

TEMA

Tema 6. Componentes da acceso a datos

Desarrollo de aplicaciones
multiplataforma

Acceso a datos

Autora: Silvia Macho



Tema 6: Componentes de acceso a datos

¿Qué aprenderás?

- Instalar y configurar una aplicación Java web.
- Crear una aplicación Java web formada por JSB, Servlet, EJB y BBDD.

¿Sabías que...?

- EJB es un componente Java web fundamental en cualquier aplicación Java web que quiera acceder a una BBDD.
- Llamamos scriptlet a cada una de las etiquetas con código Java que podemos insertar en medio del código HTML de una página JSP.



6.1. Introducción

A medida que la tecnología avanza, las aplicaciones cada vez son las complejas para poder dar respuesta a las nuevas complejidades que van surgiendo. El número de clases que compone una aplicación cada vez crece más y conseguir cohesionar todas estas clases aplicando criterios de reutilización, eficiencia y calidad es una tarea cada vez más difícil.

Para intentar hacer frente a estas nuevas dificultades, apareció el concepto de programación basada en componentes. Este tipo de programación nos permite reutilizar trozos de código obteniendo beneficios como mejor calidad, reducción del tiempo de desarrollo y mayor retorno sobre la inversión.



6.2. Componentes

Un componente, desde un punto de vista de programación, es cada una unidad ejecutable e independiente que compone una aplicación, que tiene una funcionalidad y una forma de interactuar perfectamente definidas, de forma que su ensamblaje con el resto de componentes se pueda realizar sin tener que modificar el código interno de ninguno de ellos y, además, en cualquier momento sea posible cambiar un componente por otro equivalente.

Un componente puede ser visual, como los proporcionados por los IDEs de desarrollo para poderlos incluir en interfaces gráficas de usuario, pero también pueden ser no visuales, los cuales tienen funcionalidades como si fueran librerías remotas.

Las características de un componente son las siguientes:

- Un componente es una unidad ejecutable que puede ser instalada y utilizada de forma independiente.
- Un componente puede interactuar y operar con otros componentes desarrollados por terceros, es decir, cualquier compañía o programador puede utilizar un componente y agregarlo a la aplicación que esté realizando, dicho de otra forma, los componentes los podemos componer.
- Un componente no tiene estado, al menos su estado no es visible externamente.
- Un componente y la programación orientada a componentes la podemos asociar a los componentes electrónicos y al uso que hacemos de ellos para integrarlos dentro de un sistema mayor.



La forma más común de identificar un componente es cuando hablamos de plugins o extensiones. Un plugin es una unidad ejecutable e independiente que puede formar parte de una aplicación con la cual interacciona. Su incorporación no requiere de ninguna modificación tanto de la aplicación como del propio plugin. Su instalación es muy sencilla y en cualquier momento podemos cambiar un plugin por otro o añadir nuevos.

Un plugin es un componente muy independiente que añade funcionalidad a una aplicación, pero no es necesario para la ejecución de la propia aplicación ni para el resto de funcionalidades. No todos los componentes tienen esta independencia, algunos componentes forman parte de núcleo funcional de la aplicación y, por lo tanto, su rendimiento afectará a la ejecución de la aplicación. No es posible ejecutar la aplicación si los eliminamos, por lo que si los eliminamos tendremos que cambiarlos por un componente equivalente.

Como ejemplos de modelos de componentes tenemos los ensamblados de Microsoft.NET que son una agrupación lógica de uno o más módulos o ficheros de recursos, los Entrepise Java Beans (EJB) o el modelo de componentes de CORBA (Common Object Request Broker Architecture). En nuestro caso como el objetivo es trabajar con una aplicación orientada a objeto basada en Java, la opción que trabajaremos serán los EJB.

Un EJB encapsula parte de la lógica de la aplicación, pudiendo acceder a recursos como BBDD y otros EJB. Un EJB puede ser accedido desde otros EJBs, servlets, servicios web y aplicaciones cliente.



Un componente de software, en nuestro caso los EJB, los podemos caracterizar de la siguiente forma:

- Atributos
- Operaciones + Eventos
- Comportamiento
- Protocolos + Escenarios
- Propiedades

Los atributos, las operaciones y los eventos forman la interfaz de un componente. El comportamiento representa la semántica del componente. Los protocolos establecen la interoperabilidad entre componentes, es decir, la compatibilidad en el envío de mensajes entre componentes y sus correspondientes comportamientos antes los diferentes escenarios que se puedan dar. Las propiedades son características extrafuncionales que puede tener un componente.

6.2.1. Atributos

Los atributos son uno de los elementos más importantes de la interfaz de un componente ya que representan su parte pública, es decir lo que el resto de componentes ven de ellos. Estos elementos son accesibles externamente y son muy importantes desde el punto de vista del control y uso del componente, siendo la base del resto de características que determinan como es un componente.



6.2.2. Operaciones + Eventos

La interfaz pública de un componente contiene información de la cabecera de las funciones que contiene (es decir su nombre, lista y tipo de parámetros, y retorno). Gracias a esta información, el resto de componentes saben cómo interactuar con él. Este tipo de interacción tiene el nombre de interacción proactiva, ya que las operaciones de un componente se ejecutan cuando otro componente las llama para su ejecución. El otro tipo de interacción es la reactiva y está directamente relacionada con los eventos de un componente. En este tipo de interacción, un componente genera un evento que se corresponde con la llamada a una determinada operación. El resto de componentes de la aplicación recogen estos eventos, activando la ejecución de las operaciones correspondientes al evento. Este tipo de interacción es la misma que ocurre en las interfaces gráficas de usuario.

6.2.3. Propiedades

Las propiedades de un componente determinan su estado diferenciándolo del resto de componentes. En párrafos anteriores hemos comentado que una de las características de un componente es que no tiene estado, cosa que se contradice con esto. El tema está en que los componentes tienen una serie de propiedades a las que podemos acceder y modificar de diferentes maneras.

Para poder consultar y modificar las propiedades de un componente, disponemos de métodos o funciones que nos permiten acceder a ellas:

- Métodos get: estos métodos los utilizamos para consultar el valor de una propiedad.

`tipoPropiedad getNombrePropiedad()`

- Métodos set: estos métodos los utilizamos para modificar el valor de una propiedad.

`void setNombrePropiedad(tipoPropiedad valor)`



Resumiendo, las propiedades disponen de los métodos get/set al igual que lo hace un atributo de una clase POJO.

Tenemos diferentes tipos de propiedades:

- Simples e indexadas.

Una propiedad simple representa un único valor. Una propiedad indexada representa un conjunto de valores. Todos los elementos que forman parte de una propiedad indexada tienen el mismo tipo y accedemos a ellos utilizando un índice. Para acceder a ellas, utilizaremos los correspondientes métodos get/set, teniendo en cuenta que la sintaxis de estos métodos cambia:

```
setNombrePropiedad(int índice, tipoPropiedad valor)
```

- Compartidas.

Una propiedad compartida se caracteriza por notificar a todas las partes interesadas cuando cambian. Solo notifican a las partes interesadas la naturaleza del cambio. La forma en cómo funciona el sistema de notificaciones está basada en eventos. Tenemos un componente que es el origen del cambio que mediante un evento notifica a un componente receptor cuando se produce un cambio en la propiedad compartida. Los componentes interesados en conocer los cambios en la propiedad compartida, tienen que registrarse en ella. Este tipo de propiedades no son bidireccionales por lo que los componentes receptores no pueden generar una respuesta.

Para que un componente pueda tener propiedades de este tipo, tiene que proporcionar dos métodos, que son los que utilizarán el resto de componentes para registrar o des-registrar su interés por los cambios en la propiedad compartida. Estos métodos son:

```
addPropertyChangeListener(PropertyChangeListener x)  
removePropertyChangeListener(PropertyChangeListener x)
```



- **Restringidas.**

Las propiedades restringidas funcionan al revés de las compartidas.

Cuando una propiedad quiere cambiar su valor, necesita pedir la aprobación de los componentes registrados en ella como validadores de cambios. Sin la aprobación de todos sus componentes registrados, la propiedad no podrá cambiar de valor.

Los métodos para registrar o des-registrar validadores de una propiedad restringida son:

```
addVetoableChangeListener(VetoableChangeListener x)  
removeVetoableChangeListener(VetoableChangeListener x)
```

6.2.4. Descubrimiento de características de un componente

La introspección es aquel proceso que permite a un IDE de desarrollo descubrir las características que tiene un componente, es decir, podemos definir la introspección como el mecanismo a través del cual podemos descubrir las propiedades, métodos y eventos que tiene un componente.

