# Boletín 5. Creación de una aplicacion J2EE con JPA

BDAW Tema 5 –La *Java Persistence API* (JPA) 12 de marzo de 2014

## Índice


**1 Introducción 2**
**2 Conexiones a la base de datos 2**
2.1 Configuración del *pool* de conexiones en Glassfish . . . 2
2.2 Configuración de la fuente de datos en Eclipse . . . 2
**3 Creación de la aplicación empresarial 3**
**4 Creación del proyecto EJB 3**
4.1 Añadir la perspectiva JPA a nuestro proyecto . . . 3
4.2 Definición de la `PersistenceUnit` . . . 5
4.3 Importación de las entidades desde la base de datos . . . 5
4.4 La interface `Dao` . . . 7
4.5 La clase `JpaDao` . . . 9
4.6 El DAO de la clase `Empleado` . . . 11
4.7 El DAO de la clase `Departamento` . . . 12
4.8 La clase de utilidad `FinalString` . . . 12
4.9 El EJB `EmpleadoBO` . . . 13
4.10 El EJB `DepartamentoBO` . . . 16
**5 Creación de la aplicación web dinámica 18**
5.1 El módulo de gestión de empleados . . . 18
5.1.1 Mostrar la lista de todos los empleados . . . 19
5.1.2 Añadir un nuevo empleado . . . 20
5.1.3 Buscar empleados por apellidos . . . 22
5.1.4 Buscar empleados por departamento . . . 24
5.2 El módulo de gestión de departamentos . . . 25
5.2.1 Mostrar la lista de todos los departamentos . . . 25
5.2.2 Añadir un nuevo departamento . . . 26
5.3 Desplegar y ejecutar la aplicación . . . 28
**6 Ejercicio 28**


## Índice de figuras


1 Paneles de configuración de una nueva fuente de datos en Eclipse. . . . 3
2 Panel de creación del proyecto empresarial `jpaPersonal`. . . . 4
3 Paneles de creación del proyecto EJB `jpaPersonalEJB`. . . . 4
4 Panel de propiedades del proyecto `jpaPersonalEJB` mostrando las facetas del proyecto. . . . 5
5 Panel de configuración de la faceta `Java Persistence` del proyecto `jpaPersonalEJB`. . . . 6
6 Panel de selección de tablas para la proyección ORM desde tablas a entidades Java. . . . 7
7 Configuración de la relación unidireccional entre un empleado *manager* y los departamentos que gestiona. 8
8 Configuración de la relación entre un departamento y la lista de sus empleados. . . . 8
9 Último panel de creación de las entidades. . . . 9
10 Panel de creación de la clase `JpaDao<K, E>`. . . . 10
11 Creación del EJB stateless `EmpleadoBo` y de la interface remota `EmpleadoBoRemote`. . . . 13
12 Diagrama de secuencia del caso de uso que obtiene la lista de todos los empleados. . . . 19
13 Diagrama de secuencia del caso de uso que obtiene la lista de todos los empleados. . . . 20
14 Diagrama de secuencia del caso de que permite buscar empleados por sus apellidos. . . . 22
15 Diagrama de secuencia del caso de que permite buscar empleados adscritos a un departamento. . . . 24
16 Lista de empleados tal y como se presenta a través del navegador. . . . 28
17 Vista desde el navegador de la lista de departamentos . . . 29

# Índice de listados


1 Código de la interface `Dao.java`. . . . . 7
2 Código de la clase `JpaDao.java`. . . . . 9
3 Código de la clase `EmpleadoDao.java`. . . . . 11
4 Código de la clase `DepartamentoDao.java`. . . . . 12
5 Código de la clase de utilidad `FinalString.java`. . . . . 13
6 Código de la interface `EmpleadoBoRemote.java`. . . . . 13
7 Código de la clase `EmpleadoBo.java`. . . . . 14
8 Código de la interface `DepartamentoBoRemote.java`. . . . . 16
9 Código de la clase `DepartamentoBo.java`. . . . . 16
10 Código del *servlet* `EmpleadoList.java`. . . . . 19
11 Código del *servlet* `EmpleadoNew.java`. . . . . 20
12 Código del *servlet* `EmpleadoFindByName.java`. . . . . 22
13 Código del *servlet* `EmpleadoFindByDept.java`. . . . . 24
14 Código del *servlet* `DepartamentoList.java`. . . . . 25
15 Código del *servlet* `DepartamentoNew.java`. . . . . 26


# 1 Introducción

El punto de partida de este boletín es la pequeña aplicación de gestión de personal que desarrollamos en el boletín anterior.

Sin embargo, en este ejercicio vamos a implementar los DAOs de las clases Empleado y Departamento como EJBs en un servidor de aplicaciones utilizando JPA para las operaciones básica de la base de datos.

Una cuestión que emerge como consecuencia de este cambio de modelo es que los DTOs, al menos como los hemos descrito hasta ahora, desaparecen y son sustituidos por *Entities*. Las entidades las importaremos directamente de la base de datos mediante el ORM (*Object-Relational Mapping*) de JPA. El ciclo de vida de estas entidades será gestionado directamente por JPA. Este cambio supone una diferencia fundamental respecto al caso anterior, en el que era responsabilidad del programador construir y mantener estos objetos.

Reimplementaremos después la pequeña aplicación web dinámica que gestionará los empleados y departamentos de nuestra empresa.

# 2 Conexiones a la base de datos

## 2.1 Configuración del *pool* de conexiones en Glassfish

El pool de conexiones `jdbc/personalpool` ya fue creado en el ejercicio anterior. Si por algún realmente extraño motivo necesitara volverlo a definir, por favor refierase a las instrucciones allí indicadas.

## 2.2 Configuración de la fuente de datos en Eclipse

En un proyecto JPA existe una equivalencia directa (ORM) entre los objetos definidos en la base de datos (tablas) y las clases Java. Esto se puede llevar a cabo de dos formas:

- Proyectando las clases Java en tablas del modelo relacional.
- Proyectando las tablas del modelo relacional en clases de Java. Esta es la opción elegida en este ejemplo.

Para cualquiera de estas dos operaciones es necesario definir en Eclipse una conexión a la base de datos personal. Seguimos el siguiente procedimiento:

1. Seleccionams la solapa **Data Source Explorer** . Pulsamos con el botón derecho sobre **Database Connections** y seleccionamos **New...** en el menú emergente.
2. En el primer panel (ver figura 1 izquierda), seleccionamos la línea `MySQL` en el área `Connection Profile Types` y cumplimentamos los campos `Name:` y `Description (optional):`. Pulsamos sobre **Next** para pasar al siguiente panel.
3. En este panel (figura 1 derecha) introducimos los siguientes valores:
   - Database: `personal`.
   - URL: `jdbc:mysql://localhost:3306/personal`.
   - User Name: `usuario`.
   - Password: `usupw` (los caracteres serán sustituidos por •).

- Marque la casilla ☑**Save password**.

Comprobamos que la conexión está bien definida pulsando sobre el botón **Test Connection** .

**Aviso:** Si no es posible contactar con la base de datos, repita y repase todo el proceso hasta dar con la configuración correcta.

Una vez comprobada la validez de la conexión, pulsamos sobre el botón **Finish** .

