cada 12 horas.

Los temporizadores van descontando el tiempo definido. Cuando llegan a cero (saltan), el contenedor llama al método anotado con la anotación `@Timeout` en la clase de implementación del bean. El método `@Timeout` contiene la lógica de negocio que maneja el evento temporizado.

El API está definida por la interfaz TimerService[8].

El método Timeout

Los métodos anotados con `@Timeout` en la clase de implementación del bean deben devolver `void` y tomar un objeto `javax.ejb.Timer` como su único parámetro. No pueden lanzar excepciones capturadas por la aplicación.

```
@Timeout
public void timeout(Timer timer) {
    System.out.println("TimerBean: timeout occurred");
}
```

Creando temporizadores

Para crear un temporizador, el bean debe invocar uno de los métodos `createTimer()` de la interfaz `TimerService`. Cuando el bean invoca el método `createTimer()`, el servicio temporizador comienza a contar hacia atrás la duración del timer.

[8] https://docs.oracle.com/javaee/7/api/javax/ejb/TimerService.html

Como ejemplo veamos el siguiente bean de sesión:

```
@Stateless
public class TimerSessionBean implements TimerSessionLocal {
    @Resource
    TimerService timerService;

    public void setTimer(long intervalDuration) {
        Timer timer = timerService.createTimer(
                    intervalDuration,
                    "Creado un nuevo timer");
    }

    @Timeout
    public void timeout(Timer timer) {
        System.out.println("Se ha lanzado el timeout");
    }
}
```

Vemos que el bean define un método `setTimer()` en el que se usa el objeto *timerService* obtenido en línea 3 con la anotación `@Resource`. En este método definimos un temporizador inicializándolo a los milisegundos determinados por `intervalDuration`.

El temporizador puede ser un temporizador de parada única, en el que podemos indicar el momento de la parada o el tiempo que debe transcurrir antes de la misma, o un temporizador periódico, que se lanza a intervalos regulares. Básicamente son posibles cuatro tipos de temporizadores:

- Parada única con tiempo: `createTimer(long timeoutDuration, Serializable info)`
- Parada única con fecha: `createTimer(Date firstDate, Serializable info)`
- Periódico con intervalos regulares y tiempo inicial: `createTimer(long timeoutDuration, long timeoutInterval, Serializable info)`
- Periódico con intervalos regulares y fecha inicial: `createTimer(Date firstDate, long timeoutInterval, Serializable info)`

Veamos un ejemplo de utilización de temporizadores periódico con fecha inicial. Supongamos que queremos un temporizador que expire el 1 de Mayo de 2008 a las 12h. y que, a partir de esa fecha, expire cada tres días:

```
Calendar unoDeMayo = Calendar.getInstance();
unoDeMayo.set(2008, Calendar.MAY, 1, 12, 0);
long tresDiasEnMilisecs = 1000 * 60 * 60 * 24 * 3;
timerService.createTimer(unoDeMayo.getTime(),
                         tresDiasEnMilisecs,
                         "Mi temporizador");
```

Los parámetros `Date` y `long` representan el tiempo del temporizador con una resolución de milisegundos. Sin embargo, debido a que el servicio de temporización no está orientado hacia aplicaciones de tiempo real, el callback al método `@Timeout` podría no ocurrir con precisión de milisegundos. El servicio de temporización es para aplicaciones de negocios, que miden el tiempo normalmente en horas, días o incluso duraciones más largas.

Los temporizadores son persistentes. Si el servidor se apaga (o incluso se cae), los temporizadores se graban y se activarán de nuevo en el momento en que el servidor vuelve a poner en marcha. Si un temporizador expira en el momento en que el servidor está caído, el contenedor llamará al método `@Timeout` cuando el servidor se vuelva a comenzar.

Cancelando y grabando temporizadores

Los temporizadores pueden cancelarse con los siguientes eventos:

- Cuando un temporizador de tiempo expira, el contenedor EJB llama al método `@Timeout` y cancela el temporizador.
- Cuando el bean invoca el método `cancel` de la interfaz `Timer`, el contenedor cancela el timer.