La introspección se puede producir de dos formas:

- En el momento de poner el nombre a las características de los componentes, podemos seguir una nomenclatura específica. En el caso de los EJBs, la clase Inspector se encarga de inspeccionar la estructura de un EJB aplicando patrones para descubrir sus características.
- Crear una nueva clase que contenga información implícita sobre propiedades, métodos y eventos del componente. Al inspeccionar esta clase, ya tendremos todas sus características.



Los EJBs trabajan con la introspección a diferentes niveles. Al nivel más bajo lo conocemos como reflexión. Este mecanismo permite que los objetos Java descubran información sobre los métodos públicos, atributos y constructores de clases que hemos cargado durante la ejecución de la aplicación. Para que pueda utilizarse la reflexión todo lo que tenemos que hacer es crear un método o variable público para que lo podamos descubrir a través de la reflexión.

6.2.5. La API JPA

La persistencia en una aplicación consiste en que sus objetos existan más allá de la ejecución de la aplicación, para que existan y estén disponibles cuando la aplicación se pone en marcha y que guarden los cambios que la aplicación ha realizado en ellos cuando finaliza la ejecución de la misma.

En el caso de los EJBs, la API que proporciona Java para su persistencia es la JPA (Java Persistence API). JPA es una librería proporcionada por Java para hacer persistente cualquier tipo de objeto en una BBDD relacional. Es una API desarrollada para la plataforma J2EE y está incluida como estándar desde la versión EJB 3.0. Como es una interfaz, la podemos integrar con Hibernate u otro ORM.

Las clases que componen JPA son las siguientes:

- Persistence.

Esta clase contiene una serie de métodos estáticos que utilizamos para obtener un objeto de tipo EntityManagerFactory que sea independiente del fabricante que haya implementado JPA.

- EntityManagerFactory.

Esta clase contiene métodos que utilizamos para crear objetos de tipo EntityManager utilizando el patrón de diseño Factory.



- **EntityManager.**

Esta interfaz es la interfaz principal de la API JPA y la utilizamos para la persistencia de las aplicaciones. Cada EntityManager permite realizar operaciones típicas relacionadas con la persistencia de objetos (crear, modificar, borrar y consultar).

- **Entity.**

Esta clase la añadimos a las clases de los objetos que queremos hacer serializables. Cada uno de estos objetos representará un registro en la BBDD.

- **EntityTransaction.**

Cada objeto de tipo EntityManager tiene una relación de uno a uno con un objeto EntityTransaction. Nos permite realizar operaciones sobre datos persistentes de forma que agrupados formen una transacción, en el que todos se ejecutan sobre la BBDD o todos fallan en su ejecución.

- **Query.**

Esta interfaz está implementada por cada sistema para encontrar objetos que cumplan con un criterio de búsqueda. JPA define un estándar para estas búsquedas basadas en JPQL (Java Persistence Query Language) y SQL. Para obtener una instancia, lo haremos desde un EntityManager.



6.2.6. Tipos de EJB

Existen tres tipos de EJB:

- EJB de entidad.

Entity EJBs. El objetivo de este tipo de EJB es encapsular los objetos del lado del servidor que almacenan los datos. Este tipo de EJB tienen la característica fundamental de la persistencia.

- Persistencia gestionada por el contenedor (CMP). El contenedor se encarga de almacenar y recuperar los datos del objeto de entidad mediante un mapeado en una tabla de una BBDD.
- Persistencia gestionada por el EJB (BMP). El propio objeto entidad se encarga, mediante una BBDD u otro mecanismo, de guardar y recuperar los datos a los que hace referencia.

- EJB de sesión.

Session EJB. El objetivo de este tipo de EJB es gestionar el flujo de la información en el servidor. Generalmente sirven a los clientes como una fachada de los servicios proporcionados por otros componentes disponibles en el servidor. Podemos tener de dos tipos:

- Con estado (Stateful): son objetos distribuidos que tienen estado. El estado no es persistente, pero el acceso al EJB se limita a un único cliente.
- Sin estado (Stateless): son objetos distribuidos que carecen de estado asociado, permitiendo que podemos acceder a ellos de forma concurrente. No hay garantía de que los datos de los atributos de instancia se guarden entre llamadas a métodos.

- EJB dirigidos por mensajes.

Message-driven EJB. Son los únicos EJB con funcionamiento asíncrono.

Utilizan el Java Messaging Service (JMS) para subscribirse a un tópico (topic) o a una cola (queue) y se activan cuando el correspondiente tópico o cola reciben un mensaje. No necesitan ser instanciados por los clientes.



6.3. Desarrollo de componentes

Las aplicaciones que desarrollaremos basadas en componentes en Java, utilizan como componente fundamental el EJB. Tenemos muchas opciones a la hora de empaquetar componentes, en cambio aquellas aplicaciones centradas en EJB, es decir aplicaciones J2EE, se distribuyen en archivos EAR, que son archivos Java estándar (JAR) pero con extensión diferente. La utilización de este tipo de archivos nos permite ensamblar una gran cantidad de aplicaciones J2EE.

Un módulo J2EE está formado por uno o más componentes J2EE para el mismo tipo de contenedor y un descriptor de despliegue de componentes para ese tipo. Un descriptor de despliegue especifica los atributos de una transacción y las autorizaciones de seguridad para un EJB. Un módulo J2EE sin un descriptor de despliegue de aplicación puede ser desplegado como un módulo independiente.

Las aplicaciones J2EE tienen una parte de front-end web y una parte back.end con EJB. Esto implica que tenemos que crear un .ear que contenga a la aplicación con dos módulos: un .war y un ejb-jar. Esto es una buena forma de trabajar ya que creamos una separación clara entre el front-end y el back-end. En cambio, para aplicaciones sencillas hacer esta separación resulta complicado.

Los cuatro tipos de módulos de J2EE para aplicaciones web con EJB son los siguientes:

- Módulos EJB.

Contienen los ficheros con las clases EJB y un descriptor de despliegue. Estos módulos los empaquetamos como ficheros JAR, por lo tanto, con extensión .jar.

- Módulos web.

Contienen los ficheros son servlets, ficheros JSP, fichero de soporte de clases, ficheros GIF y HTML y un descriptor de despliegue de aplicación web. Estos módulos los empaquetamos como ficheros JAR con una extensión .war (web archive).



- Módulos de aplicaciones cliente.

Contienen los ficheros con las clases y un descriptor de despliegue de aplicación de cliente. Estos módulos los empaquetamos como ficheros JAR, por lo tanto, con extensión .jar.

- Módulos de adaptadores de recursos.

Contienen todas las interfaces Java, clases, librerías nativas y otra documentación junto con su descriptor de despliegue de adaptador de recursos.

Podemos empaquetar los EJB dentro de un archivo .war. Podemos incluir las clases en el directorio WEB-INF/clases o en un archivo .jar dentro de WEB-INF/lib. El .war puede contener como máximo un archivo ejb-jar.xml, el cual puede estar ubicado en WEB-INF/ejb-jar.xml o en el directorio META-INF/ejb-jar.xml de un directorio .jar.



6.4. Creación de una aplicación Java con EJB

El tipo de aplicación que vamos a crear al trabajar con EJB es una aplicación web. La estructura de esta aplicación estará formada por los siguientes elementos:

- Cliente: Página web HTML/CSS con JSP. Para acceder a este elemento, lo haremos desde un navegador.
- Servlet: elemento en la parte del servidor que recibirá las peticiones de la parte cliente y las procesará, llamado al EJB que corresponda.
- EJB: elemento en la parte del servidor que actuará como nexo de unión entre el servlet y la BBDD.
- BBDD: elemento que permitirá la persistencia en la aplicación.

Para que la estructura servlet/EJB, que son los elementos que forman parte del servidor funcione, necesitamos un servidor que los albergue y que por lo tanto permita su ejecución. Este elemento en nuestro caso es el servidor GlassFish, elemento que ya viene integrado en el IDE NetBeans. Este tipo de servidor no es el único que podemos utilizar, nos servirá cualquier tipo de servidor web que tenga un motor de soporte para servlets, como puede ser tomcat.

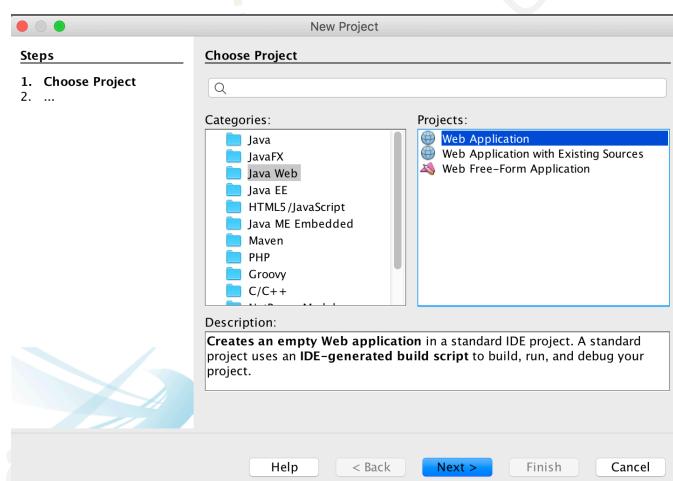


Para explicar todo el proceso de creación de una aplicación Java con EJB nos basaremos en un ejemplo. Para este ejemplo necesitaremos el IDE NetBeans, GlassFish y MySQL.