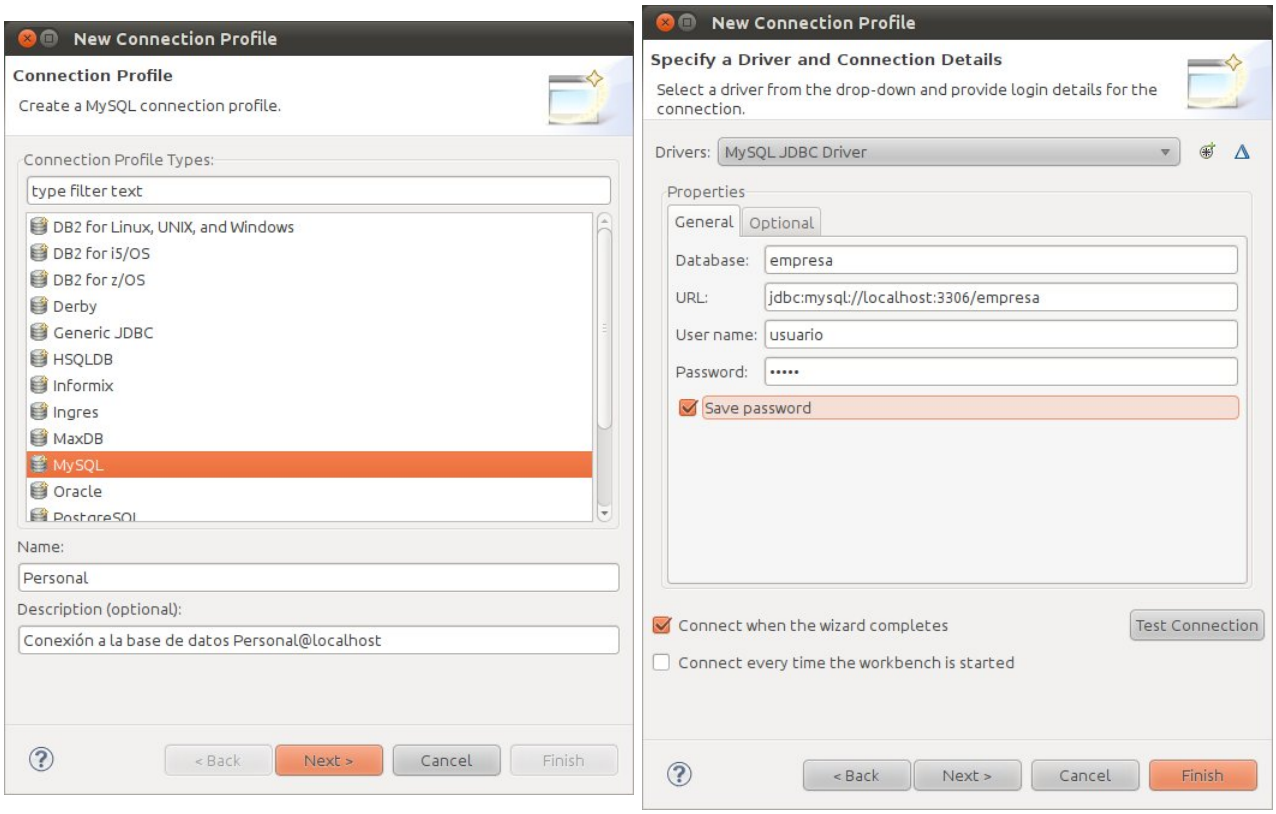


Figura 1: Paneles de configuración de una nueva fuente de datos en Eclipse.

# 3 Creación de la aplicación empresarial

En este apartado y en los siguientes se explica paso a paso el proceso de creación de la aplicación J2EE empresarial que vamos a desarrollar. Como veremos, la aplicación estará formada por varios proyectos de Eclipse interrelacionados que crearemos de forma secuencial.

Primero crearemos el proyecto empresarial que servirá de núcleo alrededor del cual se organizan los otros sub-proyectos. Para ello, en Eclipse seleccionamos el menú **File** → **New** → **Enterprise Application Project** . Crear el proyecto `jpaPersonal` a través del panel correspondiente como se muestra en la figura 2.

# 4 Creación del proyecto EJB

En Eclipse seleccionamos el menú **File** → **New** → **EJB Project** . A través del panel pondremos nombre al proyecto (`jpasPersonalEJB`) y lo asignaremos al proyecto EAR `jpaPersonal` marcando la casilla **EAR Membership** como se muestra en la figura 3.

## 4.1 Añadir la perspectiva JPA a nuestro proyecto

Para que nuestro proyecto se convierta en un proyecto JPA es necesario añadirle la perspectiva correspondiente. Para ello seleccionamos el proyecto con el botón de la derecha y en el menú emergente seleccionamos **Properties** . Los pasos a seguir son:

1. En el panel `Properties` seleccionamos la entrada `Project Facets` en la lista de la izquierda y marcamos la casilla ☑**Java Persistence** como se muestra en la figura 4

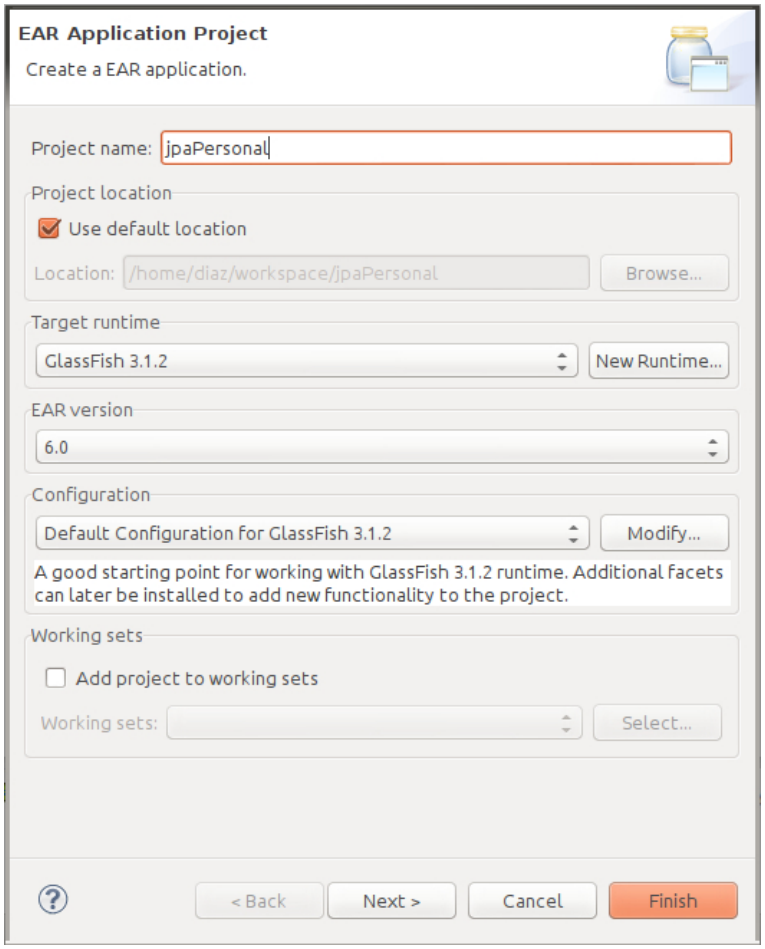


Figura 2: Panel de creación del proyecto empresarial `jpaPersonal`.

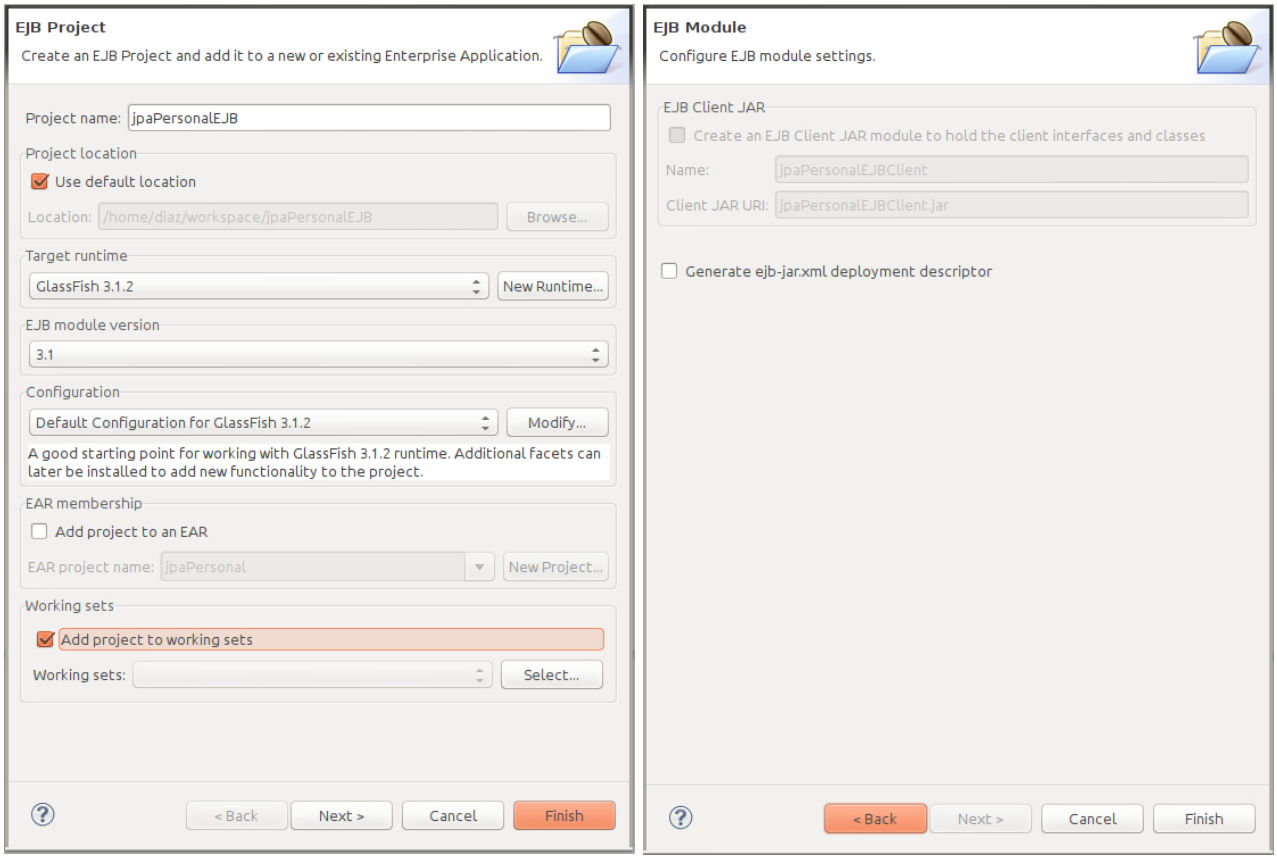


Figura 3: Paneles de creación del proyecto EJB `jpaPersonalEJB`.