Para grabar un objeto `Timer` para rerferencias futuras, podemos invocar su método `getHandle()` y almacenar el objeto `TimerHandle` en una base de datos (un objeto `TimerHandle` es serializable). Para reinstanciar el objeto `Timer`, podemos recuperar el `TimerHandle` de la base de datos e invocar en él el método `getTimer()`. Un objeto `TimerHandle` no puede pasarse como argumento de un método definido en un interfaz remoto. Esto es, los clientes remotos no pueden acceder al `TimerHandle` del bean. Los clientes locales, sin embargo, no sufren esta restricción.

Obteniendo información del temporizador

La interfaz `Timer` define también métodos para obtener información sobre los temporizadores:

```
public long getTimeRemaining();
   public java.util.Date getNextTimeout();
   public java.io.Serializable getInfo();
```

El método `getInfo()` devuelve el objeto que se pasó como último parámetro de la invocación `createTimer`. Por ejemplo, en el código anterior esta información es la cadena “Creado un nuevo timer”.

También podemos recuperar todos los temporizadores activos de un bean con el método `getTimers()`, que devuelve una colección de objetos `Timer`.

Temporizadores y transacciones

Los enterprise bean normalmente crean temporizadores dentro de transacciones. Si la transacción es anulada, también se anula automáticamente la creación del temporizador. De forma similar, si un bean cancela un temporizador dentro de una transacción que termina siendo anulada, la cancelación del temporizador también se anula.

En los beans que usan transacciones gestionadas por el contenedor, el método `@Timeout` tiene normalmente el atributo de transacción `REQUIRES` o `REQUIRES_NEW` para preservar la integridad de la transacción. Con estos atributos, el contenedor EJB comienza una nueva transacción antes de llamar al método `@Timeout`. De esta forma, si la transacción se anula, el contenedor volverá a llamar al método `@Timeout` de nuevo con los valores iniciales del temporizador.

Ventajas y limitaciones

Entre las ventajas de los temporizadores frente a la utilización de algún otro tipo de planficadores podemos destacar:

- Los temporizadores son parte del estándar Java EE con lo que la aplicación será portable y no dependerá de APIs propietarias del servidor de aplicaciones.
- La utilización del servicio de temporización de EJB viene incluido en Java EE y no tiene ningún coste adicional. No hay que realizar ninguna configuración de un planificador externo y el desarrollador no tiene que preocuparse de buscar uno.
- El temporizador es un servicio gestionado por el contenedor, y no se requiere un thread separado como en un planificador externo.
- Las transacciones se soportan completamente.
- Por defecto, los temporizadores son objetos persistentes que sobreviven a caídas del contenedor.

Limitaciones

- La mayoría de planificadores proporcionan la posibilidad de utilizar clases Java estándar; sin embargo los temporizadores requieren el uso de enterprise beans.
- Los temporizadores EJB adolecen del soporte de temporizadores al estilo cron, fechas de bloqueo, etc. disponibles en muchos planificadores de tareas.

4.3. Interceptores

Los *interceptores* (*interceptors*) son objetos que son capaces de interponerse en las llamadas a los métodos en los eventos de ciclo de vida de los beans de sesión y de mensaje. Nos permiten encapsular conductas comunes a distintas partes de la aplicación que normalmente no tienen que ver con la lógica de negocio. Los interceptores son una característica avanzada de la especificación EJB que nos permite modularizar la aplicación o incluso extender el funcionamiento del contenedor EJB. En esta sesión veremos una introducción a la definición y al funcionamiento de los interceptores para interponerse en las llamadas a los métodos de negocio.

Toda su API está definida en el package javax.interceptor[9].