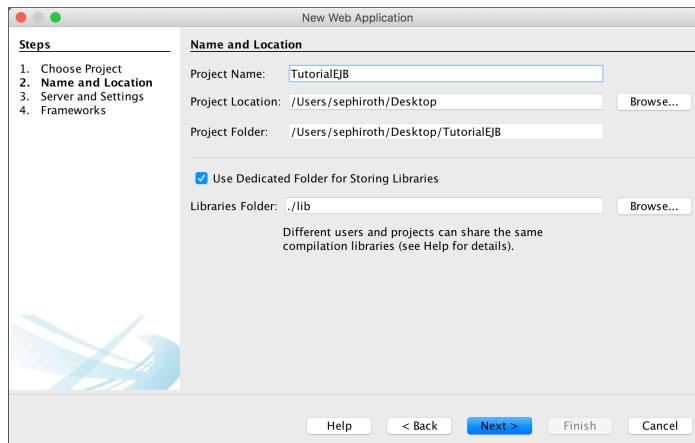
1. Creación del proyecto.

El tipo de proyecto que tenemos que crear es de tipo web, ya así tendremos la estructura necesaria para crear la parte cliente, la parte servidor e integrar en él el servidor web que permitirá que todo funcione.

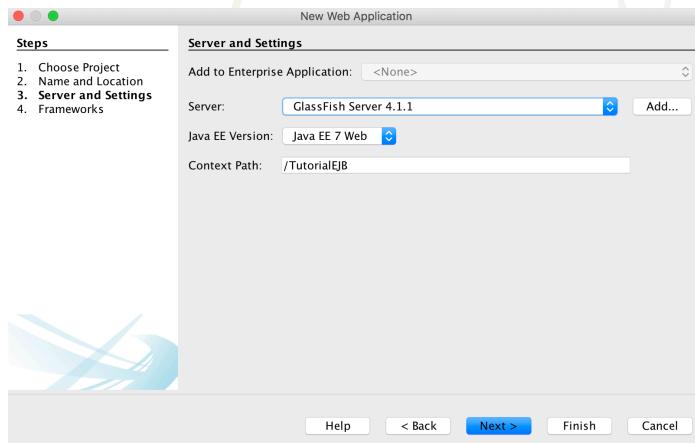
Para crear el proyecto, seleccionaremos la opción File – New Project. Aparecerá una ventana para seleccionar el tipo de proyecto que queremos crear. Como en nuestro caso es un proyecto web, tendremos que seleccionar la opción “Java Web” en el listado izquierdo y después la opción “Web Application” en el listado derecho.



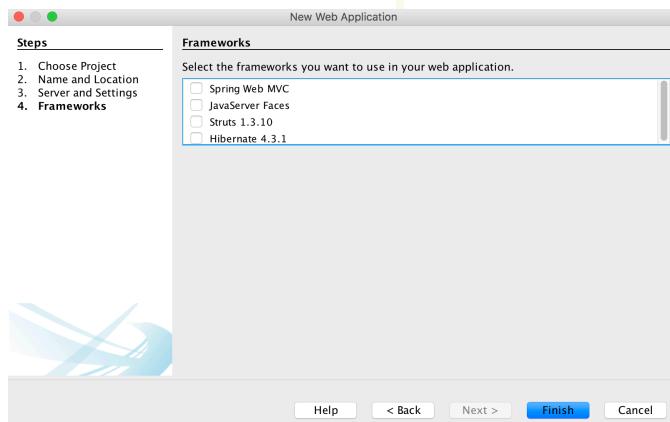
En la siguiente pantalla que nos aparece, indicaremos el nombre que queremos darle al proyecto. Además marcaremos la opción “Use Dedicated Folder for Storing Libraries” y dejar la ruta por defecto .\lib.



La siguiente pantalla nos pedirá el servidor web que queremos utilizar en el proyecto. En nuestro caso seleccionaremos GlassFish ya que viene integrado en NetBeans.

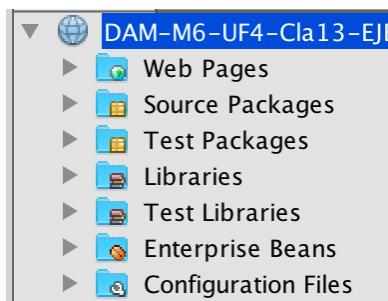


El último paso que nos queda para crear el nuevo proyecto es seleccionar los Frameworks que queremos incluir en el proyecto. Por ejemplo, uno de ellos podría ser Hibernate, en el caso de utilizar un ORM. Si no seleccionamos ninguno, JPA utiliza uno por defecto (EclipseLink).





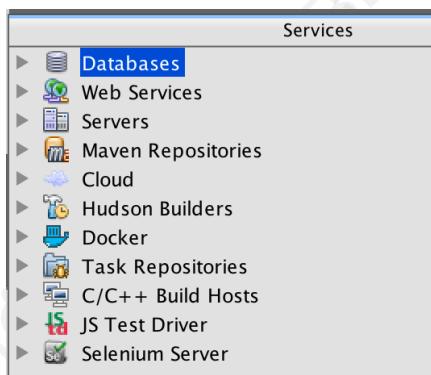
Una vez tenemos creado el nuevo proyecto, su estructura será la siguiente:



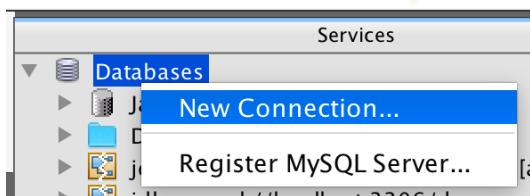
2. Base de datos

Para que una aplicación esté completa, necesitamos que tenga una BBDD. En nuestro ejemplo, esta BBDD será relacional. Para poder integrar esta BBDD en el nuevo proyecto que hemos creado, primero necesitamos que el NetBeans sepa de la existencia de esta BBDD. Para ello tenemos que crear un nuevo servicio en NetBeans que apunte a la BBDD con la que queremos trabajar.

Para crear este servicio, accedemos a la pestaña Services – Databases.

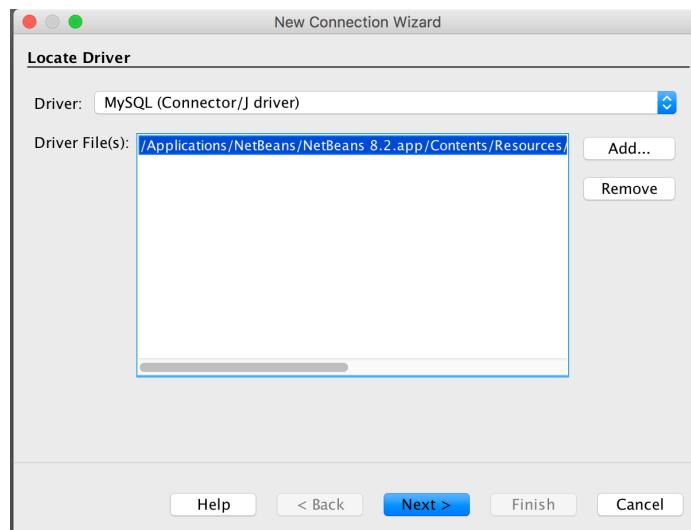


Crearemos una nueva conexión. Para ello pulsaremos con el botón derecho sobre Databases y seleccionaremos la opción New Connection:

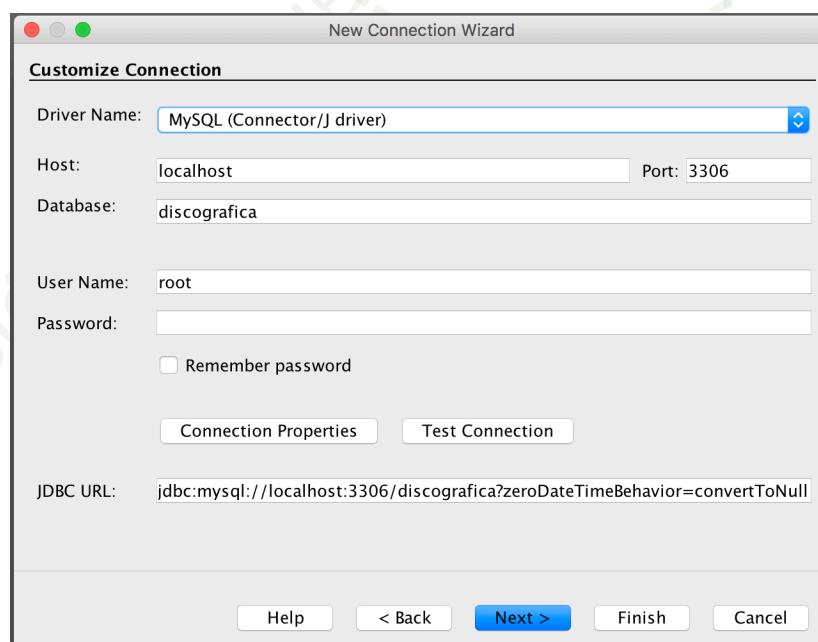




Lo siguiente que tenemos que indicar es el driver que vamos a utilizar para acceder a la BBDD. En el primer desplegable seleccionaremos MySQL. Aparecerá un driver en el campo de texto, lo seleccionaremos y pulsaremos siguiente.