2. En el mismo panel seleccionamos el enlace `Further configuration available...` y seleccionamos en el panel emergente la conexión `personal` como se muestra en la figura 5.

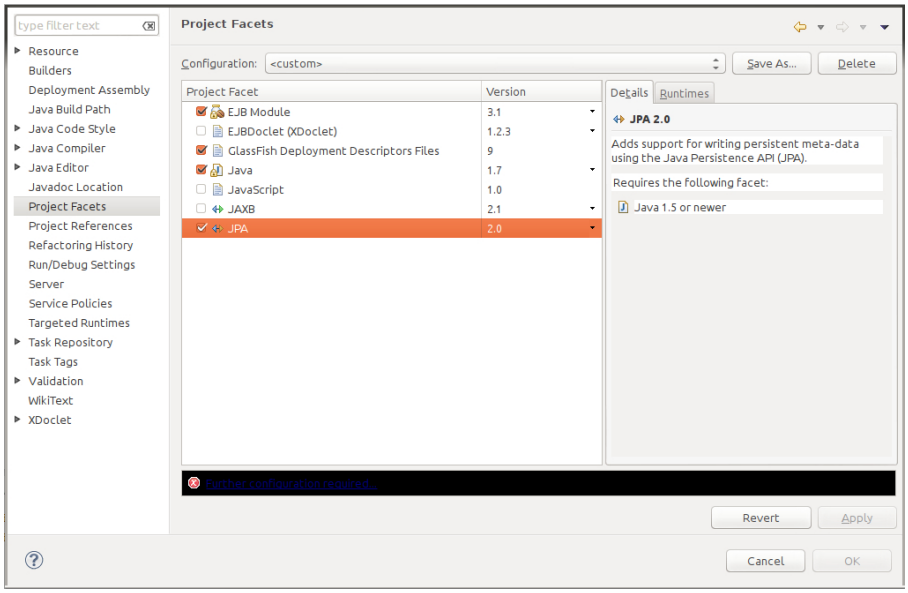


Figura 4: Panel de propiedades del proyecto `jpaPersonalEJB` mostrando las facetas del proyecto.

## 4.2 Definición de la `PersistenceUnit`

Ahora que nuestro proyecto incorpora la faceta JPA, es el momento de configurar la capa de persistencia. Como ya sabemos, la capa de persistencia de obtiene a través de una `PersistenceManagerFactory` que se configura mediante una `PersistenceUnit`. La `PersistenceUnit` se define en el fichero `persistence.xml` que se puede editar expandiendo el árbol **JPA Content** y seleccionando el icono del fichero. Editamos y configuramos el fichero del siguiente modo:

1. **Solapa General**. En la entrada `Name:` ponemos el valor `personalPersistenceUnit`. Este es el nombre de nuestra `PersistenceUnit` o de nuestro `PersistenceContext`.
2. **Solapa Connection**. En la entrada `JTA data source:` utilizamos el valor `jdbc/personalpool`, que es el nombre que utiliza nuestro servidor Glassfish para identificar el *pool* de conexiones a la base de datos `Personal`.

Finalmente, recuerde guardar los cambios realizados en el fichero antes de cerrarlo.

## 4.3 Importación de las entidades desde la base de datos

En una aplicación JPA el ORM se realiza automáticamente a través del sistema de persistencia. El procedimiento que vamos a seguir es el siguiente:

1. Seleccionamos el proyecto `jpaPersonalEJB` con el botón de la derecha y en el menú emergente seleccionamos **JPA Tools** → **Generate Entities from Tables** .
2. En el panel de configuración seleccionamos `Connection: personal` y `Schema: personal`. En el área `Tables` se mostrarán todas las tablas que tiene nuestra base de datos.
3. Seleccionamos sólo las tablas `departamentos` y `empleados` como se muestra en la figura 6.
4. Tras pulsar sobre el botón **Next >** , nos encontramos en el panel de configuración de las relaciones.
   El generador de entidades de `EclipseLink` procederá ahora a descubrir e implementar las relaciones entre los objetos seleccionados. Es una buena práctica definir los nombres con los que queremos identificar los extremos de las relaciones, ya que JPA desconoce qué *rol* tiene cada uno de los extremos de la relación e "inventa" el nombre de las variables que lo implementan.
   A continuación describiremos y configuraremos cada una de las relaciones entre `Empleados` y `Departamentos` de nuestro modelo.
5. `Many-to-One unidireccional`: Un empleado puede ser *manager* de uno o más departamentos y cada departamento cuenta con un único mánager. El empleado actúa en el rol de `manager`. En este caso hemos decidido que la relación es unidireccional y que sólo vamos a recorrerla en la dirección `Departamento` a `Empleado`. Esta operación se realiza como se muestra en la figura 7.

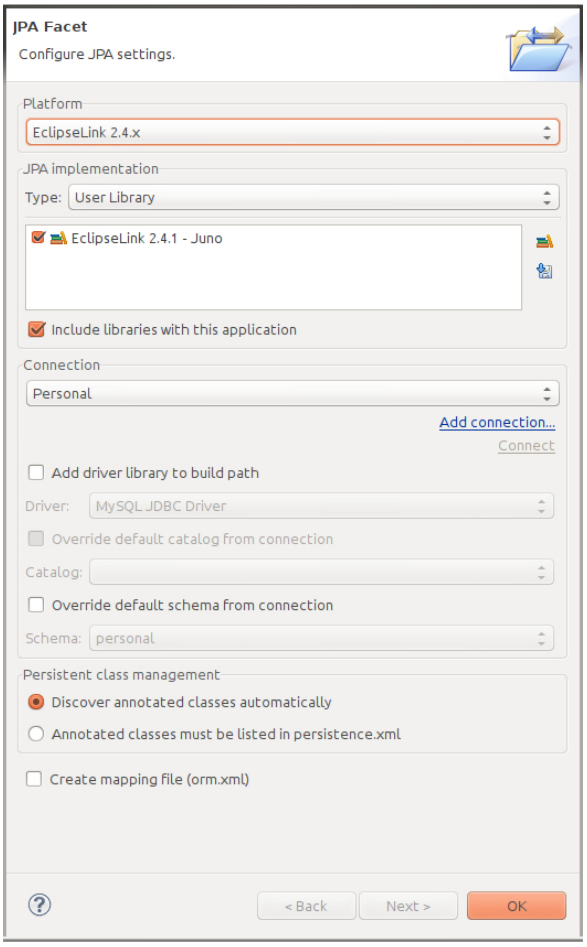


Figura 5: Panel de configuración de la faceta `Java Persistence` del proyecto `jpaPersonalEJB`.

6. `One-to-Many`: Un departamento cuenta con un conjunto de empleados y cada empleado pertenece a un departamento. Los roles en este caso son triviales y se denominan `departamento` (extremo uno) y `empleados` (extemo muchos). Ver figura 8.
7. Una vez completada la definición de las relaciones, pulsamos el botón **Next >** . En el nuevo panel definimos la forma en que vamos a gestionar la identidad de los objetos. En este caso hemos decidido aprovechar que en nuestro modelo la clave primaria se calcula automáticamente cuando se inserta una tupla, por lo que seleccionamos `Identity` en el campo `Key generator`. También decidimos cual es el paquete de destino de las entidades (`sssi.tasi.personal.entity`) como se muestra en la figura 9.

**Aviso:** En este último panel, no olvide marcar el radio-button `Eager` en la entrada `Associations fetch`.

**Nota:** Observe que en este caso hemos decidido prescindir de los *data transfer objects* y utilizar sólo *entidades*. Aunque conceptualmente son objetos similares, en ocasiones es útil mantener ambos tipos de objetos. Esto es especialmente cierto cuando no deseamos que los clientes (los *servlets*) tengan acceso completo a los objetos que persisten en la base de datos (las *entidades*) porque, por ejemplo, contienen información que debe restringirse al cliente.