La clase Interceptor

Comencemos con un ejemplo. Supongamos que queremos analizar el tiempo que tarda en ejecutarse un determinado método de negocio de un bean.

```
public void addMensajeAutor(String nombre, String texto) {
   long startTime = System.currentTimeMillis();
   Autor autor = findAutor(nombre);
   if (autor == null) {
      autor = new Autor();
      autor.setNombre(nombre);
      em.persist(autor);
   }
```

[9] https://docs.oracle.com/javaee/7/api/javax/interceptor/package-summary.html

```
        mensajeService.addMensaje(texto, nombre);
        long endTime = System.currentTimeMillis() - startTime;
        System.out.println("addMensajeAutor() ha tardado: " +
                endTime + " (ms)");*
    }
```

Aunque el método compilará y se ejecutará correctamente, el enfoque tiene serios problemas de diseño:

- Se ha añadido en el método `addMensajeAutor()` código que no tiene nada que ver con la lógica de negocio de la aplicación. El código se ha hecho más complicado de leer y de mantener.
- El código de análisis no se puede activar y desactivar a conveniencia. Hay que comentarlo y volver a recompilar la aplicación.
- El código de análisis es una plantilla que podría reutilizarse en muchos métodos de la aplicación. Pero escrito de esta forma habría que escribirlo en todos los métodos en los que queramos aplicarlo.

Los interceptores proporcionan un mecanismo para encapsular este tipo de código de una forma sencilla y aplicarlo a nuestros métodos sin interferir directamente. Los interceptores proporcionan una estructura para este tipo de conducta de forma que puedan ser ampliados y extendidos fácilmente en una clase. Por último, proporcionan un mecanismo simple y configurable para aplicar la conducta en el lugar que deseemos.

Este tipo de código que *envuelve* los métodos de la aplicación se suele denominar también *aspecto* y es la base de una técnica de programación denominada AOP (*Aspect Oriented Programming*). Un *framework* muy popular de programación dirigida por aspectos en Java es AspectJ.

Vamos ya a detallar cómo implementar los interceptores. Es muy sencillo. Basta con crear una clase Java con un método precedido por la anotación `@javax.interceptor.AroundInvoke` y con la siguiente signatura:

```
@AroundInvoke
Object <nombre-metodo>(javax.interceptor.InvocationContext invocation)
   throws Exception;
```

El método `@AroundInvoke` en una clase de intercepción envuelve la llamada al método de negocio y es invocado en la misma pila de llamada, en la misma transacción y en el mismo contexto de seguridad que el método que se está interceptando. El parámetro `javax.interceptor.InvocationContext` es una representación genérica del método de negocio que el cliente está invocando. A través de él podemos obtener información como el bean al que se está llamando, los parámetros que se están pasando en forma de un array de objetos, y una referencia al objeto `java.lang.reflect.Method` que contiene la representación del método invocado. `InvocationContext` también se usa para dirigir realmente la invocación. Veamos cómo utilizar este enfoque para hacer el análisis del tiempo anterior:

```
import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;
```

```
public class Profiler {

   @AroundInvoke ❶
   public Object profile(InvocationContext invocation)
           throws Exception {
      long startTime = System.currentTimeMillis();
      try {
         return invocation.proceed(); ❷
      } finally {
         long endTime = System.currentTimeMillis() - startTime; ❸
         System.out.println("El método " + invocation.getMethod() +
                 " ha tardado " + endTime + " (ms)");
      }
   }
}
```

❶ Marcamos el método `profile` como `@AroundInvoke`
❷ Llamada al método que estamos interceptando
❸ Cálculo final del tiempo de ejecución

El método anotado con `@AroundInvoke` en nuestra clase interceptora es el método `profile`. Tiene un aspecto muy parecido al código que escribimos en el método `addMensajeAutor`, con la excepción de que no incluimos la lógica de negocio. Se invoca a este método con `InvocationContext.proceed()`. Si se debe llamar a otro interceptor el método `proceed()` realiza esa llamada. Si no hay ningún otro interceptor en espera, el contenedor EJB llama entonces al método del bean que está siendo invocado por el cliente.

En el `finally` se calcula el tiempo de ejecución y se imprime en la salida estándar. El método `InvocationContext.getMethod()` proporciona acceso al objeto `java.lang.reflect.Method` que representa el método real que se está invocando. Se usa en la línea 14 para imprimir el nombre del método al que se está llamando.

La estructura del código permite capturar las posibles excepciones que pudiera generar la invocación del método del bean. Al poner el cálculo del tiempo final en la parte `finally` nos aseguramos de que siempre se ejecuta.