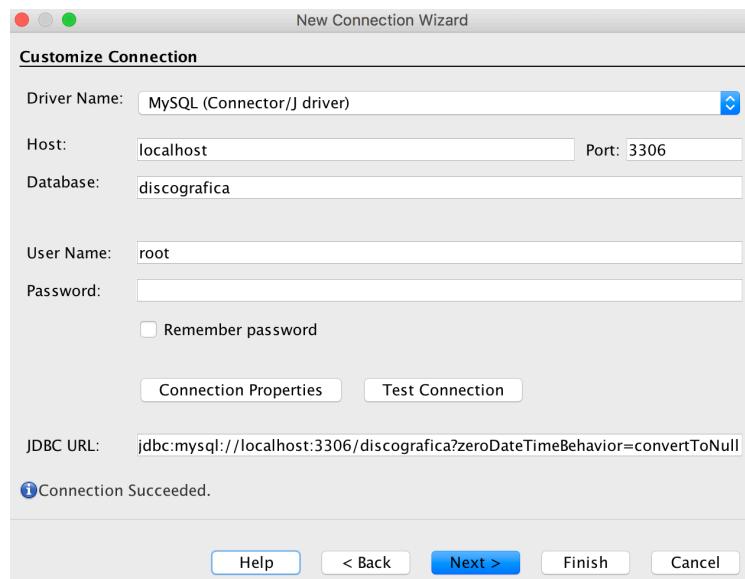


Ahora que ya hemos seleccionado el driver, configuraremos la conexión a la BBDD. Indicaremos el servidor (localhost), el puerto (3306), el nombre de la BBDD (discográfica), el nombre de usuario para acceder y la contraseña.

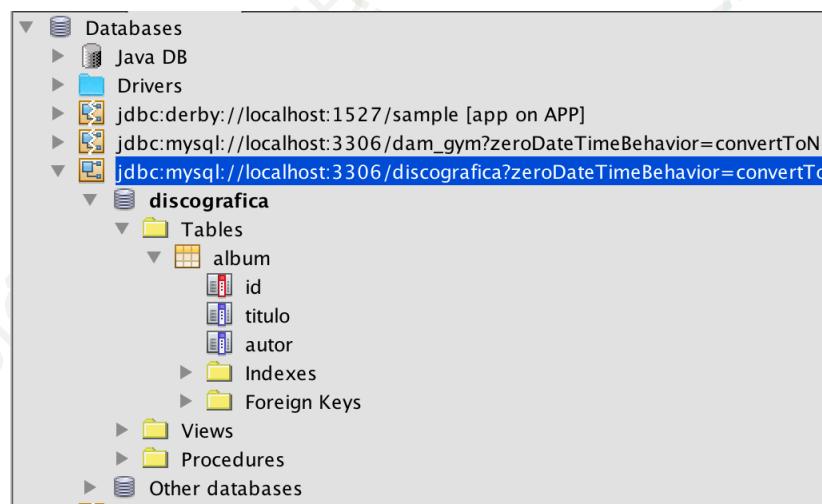




Antes de seguir con la configuración, podemos probar si los datos que hemos puesto son correctos y la conexión con la BBDD funciona. Para ello pulsaremos el botón “Test Connection” y si todo es correcto, veremos el mensaje de “Connection Succeeded” en la parte inferior izquierda de la pantalla.



En el resto de pantallas, dejamos las opciones por defecto. Al finalizar este proceso, tendremos una nueva conexión a la BBDD.

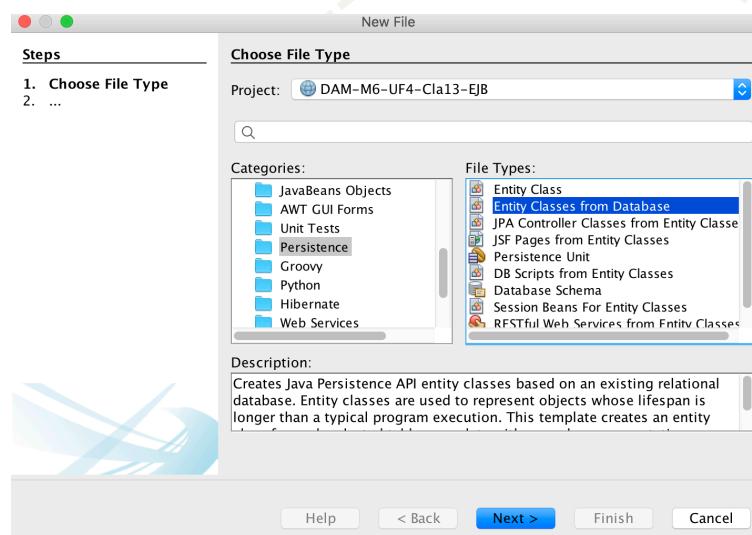




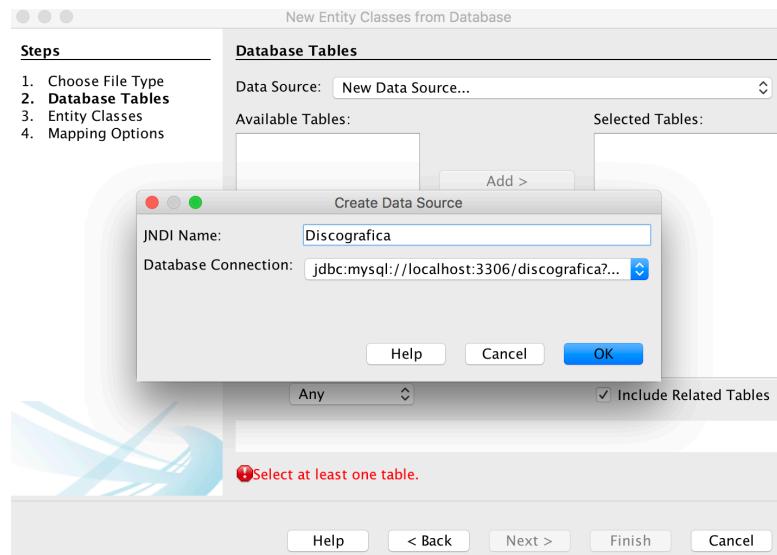
3. JPA

Ahora que ya tenemos la conexión a la BBDD, podemos añadir el soporte de JPA a nuestro proyecto. La API JPA es la que permite la persistencia de objetos Java y por lo tanto es un elemento necesario para que la aplicación esté completa.

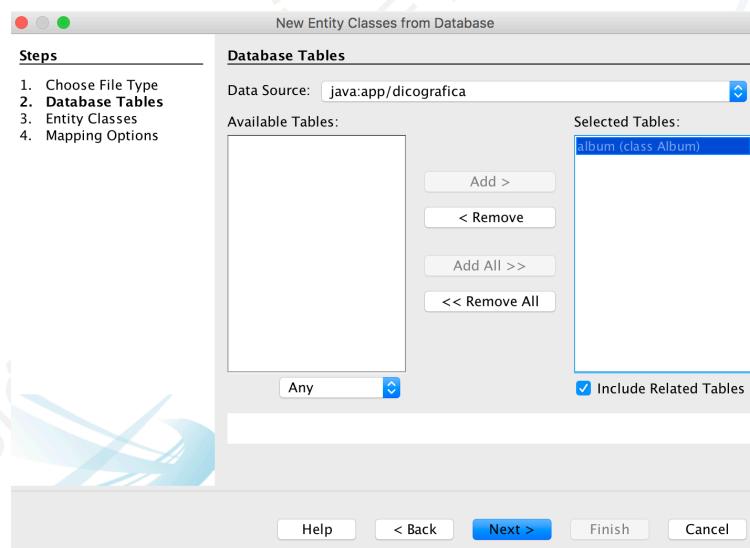
Para añadir el soporte JPA, en la raíz del proyecto pulsaremos con el botón derecho y seleccionaremos la opción New – Other. En la ventana que nos aparecerá, seleccionaremos Persistence en el listado izquierdo y “Entity Classes From Database” en el listado de la derecha.