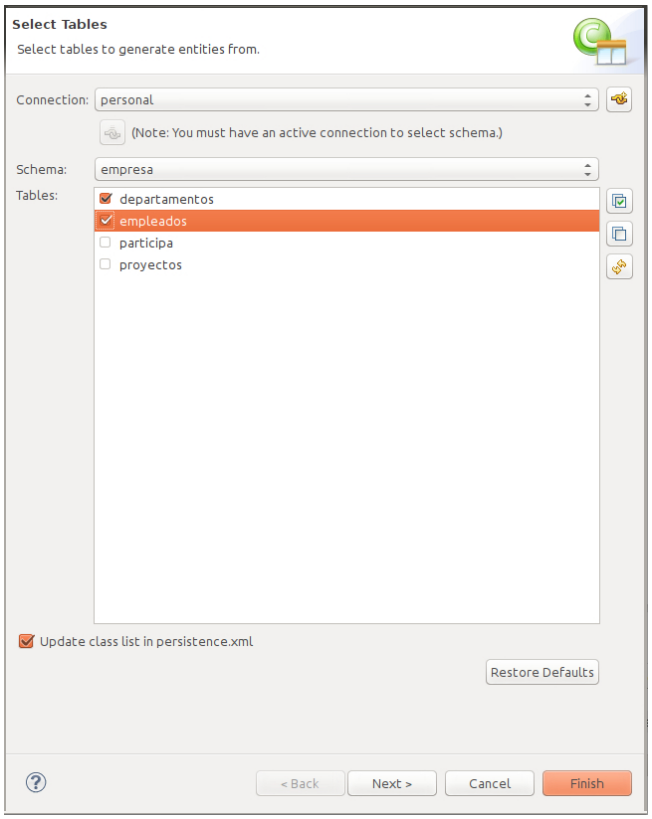


Figura 6: Panel de selección de tablas para la proyección ORM desde tablas a entidades Java.

## 4.4 La interface `Dao`

La interface `Dao` es una interface genérica, independiente del tipo de Dao a implementar (Hibernate, JPA, etc...), que define el conjunto de funciones básicas que queremos que implemente un DAO.

El código de la interface se muestra en la figura 1. El primer parámetro `K` es el tipo de la clave primaria y el segundo parámetro, `E`, es del tipo de la `Entity` o del `DTO`. Además de las funciones básicas (métodos `persist`, `remove`, y `findById`) es posible añadir más métodos dependiendo de nuestra aplicación.

Para crear la interface pulsamos con el botón derecho sobre el ícono **jpaPersonalEJB** del **Project Explorer** y en el menú contextual seleccionamos **New** → **Interface** . En la línea seleccionaremos el paquete `sssi.tasi.personal.dao`.

Listado 1: Código de la interface `Dao.java`.

```
1 package sssi.tasi.personal.dao;
2
```

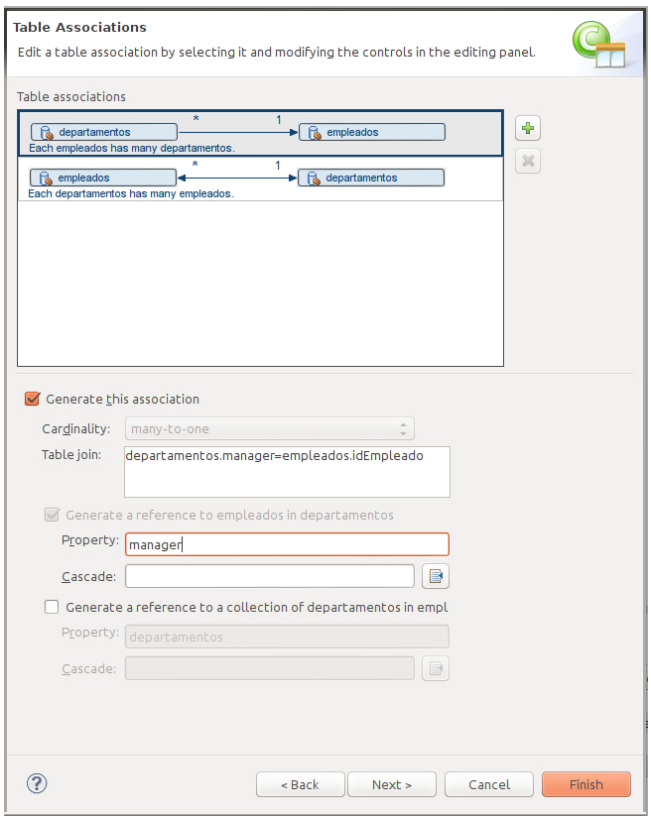


Figura 7: Configuración de la relación unidireccional entre un empleado *manager* y los departamentos que gestiona.

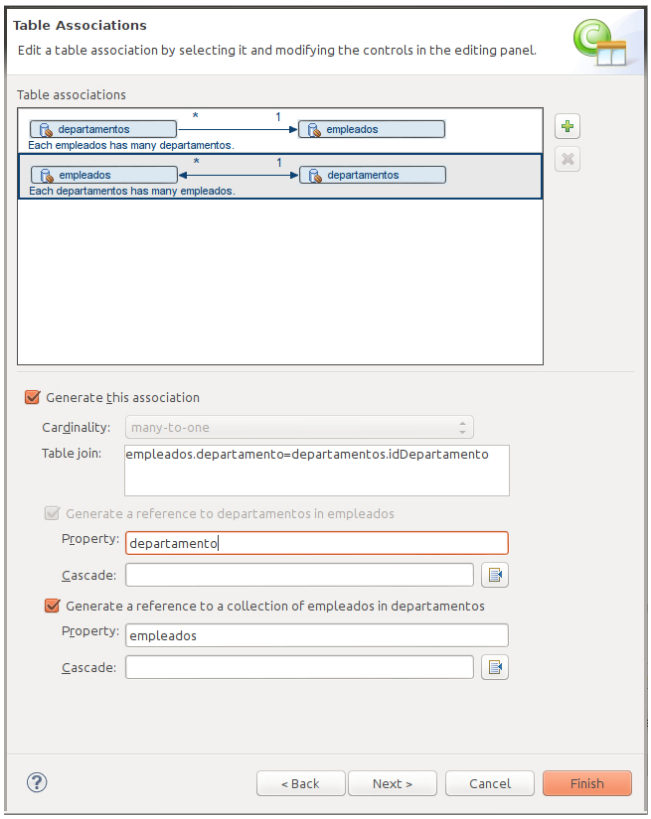


Figura 8: Configuración de la relación entre un departamento y la lista de sus empleados.

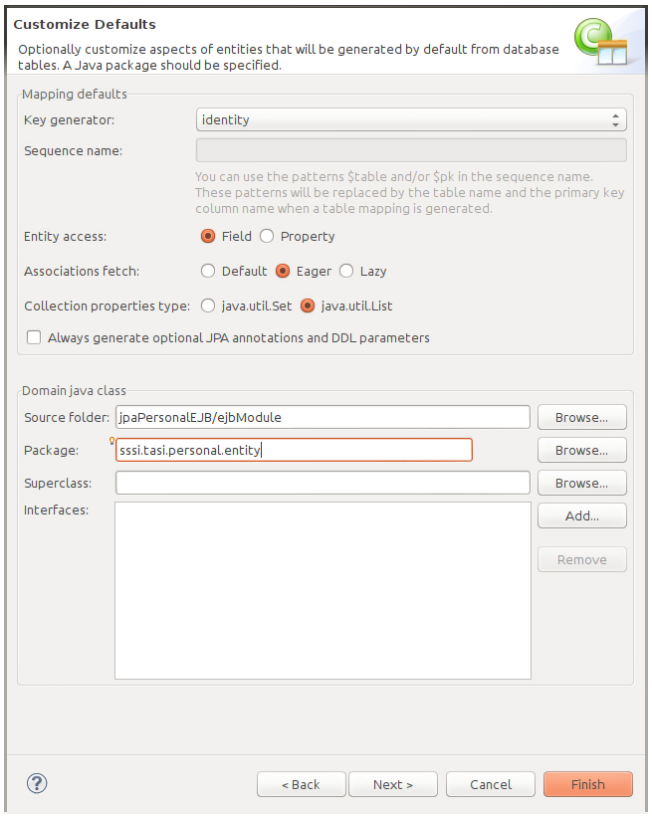


Figura 9: Último panel de creación de las entidades.

```
3 public interface Dao<K, E> {
4   void persist(E entity);
5   void remove(E entity);
6   E findById(K id);
7 }
```

## 4.5 La clase `JpaDao`

Ahora crearemos una implementación base para el DAO JPA. Esta clase contendrá una implementación básica de todos los métodos de la interface Dao estándar que creamos en el punto anterior.