Además del método `getMethod()`, la interface `InvocationContext` tiene otros métodos interesantes. En el siguiente enlace se puede consultar su API en Java EE 7[10].

```
package javax.interceptor;

public interface InvocationContext {
   public Object getTarget();
   public Method getMethod();
   public Object[] getParameters();
   public void setParameters(Object[] newArgs);
   public java.util.Map<String, Object> getContextData();
   public Object proceed() throws Exception;
}
```

El método `getTarget()` devuelve una referencia a la instancia del bean objetivo. Podríamos cambiar nuestro método `profile` para que imprimiera los parámetros del método al que se

[10] https://docs.oracle.com/javaee/7/api/javax/interceptor/InvocationContext.html

está invocando utilizando el método `getParameters()`. El método `setParameters()` permite modificar los parámetros de la invocación al método del bean, de forma que podemos hacer que el bean ejecute el código con los parámetros proporcionados por el interceptador. El método `getContextData()` devuelve un objeto `Map` que está activo durante toda la invocación al método. Los interceptores pueden utilizar este objeto para pasarse datos de contexto entre ellos durante la misma invocación.

Una vez que hemos escrito la clase de intercepción, es hora de aplicarlo a la llamada al bean. Podemos hacerlo utilizando anotaciones o utilizando descriptores de despliegue XML. Veremos ambas opciones en los siguientes apartados.

Aplicando interceptores con anotaciones

La anotación `@javax.interceptor.Interceptors` se puede usar para aplicar interceptores a un método particular de un bean o todos los métodos de un bean. Para aplicar el método anterior de profiling basta con aplicar la anotación en su definición:

```
   @Interceptors(Profiler.class)
    public void addMensajeAutor(String nombre, String texto) {
       // ...
    }
```

La anotación `@Interceptors` también puede aplicarse a todos los métodos de negocio de un bean realizando la anotación en la clase:

```
@Stateless
@Interceptors(Profiler.class)
public class AutorServiceBean implements AutorServiceLocal {

   @PersistenceContext(unitName = "ejb-jpa-ejbPU")
   EntityManager em;
   @EJB
   MensajeServiceDetachedLocal mensajeService;

   public Autor findAutor(String nombre) {
      return em.find(Autor.class, nombre);
   }
   // ...
```

Aplicando interceptores con XML

Aunque la anotación `@Interceptors` nos permite aplicar fácilmente los interceptores, tiene el inconveniente de que nos obliga a modificar y recompilar el código cada vez que queremos añadirlos o deshabilitarlos. Es más interesante definir los interceptores mediante XML, en el fichero descriptor de despliegue `ejb-jar.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns = "http://java.sun.com/xml/ns/javaee"
         version = "3.0"
         xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation = "http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">
```

```
<assembly-descriptor>
   <interceptor-binding>
      <ejb-name>AutorServiceBean</ejb-name>
      <interceptor-class>es.ua.jtech.ejb.interceptor.Profiler</
interceptor-class>
      <method>
         <method-name>addMensajeAutor</method-name>
      </method>
   </interceptor-binding>
</assembly-descriptor>
</ejb-jar>
```

Una de las ventajas de utilizar el descriptor de despliegue XML es que es posible aplicar el interceptor a todos los métodos de todos los beans desplegados en el módulo:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns = "http://java.sun.com/xml/ns/javaee"
         version = "3.0"
         xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation = "http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">
<assembly-descriptor>
   <interceptor-binding>
      <ejb-name>*</ejb-name>
      <interceptor-class>es.ua.jtech.ejb.interceptor.Profiler</
interceptor-class>
   </interceptor-binding>
</assembly-descriptor>
</ejb-jar>
```

4.4. Ejercicios

(0,75 puntos) Bean temporizador

En el módulo `saludo` añade un ejemplo de bean temporizador, junto con un servlet que lo lance. Actualiza en `index.jsp` para poder invocar el servlet que lanza el bean.

(0,75 puntos) Bean interceptor

Define también en el módulo `saludo` un bean interceptor que escriba en la salida estándar todas las peticiones de saludo realizadas y todos los saludos devueltos.