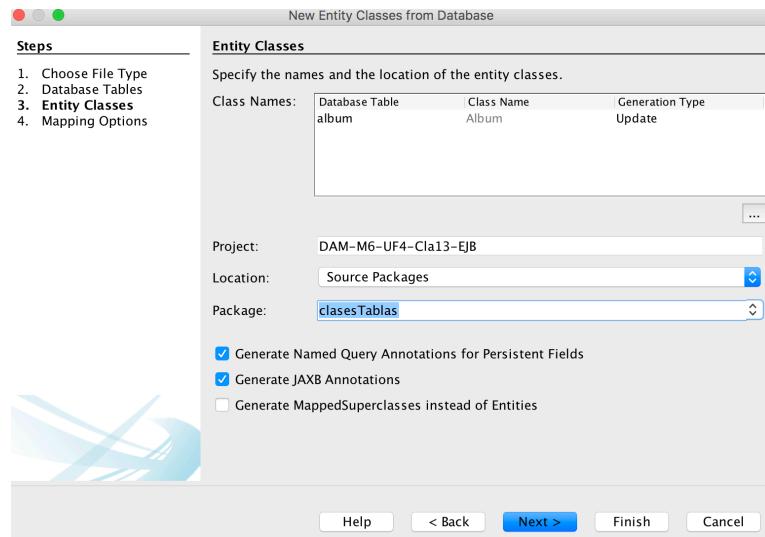
El objetivo de estos pasos es conectarnos a la BBDD y obtener las clases correspondientes a las tablas que tenemos en la BBDD. En la siguiente pantalla que nos aparecerá, en el desplegable seleccionaremos la opción “New Data Source” y configuraremos la conexión que hemos creado en el paso anterior:



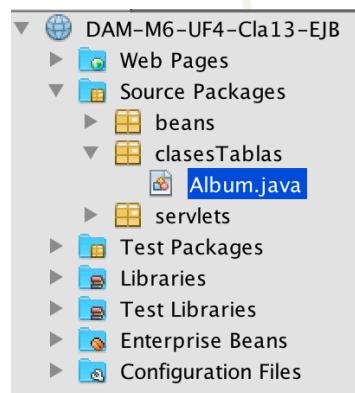
Cuando tengamos el nuevo Data Source creado, veremos el listado de tablas que contiene la BBDD en el panel de la izquierda. Tenemos que pasar al panel de la derecha todas las tablas de las que queramos crear la clase correspondiente en la aplicación. Para ello entre los dos paneles los botones para añadir y eliminar dichas tablas.



En la siguiente pantalla veremos la relación entre las tablas y las clases que generaremos. Es recomendable poner estas clases en un paquete y no dejarlas en el DefaultPackage de la aplicación. Para hacerlo, indicaremos el nombre del paquete en el apartado Package.



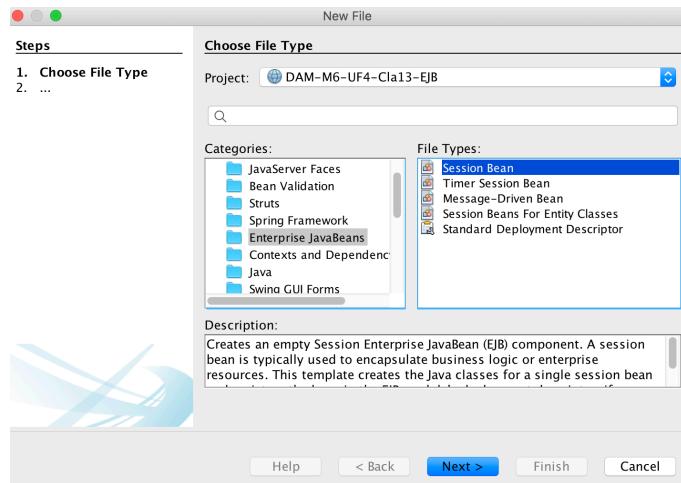
El resto de opciones del resto de pantallas las dejaremos con los valores por defecto. En este momento ya tenemos el elemento de persistencia asociado a la aplicación y las clases que se corresponden con las tablas de la BBDD creadas.



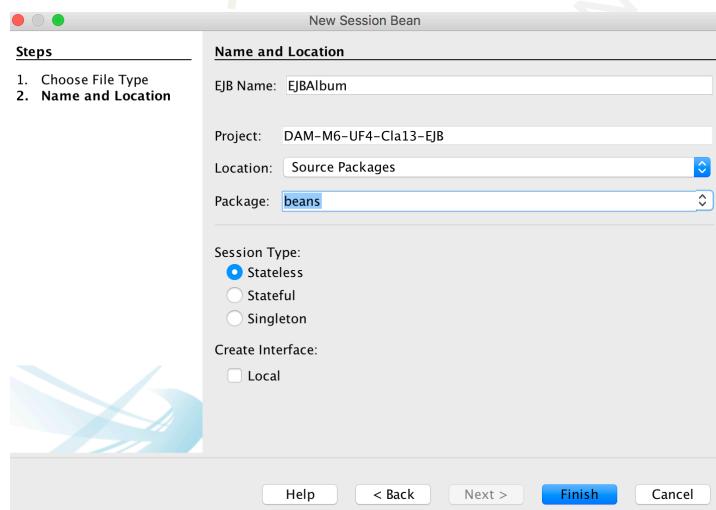
4. Creación del EJB

El siguiente elemento que tenemos que crear es el EJB. Para este ejemplo, crearemos un EJB de sesión stateless, es decir, cuando el cliente llama a un método del EJB, éste está listo para ser reutilizado por otro cliente, y la información en el EJB es desechada cuando el cliente deja de acceder al EJB.

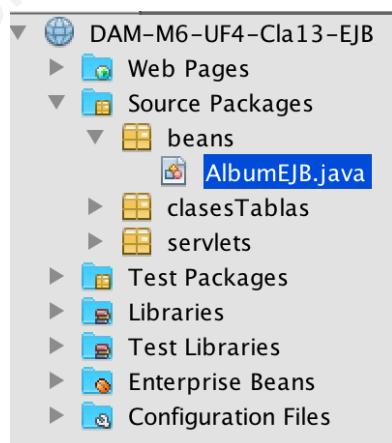
Para crear el EJB, lo primero que haremos será pulsar con el botón derecho y seleccionaremos la opción New – Other. En la ventana que nos aparecerá, seleccionaremos “Enterprise JavaBeans” en el panel izquierdo y “Session Bean” en el panel derecho.



En la siguiente pantalla le ponemos nombre, lo marcamos como Stateless y le indicamos en qué paquete del proyecto Java lo queremos guardar (en este caso el paquete se llamará beans).



Ahora ya tendremos el EJB creado.