El código es en general bastante sencillo, sin embargo hay un par de cuestiones que es necesario destacar:

- El constructor `JpaDao` utiliza un conjunto de instrucciones para determinar la clase concreta de la entidad `E`. El significado exacto de estas líneas queda fuera del ámbito de este curso.
- El constructor recibe como parámetro un `EntityManager` que será proporcionado por el EJB que haga uso del DAO.
- Ambos atributos (`entityClass` y `em`) son declarados `protected` de modo que las implementaciones específicas del DAO (las subclases) puedan accederlos.

Para crear la clase pulsamos con el botón derecho sobre el ícono **jpaPersonalEJB** del **Project Explorer** y en el menú contextual seleccionamos **New** → **Class** . Configure el panel del siguiente modo (vea la figura 10:

- En `Package:` utilice `sssi.tasi.personal.dao`.
- El nombre de la clase será `JpaDao<K, E>`.
- Añadiremos ( **Add** ) la interface `Dao` en el área `Interfaces:`.

Listado 2: Código de la clase `JpaDao.java`.

```
1 package sssi.tasi.personal.dao;
2
3 import java.lang.reflect.ParameterizedType;
4
5 import javax.persistence.EntityManager;
6
```

```java
public abstract class JpaDao<K, E> implements Dao<K, E> {

  protected Class<E> entityClass;
  protected EntityManager em;

  @SuppressWarnings("unchecked")
  public JpaDao(EntityManager em) {
    ParameterizedType genericSuperclass =
    (ParameterizedType)getClass().getGenericSuperclass();
    this.entityClass = (Class<E>)genericSuperclass.getActualTypeArguments()[1];
    this.em = em;
  }
  @Override
  public E findById(K id) {
    E entity;
    entity = em.find(entityClass, id);
    return entity;
  }

  @Override
  public void persist(E entity) {
    em.persist(entity);
    em.flush();
  }
  @Override
  public void remove(E entity) {
    em.remove(entity);
    em.flush();
  }
}
```

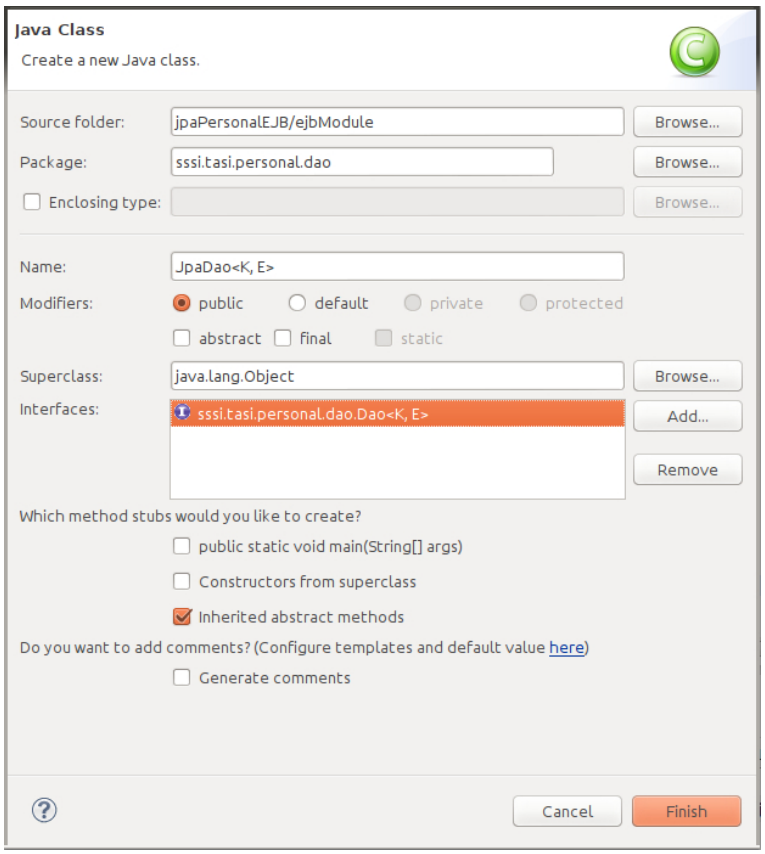


Figura 10: Panel de creación de la clase `JpaDao<K, E>`.

## 4.6 El DAO de la clase `Empleado`

Ahora crearemos la clase `EmpleadoDao`. Esta clase es una implementación específica de la implementación DAO y por tanto extiende la clase `JpaDao` básica e implementa la interface `Dao` específica.

Antes de crear este DAO, vamos a modificar ligeramente la entidad `Empleado.java` añadiendo una serie de anotaciones que definen las consultas JDOQL que vamos a utilizar en esta clase. Para ello, editamos el fichero `sssi.tasi.personal.entity.Empleado.java` y añadimos el siguiente bloque de código justo antes de la definición de la clase:

```
@NamedQueries({
  @NamedQuery(
    name = "Empleado.findAll",
    query = "SELECT e " +
            "  FROM Empleado e " +
            " ORDER BY e.apellidos"),
  @NamedQuery(
    name = "Empleado.findByName",
    query = "SELECT e FROM Empleado e " +
            " WHERE e.apellidos LIKE :apellidos " +
            " ORDER BY e.apellidos")
})
```

**Nota:** Cuidado con este bloque de código. Si por algún motivo es necesario volver a importar las entidades a partir de las tablas, este bloque se perderá al reescribir el código de la clase por lo que será necesario volver a modificar el fichero `Empleado.java`.
Podría parecer que este no es el punto más adecuado para añadir la anotación con las consultas con nombre. Sin embargo, es el único que permite la especificación de JPA 2.0. Otra alternativa válida es definir las consultas con nombre en un fichero xml dentro del proyecto.

Ahora pasamos a crear la clase pulsando con el botón derecho sobre el ícono **jpaPersonalEJB** del **Project Explorer** y seleccionando en el menú contextual **New** → **Class** . La clase `EmpleadoDAO` se implementa en el mismo paquete que el resto de las clases. En el campo `Superclass:` utilice el botón **Browse...** para indicar que nuestra clase deriva de la clase `JpaDao`.

El código de la clase `EmpleadoDao` se muestra en el listado 3. La clase hereda de `JpaDao` los métodos genéricos allí definidos y además implementa tres métodos nuevos:

- El constructor `EmpleadoDao`, que recibe como parámetro un `EntityManager` y que se limita a pasarlo al constructor de la superclase `JpaDao` mediante la sentencia `super(em)`.
- El método `getAllEmpleados` que obtiene y devuelve una lista con todos los empleados de la base de datos.
- El método `findEmpleadosByApellidos` que obtiene la lista (posiblemente vacía) con todos los empleados cuyos apellidos empiezan por la cadena pasada como parámetro.

Listado 3: Código de la clase `EmpleadoDao.java`.

```
package sssi.tasi.personal.dao;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.TypedQuery;

import sssi.tasi.personal.entity.Empleado;

public class EmpleadoDao extends JpaDao<Integer, Empleado> {

  public EmpleadoDao(EntityManager em) {
    super(em);
  }

  public List<Empleado> getAllEmpleados() {
    TypedQuery<Empleado> query = em.createNamedQuery(
      "Empleado.findAll",
      Empleado.class
    );
    return query.getResultList();
```

```
22    }
23
24    public List<Empleado> findEmpleadoByName(String apellidos) {
25      TypedQuery<Empleado> query = em.createNamedQuery(
26        "Empleado.findByName",
27        Empleado.class
28      );
29      query.setParameter("apellidos", apellidos);
30      return query.getResultList();
31    }
32  }
```

## 4.7 El DAO de la clase `Departamento`

Siguiendo un procedimiento similar al descrito en el apartado anterior, vamos a crear el DAO local de la clase `Departamento`.

Previamente modificaremos el fichero `Departamento.java` para añadir las anotaciones `@NamedQuery` que definen las consultas sobre la clase:

```
1  @NamedQuery(
2    name = "Departamento.findAll",
3    query = "SELECT d " +
4            "  FROM Departamento d " +
5            " ORDER BY d.nombre")
```

Como en el caso anterior, la clase `DepartamentoDao` extenderá de la clase `JpaDao` e implementará la interface `Dao`. El código de la clase se muestra en el listado 4. En este caso la clase sólo añade el método `getAllDepartamentos`.