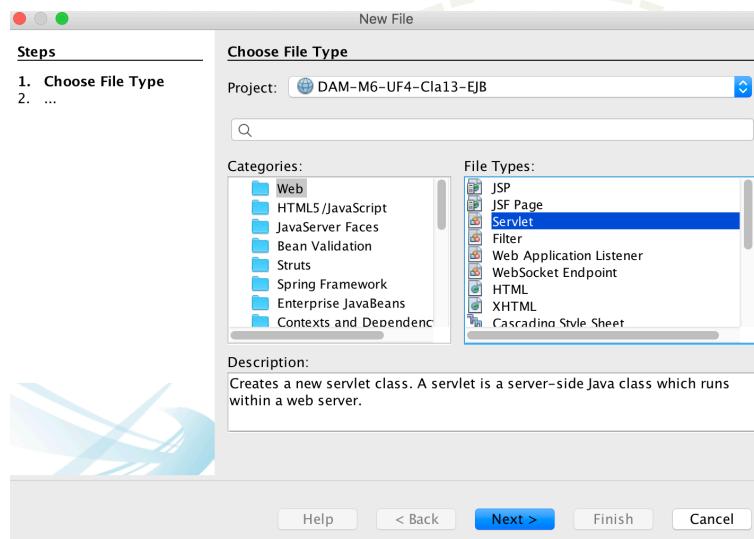




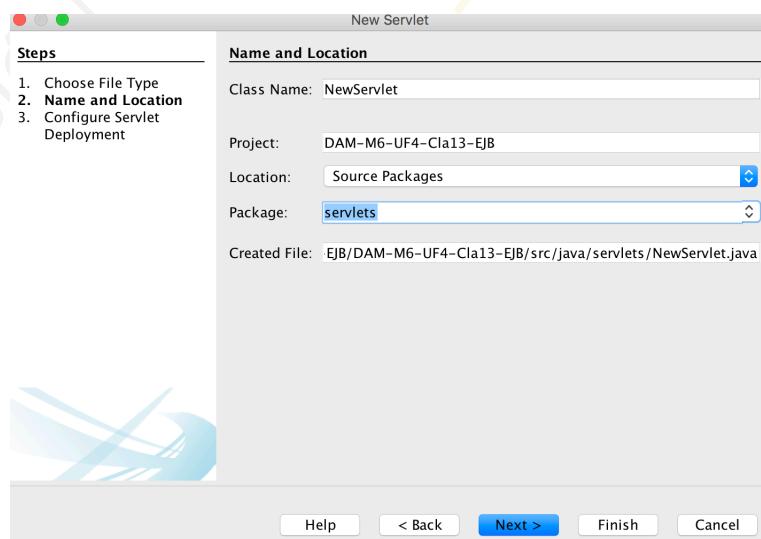
5. Creación del Servlet

El último elemento que nos queda por crear es el Servlet. Este elemento recibirá las peticiones del cliente, las procesará y llamará a una de las funcionalidades que engloba el EJB para que ejecute.

Para crear el Servlet pulsaremos con el botón derecho y seleccionaremos la opción New – Other. En la ventana que nos aparecerá, seleccionaremos Web en el panel izquierdo y Servlet en el panel derecho.

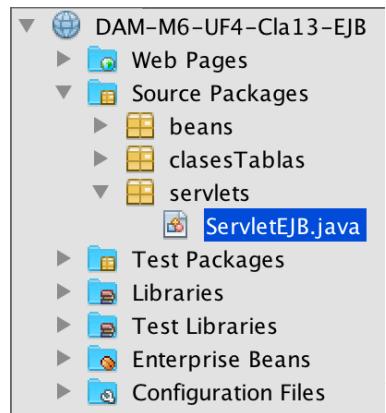


En la siguiente pantalla le ponemos nombre y le indicamos en qué paquete del proyecto Java lo queremos guardar (en este caso el paquete se llamará **servlets**).

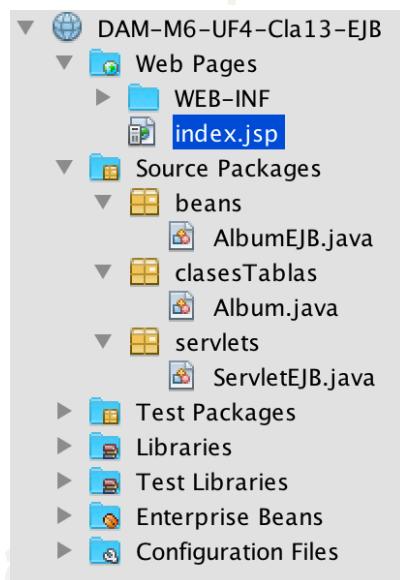




El resto de opciones las dejamos con los valores por defecto y ya tendremos el servlet creado.



En este punto ya tenemos la estructura del proyecto creada, ahora nos queda programar cada uno de los elementos para darle funcionalidad a la aplicación.





6.5. Persistencia

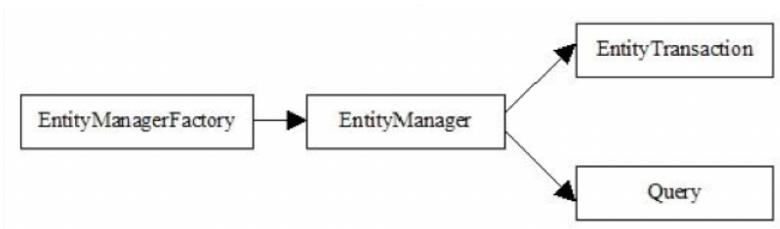
JPA o Java Persistence API es el standard de Java encargado de automatizar dentro de lo posible la persistencia de nuestros objetos en base de datos.

De esta forma tendremos a nuestra disposición un EntityManagerFactory con el que empezar a gestionar las entidades que se encuentran definidas. Ahora bien, muchas aplicaciones JEE se conectan a varias bases de datos y generan distintos EntityManagerFactories.

En un primer lugar un EntityManagerFactory es único y es con el que nosotros gestionamos todas las entidades. Si tenemos varias conexiones a base de datos deberemos definir un nuevo concepto que nos permite clarificar que tenemos dos EntityManagerFactories distintos.

Este concepto es el que se conoce como PersistenceUnit. Cada PersistenceUnit tiene asociado un EntityManagerFactory diferente que gestiona un conjunto de entidades distinto.

Una vez disponemos de un EntityManagerFactory este será capaz de construir un objeto de tipo EntityManager que como su nombre indica gestiona un conjunto de entidades o objetos. En principio estas entidades son objetos POJO normales con los cuales estamos trabajando en nuestro programa Java. El EntityManager será el encargado de guardarlos en la base de datos, eliminarlos de la base de datos etc.





- persist: encargado de almacenar entidades en la base de datos.
- find: se encarga de localizar una Entidad a través de su clave primaria. Para ello necesita que le pasemos la clave y el tipo de Entidad a buscar.
- remove: se encarga de eliminar una entidad de la base de datos.
- flush: este método es el encargado de sincronizar el PersistenceContext contra la base de datos. El EntityManager irá almacenando las modificaciones que nosotros realizamos sobre las entidades, para más adelante persistirlas contra la base de datos todas de golpe ya sea invocando flush o realizando un commit a una transacción.
- close: cierra la EntityManager.
- getTransaction: devuelve un objeto de tipo EntityTransaction.

6.5.1. JPQL

Java Persistence Query Language (JPQL) es un lenguaje de consulta orientado a objetos independientes de la plataforma definido como parte de la especificación Java Persistence API (JPA).

JPQL es usado para hacer consultas contra las entidades almacenadas en una base de datos relacional. Está inspirado en gran medida por SQL, y sus consultas se asemejan a las consultas SQL en la sintaxis, pero opera con objetos entidad de JPA en lugar de hacerlo directamente con las tablas de la base de datos.

Su estructura es la siguiente:

```
SELECT ... FROM ...
[WHERE ...]
[GROUP BY ... [HAVING ...]]
[ORDER BY ...]
```



Ejemplos:

```
// Query for a List of objects.  
Query query = em.createQuery("Select e FROM Employee e  
WHERE e.salary > 100000");  
List<Employee> result = query.getResultList();  
  
// Query for a single object.  
Query query = em.createQuery("Select e FROM Employee e  
WHERE e.id = :id");  
query.setParameter("id", id);  
Employee result2 = (Employee) query.getSingleResult();  
  
// Query for a List of data elements.  
Query query = em.createQuery("Select e.firstName FROM  
Employee e");  
List<String> result4 = query.getResultList();  
  
// Query for a List of element arrays.  
Query query = em.createQuery("Select e.firstName, e.lastName  
FROM Employee e");  
List<Object[]> result5 = query.getResultList();
```

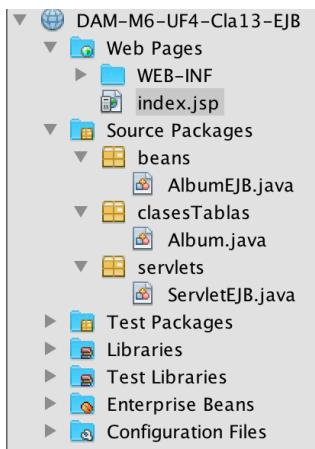


6.6. Aplicación final

En este apartado vamos a ver con detalle un ejemplo de aplicación web con EJB. En apartados anteriores hemos visto los pasos necesarios para crear el proyecto Java web y todos los elementos necesarios (conexión a la BBDD, EJB, servlet), pero solo hemos creado un proyecto vacío, el punto de partida. Lo que veremos ahora es cómo programar cada uno de esos elementos para poder poner en funcionamiento la aplicación y que realice lo que corresponda.