Listado 4: Código de la clase `DepartamentoDao.java`.

```
1  package sssi.tasi.personal.dao;
2
3  import java.util.List;
4
5  import javax.persistence.EntityManager;
6  import javax.persistence.TypedQuery;
7
8  import sssi.tasi.personal.entity.Departamento;
9
10 public class DepartamentoDao extends JpaDao<Integer, Departamento> {
11
12   public DepartamentoDao(EntityManager em) {
13     super(em);
14   }
15
16   public List<Departamento> getAllDepartamentos() {
17     TypedQuery<Departamento> query = em.createNamedQuery(
18       "Departamento.findAll",
19       Departamento.class
20     );
21     return query.getResultList();
22   }
23 }
```

## 4.8 La clase de utilidad `FinalString`

Como en el ejercicio anterior, la clase `FinalString` es una pequeña clase de utilidad que utilizarán los *Bussines Objects* y cuya misión es dar formato a los `String` que se recogen a través de los formularios html. `FinalString` hace un tratamiento muy simple de las cadenas de caracteres:

- Elimina los caracteres en blanco al principio y final de línea.
- Consolida cualquier combinación de uno o más espacios en blanco en un sólo blanco.
- Convierte los caracteres de la cadena a mayúsculas.

Esta clase se creará dentro del paquete `sssi.tasi.personal.util`, cuenta con un único método *static* `arregla` y su código se muestra en el listado 5

Listado 5: Código de la clase de utilidad `FinalString.java`.

```
1  package sssi.tasi.personal.util;
2
3  public class FinalString {
4    public static String arregla(String s) {
5      String n;
6      n = s.trim().toUpperCase();
7      n = n.replaceAll("( )+", " ");
8      return n;
9    }
10 }
```

## 4.9 El EJB `EmpleadoBO`

Del mismo modo que los DAOs encapsulan el acceso a la base de datos, los *Business Objects* encapsulan las reglas de negocio. En nuestro caso, el EJB `EmpleadoBo` se encarga, por ejemplo, del "negocio" de suministrar la lista de `Empleado` que utilizará nuestro *servlet*. El EJB presentará una interface remota, de modo que responderá a las invocaciones externas al servidor de aplicaciones que realizará el *servlet*.

Dentro del proyecto `jpaPersonalEJB` crearemos el paquete `sssi.tasi.personal.bo` y en él crearemos el EJB `EmpleadoBo` como se muestra en la figura 11. La interface `EmpleadoBoRemote`, que se muestra en el listado 6, define cuatro métodos:

- Un método `newEmpleado` que creará y hará persistente un objeto de la clase `Empleado`.
- Un método para obtener la lista de todos los empleados de la base de datos: `listaEmpleados`.
- Dos métodos de búsqueda: `findEmpleado` y `buscarEmpleado`.
- El método de utilidad `keysEmpleado` que utilizaremos para construir las listas desplegables de nuestra pequeña aplicación web.

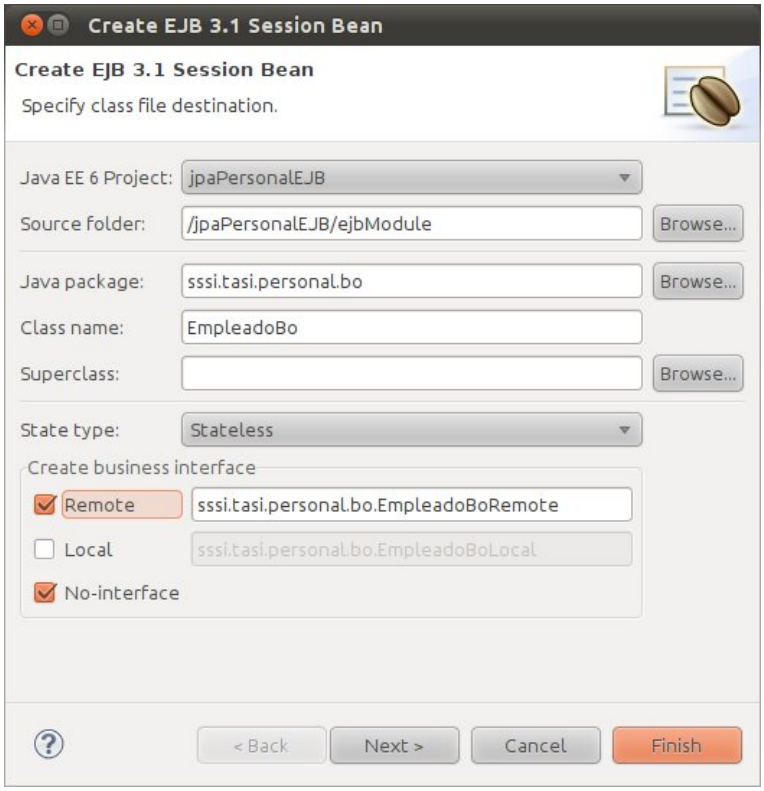


Figura 11: Creación del EJB stateless `EmpleadoBo` y de la interface remota `EmpleadoBoRemote`.

Listado 6: Código de la interface `EmpleadoBoRemote.java`.

```
1  package sssi.tasi.personal.bo;
```

```
2
3  import java.math.BigDecimal;
4  import java.util.Date;
5  import java.util.List;
6  import java.util.TreeMap;
7
8  import javax.ejb.Remote;
9
10 import sssi.tasi.personal.entity.Empleado;
11
12 @Remote
13 public interface EmpleadoBoRemote {
14   List<Empleado> listaEmpleados();
15   public Empleado findEmpleado(int id);
16   public List<Empleado> buscarEmpleado(String apellidos);
17   public void newEmpleado(String nombre, String apellidos, String puesto,
18       Date date, Short nivelEducacion, BigDecimal sueldo,
19       BigDecimal complemento, int depto);
20   public TreeMap<String, Integer> keysEmpleado();
21 }
```

En el listado 7 se muestra el código del EJB `EmpleadoBp` que implenta los métodos definidos en la interfaz.

Listado 7: Código de la clase `EmpleadoBo.java`.

```
1  package sssi.tasi.personal.bo;
2
3  import java.math.BigDecimal;
4  import java.util.Date;
5  import java.util.List;
6  import java.util.TreeMap;
7
8  import javax.annotation.PostConstruct;
9  import javax.annotation.PreDestroy;
10 import javax.ejb.Stateless;
11 import javax.persistence.EntityManager;
12 import javax.persistence.PersistenceContext;
13
14 import sssi.tasi.personal.dao.DepartamentoDao;
15 import sssi.tasi.personal.dao.EmpleadoDao;
16 import sssi.tasi.personal.entity.Departamento;
17 import sssi.tasi.personal.entity.Empleado;
18 import sssi.tasi.personal.util.FinalString;
19
20 /**
21 * Session Bean implementation class EmpleadoBo
22 */
23 @Stateless
24 public class EmpleadoBo implements EmpleadoBoRemote {
25
26   @PersistenceContext(name = "personalPersistenceUnit")
27   private EntityManager em;
28   private EmpleadoDao edao;
29   private DepartamentoDao ddao;
30
31   @PostConstruct
32   public void init()  {
33     edao = new EmpleadoDao(em);
34     ddao = new DepartamentoDao(em);
35   }
36
37   @PreDestroy
38   public void finaliza() {
39     em.close();
40   }
41
42   /**
43   * Default constructor.
```

```
44    */
45    public EmpleadoBo() {}
46
47    @Override
48    public List<Empleado> listaEmpleados() {
49      List<Empleado> empleados = edao.getAllEmpleados();
50      return empleados;
51    }
52
53    @Override
54    public Empleado findEmpleado(int id) {
55      return edao.findById(id);
56    }
57
58    @Override
59    public List<Empleado> buscarEmpleado(String apellidos) {
60      // Adaptamos el string y concatenamos el comodin al final
61      apellidos = FinalString.arregla(apellidos) + "%";
62      return edao.findEmpleadoByName(apellidos);
63    }
64
65    @Override
66    public void newEmpleado(String nombre, String apellidos, String
67        puesto, Date date, Short nivelEducacion, BigDecimal sueldo,
68        BigDecimal complemento, int depto) {
69
70      Empleado nuevo = new Empleado();
71      nuevo.setNombre(FinalString.arregla(nombre));
72      nuevo.setApellidos(FinalString.arregla(apellidos));
73      nuevo.setPuesto(puesto);
74      nuevo.setFechaContrato(date);
75      nuevo.setNivelEducacion(nivelEducacion);
76      nuevo.setSueldo(sueldo);
77      nuevo.setComplemento(complemento);
78      /*
79      * Localizamos el departamento.
80      */
81      Departamento departamento= ddao.findById(depto);
82      nuevo.setDepartamento(departamento);
83      departamento.getEmpleados().add(nuevo);
84
85      // Salvamos el nuevo empleado
86      edao.persist(nuevo);
87    }
88
89    @Override
90    public TreeMap<String, Integer> keysEmpleado() {
91      String nomCompleto;
92      TreeMap<String, Integer> keyEmp = new TreeMap<String, Integer>();
93      List<Empleado> empleados = edao.getAllEmpleados();
94      for (Empleado emp : empleados) {
95        nomCompleto = emp.getApellidos() + ", " + emp.getNombre();
96        keyEmp.put(nomCompleto, emp.getIdEmpleado());
97      }
98      return keyEmp;
99    }
100 }
```

**Nota:** Observe con la anotación `@PersistenceUnit(name = "personalPersistenceUnit")` se solicita del servidor de aplicaciones la inyección de una instancia de la clase `EntityManagerFactory`. A partir de esta instancia se construye el `EntityManager` que pasaremos como parámetro para la creación del DAO `EmpleadoDao`. Observe como se hace uso de los métodos etiquetados como `@PostConstruct` y `@PreDestroy` para realizar estas tareas de inicialización.

## 4.10 El EJB `DepartamentoBO`

Ahora crearemos el EJB `DepartamentoBo`, que se encargará de las reglas de "negocio" que implican departamentos. En este caso la interface `DepartamentoBoRemote` define cuatro métodos:

- El método `listaDepartamentos` que devuelve la lista de departamentos almacenados en la base de datos.
- `findDepartamento`, que obtiene un departamento a partir de su `id`.
- El método `newDepartamento` que construye y hace persistente un objeto departamento a partir de los parámetros suministrados.
- El método `findEmpleadosByDepto`, que devuelve la lista de todos los empleados adscritos a un departamento. En la práctica anterior, este método estaba definido en el BO de la clase Empleado. En esta práctica se ha decidido implementarlo en `DepartamentoBo` por simplicidad.
- El método de utilidad `keysDepartamento`, que como en el caso anterior nos servirá de apoyo para la construcción de los desplegables de nuestra aplicación.

El código de la interface se muestra en el listado 8 y el código del EJB que la implementa en el listado 9.

Listado 8: Código de la interface `DepartamentoBoRemote.java`.

```
1  package sssi.tasi.personal.bo;
2
3  import java.util.List;
4  import java.util.TreeMap;
5  import javax.ejb.Remote;
6
7  import sssi.tasi.personal.entity.Departamento;
8  import sssi.tasi.personal.entity.Empleado;
9
10 @Remote
11 public interface DepartamentoBoRemote {
12   public List<Departamento> listaDepartamentos();
13   public Departamento findDepartamento(int id);
14   public void newDepartamento(String nombre, int manager);
15   public TreeMap<String, Integer> keysDepartamento();
16   public List<Empleado> findEmpleadosByDepto(int id);
17 }
```

Listado 9: Código de la clase `DepartamentoBo.java`.

```
1  package sssi.tasi.personal.bo;
2
3  import java.util.ArrayList;
4  import java.util.List;
5  import java.util.TreeMap;
6  import javax.annotation.PostConstruct;
7  import javax.annotation.PreDestroy;
8  import javax.ejb.Stateless;
9  import javax.persistence.EntityManager;
10 import javax.persistence.EntityManagerFactory;
11 import javax.persistence.PersistenceUnit;
12
13 import sssi.tasi.personal.dao.DepartamentoDao;
14 import sssi.tasi.personal.dao.EmpleadoDao;
15 import sssi.tasi.personal.entity.Departamento;
16 import sssi.tasi.personal.entity.Empleado;
17 import sssi.tasi.personal.util.FinalString;
18
19 /**
20 * Session Bean implementation class DepartamentoBo
21 */
22 @Stateless
23 public class DepartamentoBo implements DepartamentoBoRemote {
24
25   @PersistenceUnit(name = "personalPersistenceUnit")
26   private EntityManagerFactory emf;
27   private EntityManager em;
28   private DepartamentoDao ddao;
29   private EmpleadoDao edao;
```

```
30
31  @PostConstruct
32  public void init()  {
33    em = emf.createEntityManager();
34    ddao = new DepartamentoDao(em);
35    edao = new EmpleadoDao(em);
36  }
37
38  @PreDestroy
39  public void finaliza() {
40    em.close();
41  }
42
43  public DepartamentoBo() {}
44
45  @Override
46  public List<Departamento> listaDepartamentos() {
47    List<Departamento> departamentos = ddao.getAllDepartamentos();
48    return departamentos;
49  }
50
51  @Override
52  public Departamento findDepartamento(int id) {
53    return ddao.findById(id);
54  }
55
56  @Override
57  public TreeMap<String, Integer> keysDepartamento() {
58    TreeMap<String, Integer> keyDepto = new TreeMap<String, Integer>();
59
60    List<Departamento> departamentos = ddao.getAllDepartamentos();
61    for (Departamento depto : departamentos) {
62      keyDepto.put(depto.getNombre(), depto.getIdDepartamento());
63    }
64    return keyDepto;
65  }
66
67  @Override
68  public void newDepartamento(String nombre, int m) {
69    Departamento nuevo = new Departamento();
70    nuevo.setNombre(FinalString.arregla(nombre));
71    /*
72    * Buscamos el manager del departamento y lo asignamos
73    */
74    Empleado manager = edao.findById(m);
75    nuevo.setManager(manager);
76    /*
77    * Creamos el conjunto de empleados del departamento.
78    */
79    List<Empleado> empleados = new ArrayList<Empleado>();
80    nuevo.setEmpleados(empleados);
81    /*
82    * Damos de baja al empleado de su antiguo departamentos
83    */
84    manager.getDepartamento().getEmpleados().remove(manager);
85    /*
86    * Asignamos el manager al nuevo departamento. Es necesario
87    * actualizar ambos extremos de la asociacion.
88    */
89    empleados.add(manager);
90    manager.setDepartamento(nuevo);
91    /*
92    * Guardamos el nuevo departamento
93    */
94    ddao.persist(nuevo);
95  }
96
```

```
97    @Override
98    public List<Empleado> findEmpleadosByDepto(int id) {
99      List<Empleado> empleados = new ArrayList<Empleado>();
100
101     Departamento depto = ddao.findById(id);
102     for (Empleado emp : depto.getEmpleados()) {
103       empleados.add(emp);
104     }
105     return empleados;
106   }
107 }
```

**Nota:** Analice atentamente el código del método `newDepartamento` de la clase `DepartamentoBo`. Este método implementa el caso de uso "crear departamento" y por tanto se ocupa de llevar a cabo todas las operaciones de la lógica de negocio, que en este caso implican:

- Crear el nuevo departamento:

```
1     Departamento nuevo = new Departamento();
2     nuevo.setNombre(FinalString.arregla(nombre));
```

- Localizar el *manager*, recuperarlo de la base de datos y asignarlo como mánager del nuevo departamento:

```
1     Empleado manager = edao.findById(m);
2     nuevo.setManager(manager);
```

- Dar de baja al *manager* de su antiguo departamento:

```
1     manager.getDepartamento().getEmpleados().remove(manager);
```

- Añadir al *manager* a la lista de empleados del departamentos y al revés (indicar que el departamento al que pertenece el mánager es ahora el nuevo departamento). Observe que es necesario actualizar ambos extremos de la relación:

```
1     empleados.add(manager);
2     manager.setDepartamento(nuevo);
```

# 5 Creación de la aplicación web dinámica

La última fase del desarrollo del programa consiste en crear una aplicación web dinámica que interactuará con los *Enterprise Java Beans* `EmpleadoBo` y `DepartamentoBo` a través de varios *servlets* para implementar un conjunto mínimo de casos de uso:

- **Empleados**: Obtener la lista de empleados, añadir un nuevo empleado y buscar empleados por apellidos.
- **Departamentos**: Obtener la lista de departamentos y añadir un nuevo departamento.

Para crear el proyecto, en el menú menfFile de Eclipse seleccionamos **New** → **Dynamic Web Project** y creamos el proyecto `jpaPersonalWA` (de modo similar a como se explicó en el ejercicio anterior).