Los elementos que vamos a programar son los siguientes:

- Página JSP (index.jsp): página HTML/CSS con código Java insertado que ejecutaremos en el navegador, es decir, es la página cliente.
- BBDD: NetBeans tendrá una conexión a una BBDD creada como servicio.
- Clases Java: para cada una de las tablas de la BBDD tendremos una clase Java para representar los objetos que serán persistentes. Estas clases no las crearemos a mano, tal y como hemos visto en apartados anteriores, las crearemos con un proceso automático.
- GlassFish: servidor web con soporte de ejecución de servlets y EJBs que asignaremos al proyecto en su creación.
- EJB: componente de acceso a datos que contendrá la lógica de la aplicación y el acceso a la BBDD.
- Servlet: componente del servidor que recibirá las peticiones de los clientes, es decir la JSP, procesará dicha petición y llamará al método que corresponda del EJB.



6.6.1. JSP

Una página JSP es una página HTML (que puede contener estilos a través de un css) que tiene código java incrustado. A primera vista parece una página HTML normal, pero veremos la inserción de código Java a través de etiquetas <%@> que irán indicando la acción de la parte Java. Otra diferencia con una página web estándar es la extensión del archivo, en este caso es .jsp.

En nuestro ejemplo, la página se llama index.jsp y es la página que se cargará inicialmente en el navegador. En el ejemplo que estamos comentando, el proyecto lo tenemos en NetBeans por lo que tenemos la opción de ejecutar directamente la aplicación desde el IDE sobre el navegador. Para hacerlo, primero tendremos que asignar el navegador a NetBeans. Para ello tenemos un botón en la barra superior de iconos del NetBeans. Pulsando ese botón, se nos desplegarán una serie de opciones de las que tenemos que seleccionar el navegador que queramos asignar.



Una vez tengamos el navegador asignado, al ejecutar el proyecto y ser de tipo web, se abrirá el navegador con la página inicial, index.jsp. No nos tenemos que olvidar de poner en funcionamiento el servidor GlassFish.

Otra opción que tenemos es abrir el navegador y escribir la dirección de la página principal, tal y como haríamos para acceder a cualquier servidor web.

La página index.jsp es la siguiente:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h1>Albumes. Obtiene todos los álbumes con EJB </h1>
        <form action="ServletEJB" method="POST">Presiona el botón para obtener los datos.
            <input type="submit" name="enviar" value="Enviar" /></form>
    </body>
</html>
```

La página empieza con la etiqueta <%@page> indicando que es una JSP. Esta etiqueta tiene dos atributos, uno para indicar que es una página de texto/html y el otro para establecer la codificación, en este caso UTF-8.

El resto del contenido de la página es un HTML estándar. En este caso la pagina contiene un formulario con un botón. Al pulsar este botón, llamaremos al servlet llamado ServletEJB que tenemos en el lado del servidor.



6.6.2. Clases

Un elemento imprescindible en cualquier aplicación Java son las clases. En este caso las clases que se corresponden con los objetos persistentes las generaremos de forma automatizada para que su estructura sea equivalente al elemento de persistencia con el que trabajamos.

En apartados anteriores hemos visto dicho procedimiento. Para nuestro ejemplo, la clase Album que se corresponde con la tabla álbum de la BBDD quedaría de la siguiente forma. Debemos tener en cuenta que al generar la clase y ser un proyecto web, se han creado los elementos necesarios para que la clase pueda funcionar de forma correcta desde un punto de vista de la lógica de la aplicación y la persistencia.

La comentaremos por partes para poder ver claramente la estructura de la clase.

```
@Entity  
@Table(name = "album")  
@XmlRootElement  
@NamedQueries({  
    @NamedQuery(name = "Album.findAll", query = "SELECT a FROM Album a"),  
    @NamedQuery(name = "Album.findById", query = "SELECT a FROM Album a WHERE a.id = :id"),  
    @NamedQuery(name = "Album.findByTitulo", query = "SELECT a FROM Album a WHERE a.titulo = :titulo"),  
    @NamedQuery(name = "Album.findByAutor", query = "SELECT a FROM Album a WHERE a.autor = :autor")})  
public class Album implements Serializable {  
    private static final long serialVersionUID = 1L;  
    @Id  
    @Basic(optional = false)  
    @NotNull  
    @Column(name = "id")  
    private Integer id;  
    @Basic(optional = false)  
    @NotNull  
    @Size(min = 1, max = 20)  
    @Column(name = "titulo")  
    private String titulo;  
    @Basic(optional = false)  
    @NotNull  
    @Size(min = 1, max = 20)  
    @Column(name = "autor")  
    private String autor;
```

La clase empieza con la etiqueta `@Entity` que es una marca que indica que es una entidad con la que trabajaremos en los Servlets y EJB que puedan formar parte de la aplicación. Después tenemos la asociación con la tabla de la BBDD a través de la etiqueta `@Table`.



Lo siguiente que encontramos es la definición de una serie de consultas predefinidas. Cada una de ellas está formada por un nombre y la sentencia SQL correspondiente, de tal forma que cuando queramos ejecutarla, solo tendremos que llamarla por su nombre. Para cada clase tendremos la consulta de busca todos los elementos de la BBDD, es decir, un SELECT * FROM y una consulta por cada uno de los atributos que tenga la clase, de tal forma que podamos buscar en la tabla en función del valor concreto que tenga un atributo. Todas estas consultas están englobadas en la etiqueta @NamedQueries y la definición de cada una empieza con @NamedQuery.

Después tenemos la definición de la clase que como podemos ver implementa la interfaz Serializable, que es la marca que ponemos en todas las clases de las que queremos hacer persistentes sus objetos.

El primer elemento que tenemos en la clase es un atributo static que simplemente es un número para identificar la clase. Después tenemos la declaración de todos los atributos. Para cada uno de ellos tendremos una serie de etiquetas que empiezan con @, que indican propiedades del campo correspondiente en la BBDD, seguido de la propia definición en Java del atributo.

```
public Album() {  
}  
  
public Album(Integer id) {  
    this.id = id;  
}  
  
public Album(Integer id, String titulo, String autor) {  
    this.id = id;  
    this.titulo = titulo;  
    this.autor = autor;  
}
```



Después de la lista de atributos encontraremos los constructores. Como podemos ver en la imagen, tendremos 3 constructores:

- El constructor por defecto.
- El constructor con un parámetro que nos sirve para inicializar el atributo que se corresponde con la clave primaria de la tabla.
- El constructor que tiene tantos parámetros como atributos tiene la clase y nos sirve para poder inicializarlos.

```
public Integer getId() {  
    return id;  
}  
  
public void setId(Integer id) {  
    this.id = id;  
}  
  
public String getTitulo() {  
    return titulo;  
}  
  
public void setTitulo(String titulo) {  
    this.titulo = titulo;  
}  
  
public String getAutor() {  
    return autor;  
}  
  
public void setAutor(String autor) {  
    this.autor = autor;  
}
```

Lo siguiente es la lista de los métodos get/set para cada uno de los atributos.

```
@Override  
public int hashCode() {  
    int hash = 0;  
    hash += (id != null ? id.hashCode() : 0);  
    return hash;  
}  
  
@Override  
public boolean equals(Object object) {  
    // TODO: Warning - this method won't work in the case the id fields are not set  
    if (!(object instanceof Album)) {  
        return false;  
    }  
    Album other = (Album) object;  
    if ((this.id == null & other.id != null) || (this.id != null & !this.id.equals(other.id))) {  
        return false;  
    }  
    return true;  
}  
  
@Override  
public String toString() {  
    return "clasesTablas.Album[ id=" + id + "]";  
}
```

Lo último que tenemos en la clase es la sobrescritura de los métodos hashCode, equals y toString de forma adaptada a la clase.



Como hemos podido ver, la parte más interesante de la clase son las @NamedQueries ya que son las que nos van a permitir hacer consultas rápidas sobre la BBDD, el resto de la clase es básicamente una clase POJO con alguna adaptación por el hecho de trabajar con EJB.

6.6.3. Servlet

Un Servlet es un elemento Java en el lado del servidor que recibe las peticiones del cliente, las procesa y llama al método correspondiente del EJB. Cuando el EJB devuelve el resultado, el Servlet generará una página web de respuesta que enviará al cliente para que se visualice en el navegador.

La forma que tiene un Servlet de crear la página web de respuesta es enviando el código de la página HTML con valores de tipo String a través del método out.println. Por lo tanto, como programadores del Servlet tendremos que crear manualmente la página HTML de respuesta.



El Servlet del ejemplo es el siguiente:

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

    @EJB
    AlbumEJB aEJB;
    /**
     * Processes requests for both HTTP <code>GET</code> and <code>POST</code>
     * methods.
     *
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            List<Album> l = aEJB.findAll();
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet ServletEJB</title>");
            out.println("<link href='styles.css' rel='stylesheet' type='text/css' />");
            out.println("</head>");
            out.println("<body>");
            //out.println("<h1>Servlet ServletEJB at " + request.getContextPath() + "</h1>");
            for(int i=0; i<l.size();i++){
                out.print("<b>Titulo: </b> " +
                         l.get(i).getTitulo() +
                         ", <b>Autor: </b>" +
                         l.get(i).getAutor() + "<br>");
            }
            out.println("</body>");
            out.println("</html>");
        }
    }
}
```



La clase empieza con la etiqueta `@WebServlet` que marca que es un Servlet web. Los atributos de esta etiqueta son el nombre del Servlet y el patrón. Después ya tenemos la declaración de la clase que representa al Servlet, que como podemos ver hereda de la clase `HttpServlet` porque es un Servlet web.

Lo primero que encontramos en la clase en la declaración del EJB (`@EJB`) con el que trabajará el Servlet. En el ejemplo solo tenemos uno, pero podemos tener tantos elementos de tipo EJB como EJBs diferentes utilice el Servlet.

Uno de los métodos que hereda la clase de `HttpServlet` es `processRequest` y es el método que tenemos que sobreescibir para dar funcionalidad al procesamiento de la petición del cliente y la creación de la respuesta por parte del Servlet. Como podemos ver, este método tiene dos parámetros de entrada:

- El parámetro `request` representa a la petición que viene por parte del cliente y que por lo tanto utilizaremos para obtener la información que nos envía el cliente.
- El parámetro `response` representa la respuesta que generaremos para enviarla al cliente una vez hayamos procesado la petición. Es a través de este parámetro como crearemos y enviaremos la página web de respuesta.

También podemos ver que el método está marcado como que puede generar excepciones de tipo `ServletException` e `IOException`.

La llamada al método `response.setContentType("text/html; charset=UTF-8")`; establece el contenido de la página web de respuesta, tal y como declaramos en cualquier página web.

La llamada al método `PrintWriter out = response.getWriter();` obtiene un stream de salida de datos que es el que utilizaremos para ir enviando al navegador cliente la página web de respuesta. Como hemos comentado en apartados anteriores, esta web la enviaremos de forma manual y etiqueta por etiqueta.



Una vez ya tenemos el stream para ir enviando la respuesta, el ejemplo realizar una llamada a uno de los métodos que tenemos en el EJB, en este caso findAll(). En el siguiente apartado veremos cómo está programado nuestro EJB, pero de forma genérica decir que lo que hace este método es realizar una consulta a la BBDD para devuelva todos los registros existentes en la tabla Album y los almacene en una colección de tipo List. De ahí que la sentencia sea

```
List<Album> l = aEJB.findAll();
```

Después de esta sentencia, podemos ver cómo creamos y enviamos al navegador la página web con la respuesta. Como la primera parte es fija, son las etiquetas con la estructura de la página, podemos enviar ese contenido tal cual.

```
out.println("<!DOCTYPE html>");  
out.println("<html>");  
out.println("<head>");  
out.println("<title>Servlet EJB</title>");  
out.println("<link href='styles.css' rel='stylesheet' type='text/css' />");  
out.println("</head>");  
out.println("<body>");
```

Cuando el contenido dependa de valores de la aplicación, como en este caso, el valor String que enviamos al método out.println tendremos que crearlo con la concatenación de la parte fija HTML con los valores de la aplicación. En este caso, utilizamos los valores de los atributos título y autor de cada uno de los objetos devueltos por la consulta para mostrarlos en la respuesta. Por ello, vemos en el ejemplo que tenemos un bucle que se recorre la colección respuesta de la consulta a la BBDD y para cada objeto lo que hacemos es obtener el valor de estos atributos y concatenarlo con el resto de valor String que forma el HTML.

```
for(int i=0; i<l.size();i++){  
    out.print("<b>Titulo: </b>" +  
        l.get(i).getTitulo() +  
        ", <b>Autor: </b>" +  
        l.get(i).getAutor() + "<br>");  
}
```



6.6.4. EJB

EL EJB es el componente de acceso a datos de la aplicación y es el que contiene la funcionalidad principal de la aplicación. Tiene la definición de una serie de métodos que llamaremos desde el Servlet o desde otros EJB. Una de las funcionalidades que puedes proporcionar los EJBs es el acceso a la BBDD de la aplicación.

El EJB de nuestro ejemplo es el siguiente:

```
@Stateless
public class AlbumEJB {
    @PersistenceUnit
    EntityManagerFactory emf;
    public List findAll(){
        return emf.createEntityManager().createNamedQuery("Album.findAll").getResultList();
    }
}
```

Inicialmente el EJB está marcado con la etiqueta `@Stateless` ya que hemos creado un EJB sin estado. A continuación, ya tenemos de definición del EJB.

Los primero que encontramos es la creación del elemento que representa la unidad de persistencia y el nombre que le queremos dar al correspondiente objeto `@PersistenceUnit EntityManagerFactory emf;`

El método que tenemos en el EJB es un método de ejecuta una de las consultas predefinidas en la clase Album, en concreto la consulta que ejecuta es la de obtener todos los registros de la BBDD. La sentencia que utilizamos para ello es `emf.createEntityManager().createNamedQuery("Album.findAll").getR esultList();`

Sobre el objeto que representa la unidad de persistencia `emf`, creamos un objeto `EntityManager`. Sobre éste creamos una `NamedQuery` y le indicamos a través de un valor `String` el nombre de la `NamedQuery` que queremos ejecutar. Después sobre esta `NamedQuery`, llamamos al método `getResultSet` que nos devolverá en forma de colección de tipo `List` los objetos que concuerdan con la consulta que hemos ejecutado.

Como vemos en la cabecera del método, esta lista la devolvemos y es lo que recogíamos en el Servlet cuando hemos comentado la llamada al método del EJB.



En este ejemplo el EJB solo tiene un método que utiliza una de lasNamedQuery creadas en una de las clases de la aplicación, pero cualquier EJB además de la utilización de estasNamedQuery podemos crear métodos que realicen el resto de operaciones que podemos realizar en una BBDD, es decir, insertar, modificar, eliminar. Para ello utilizaremos los métodos persist y remove comentado anteriormente.

Aquí tenemos un pequeño ejemplo de lo que sería insertar en la BBDD un objeto de tipo Actividad:

```
public void insertActividad(Actividad a) {  
    EntityManager em = emf.createEntityManager();  
    em.persist(a);  
    em.close();  
}
```



Recursos y Enlaces

- [Java](#)



- [API Java 11](#)



- [NetBeans](#)



- [JSP Java API](#)





- [Servlet Java API](#)



- [EJB Java API](#)



Conceptos clave

- **EJB:** componente de acceso a datos programado en Java. Es el elemento cuyos métodos son llamados por un Servlet u otro EJB para ejecutar una funcionalidad como puede ser el acceso a una BBDD.
- **JSP:** elemento Java web de la parte cliente. Está formado por una página con código HTML y CSS con código Java insertado.
- **Servlet:** elemento Java de la parte del servidor que recibe las peticiones de un cliente, las procesa, ejecuta la funcionalidad correspondiente y genera una página web de respuesta.
- **JPA:** API de Java que contiene clases y métodos para gestionar la persistencia de una aplicación Java web.



Test de autoevaluación

1. Pon el significado de cada una de las siglas de JPQL:
 - a) J
 - b) P
 - c) Q
 - d) L
2. ¿Cómo se llama el elemento del lado de servidor de una aplicación Java web encargado de procesar la petición del cliente?
 - a) EJB
 - b) JSP
 - c) Servlet
 - d) BBDD
3. Cuál de los siguientes NO es un tipo de EJB:
 - a) Persistencia
 - b) Sesión
 - c) Entidad
 - d) Dirigido por mensaje



Ponlo en práctica

Actividad 1

Crea una función dentro de un EJB que elimine un objeto.





SOLUCIONARIOS

Test de autoevaluación

1. Pon el significado de cada una de las siglas de JPQL:

- a) J – Java
- b) P – Persistence
- c) Q – Query
- d) L – Language

2. ¿Cómo se llama el elemento del lado de servidor de una aplicación Java web encargado de procesar la petición del cliente?

- a) EJB
- b) JSP
- c) Servlet
- d) BBDD

3. Cuál de los siguientes NO es un tipo de EJB:

- a) Persistencia
- b) Sesión
- c) Entidad
- d) Dirigido por mensaje



Ponlo en práctica

Actividad 1

Crea una función dentro de un EJB que elimine un objeto.

Solución:

```
public boolean deleteMatricula(Matricula m) {  
    EntityManager em = emf.createEntityManager();  
    Matricula aux = em.find(Matricula.class, m.getIdmatricula());  
    if (aux != null) {  
        em.remove(aux);  
        em.close();  
        return true;  
    }  
    return false;  
}
```