Una vez creado el proyecto, realizamos las siguientes operaciones:

1. Para que nuestro proyecto tenga acceso a los DTOs y los BOs, añadimos el proyecto `jpaPerosnalEJB` al *Build Path* de nuestro proyecto web a través del menú contextual opción **Preferencias** .
2. Descomprima los ficheros suministrados en el fichero comprimido `boletin_t5b5_ficheros_WA.zip` que acompaña a la práctica. Copie el contenido de la carpeta `WebContent` en su proyecto:

```
1     cp -r WebContent/* ~/workspace/jpaPersonalWA/WebContent
```

y refresque el directorio de Eclipse (tecla **F5** ).

## 5.1 El módulo de gestión de empleados

En primer lugar, crearemos el paquete `sssi.tasi.personal.servlet` en el que implementaremos los servlet. Dentro del paquete crearemos los diferentes *servlets* que implementan los casos de uso de la gestión de empleados y a los que se podrá acceder desde el menú principal de la aplicación. El menú asociado a los empleados contempla los siguientes casos de uso:

- Mostrar la lista de todos los empleados.
- Añadir un nuevo empleado, para lo que presenta un formulario solicitando la información del nuevo empleado.
- Buscar empleado por apellidos.
- Mostrar la lista de empleados adscritos a un departamento.

### 5.1.1 Mostrar la lista de todos los empleados

En el diagrama 12 se representa el diagrama de secuencia de este caso de uso. En el listado 10 se proporciona el código del servlet que coordina la acción.

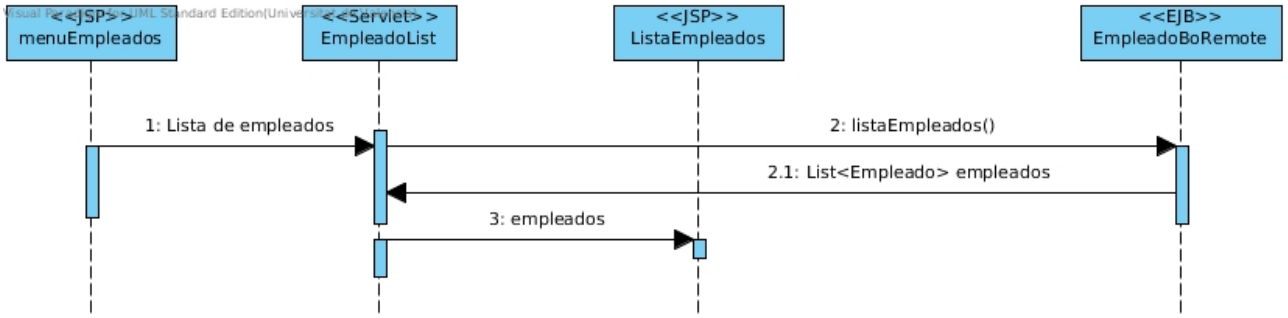


Figura 12: Diagrama de secuencia del caso de uso que obtiene la lista de todos los empleados.

Listado 10: Código del *servlet* `EmpleadoList.java`.

```
1  package sssi.tasi.personal.servlet;
2
3  import java.io.IOException;
4  import java.util.List;
5
6  import javax.ejb.EJB;
7  import javax.servlet.ServletException;
8  import javax.servlet.annotation.WebServlet;
9  import javax.servlet.http.HttpServlet;
10 import javax.servlet.http.HttpServletRequest;
11 import javax.servlet.http.HttpServletResponse;
12
13 import sssi.tasi.personal.bo.EmpleadoBoRemote;
14 import sssi.tasi.personal.entity.Empleado;
15
16 /**
17 * Servlet implementation class EmpleadoList
18 */
19 @WebServlet("/EmpleadoList")
20 public class EmpleadoList extends HttpServlet {
21   private static final long serialVersionUID = 1L;
22   @EJB
23   private EmpleadoBoRemote empleadoBO;
24
25   /**
26   * @see HttpServlet#HttpServlet()
27   */
28   public EmpleadoList() {
29     super();
30   }
31
32   protected void processRequest(HttpServletRequest request, HttpServletResponse
         response) throws ServletException, IOException {
33     List<Empleado> empleados;
34
35     empleados = empleadoBO.listaEmpleados();
36     request.setAttribute("empleados", empleados);
37     request.getRequestDispatcher("/empleados/listaEmpleados.jsp").forward(request
           ,  response);
38   }
```

```
39
40    /**
41     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse
          response)
42     */
43    protected void doGet(HttpServletRequest request, HttpServletResponse response)
          throws ServletException, IOException {
44      processRequest(request, response);
45    }
46
47    /**
48     * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse
          response)
49     */
50    protected void doPost(HttpServletRequest request, HttpServletResponse response)
           throws ServletException, IOException {
51      processRequest(request, response);
52    }
53  }
```

### 5.1.2 Añadir un nuevo empleado

La figura 13 muestra el diagrama de secuencia del caso de uso añadir un nuevo empleado. En el listado 11 se proporciona el código del servlet que coordina la acción.

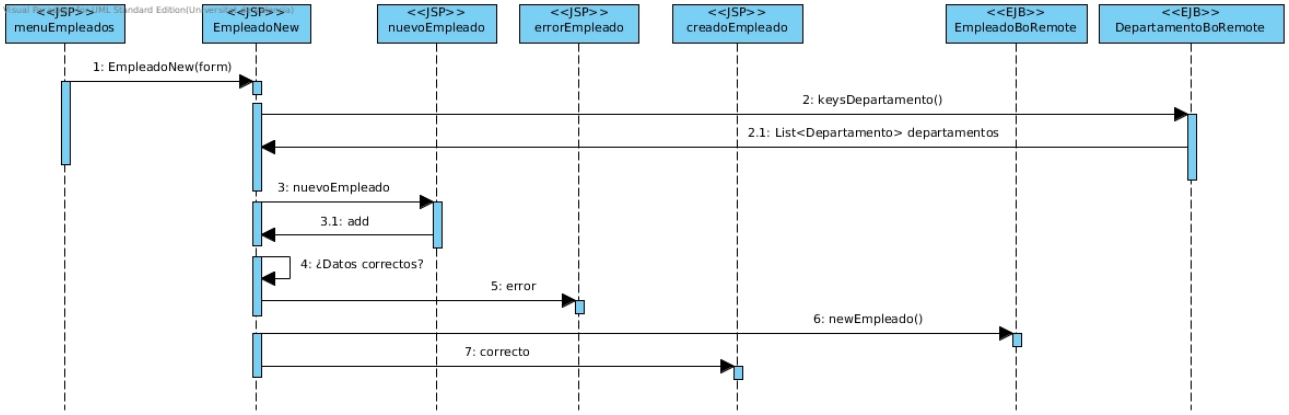


Figura 13: Diagrama de secuencia del caso de uso que obtiene la lista de todos los empleados.

Listado 11: Código del *servlet* `EmpleadoNew.java`.

```
1  package sssi.tasi.capitulo4.servlet;
2
3  import java.io.IOException;
4  import java.sql.Date;
5  import java.util.TreeMap;
6
7  import javax.ejb.EJB;
8  import javax.servlet.ServletException;
9  import javax.servlet.annotation.WebServlet;
10 import javax.servlet.http.HttpServlet;
11 import javax.servlet.http.HttpServletRequest;
12 import javax.servlet.http.HttpServletResponse;
13
14 import sssi.tasi.capitulo4.bo.DepartamentoBoRemote;
15 import sssi.tasi.capitulo4.bo.EmpleadoBoRemote;
16
17 /**
18 * Servlet implementation class EmpleadoNew
19 */
20 @WebServlet("/EmpleadoNew")
21 public class EmpleadoNew extends HttpServlet {
```

```
22  private static final long serialVersionUID = 1L;
23  @EJB(mappedName = "EmpleadoBo")
24  private EmpleadoBoRemote empleadoBO;
25  @EJB(mappedName = "DepartamentoBo")
26  private DepartamentoBoRemote departamentoBO;
27
28  /**
29  * @see HttpServlet#HttpServlet()
30  */
31  public EmpleadoNew() {
32    super();
33  }
34
35  protected void processRequest(HttpServletRequest request, HttpServletResponse
        response)
36  throws ServletException, IOException {
37    String nombre, apellidos, puesto, ano, mes, dia;
38    Date date;
39    Short nivelEd = 0;
40    float sueldo = 0, complemento = 0;
41    int depto = 0;
42    TreeMap<String, Integer> departamentos;
43    Boolean error = false;
44
45    String accion = request.getParameter("action");
46
47    switch (accion) {
48
49        case "form":
50        departamentos = departamentoBO.keysDepartamento();
51        request.setAttribute("departamentos", departamentos);
52        request.getRequestDispatcher("/empleados/nuevoEmpleado.jsp").forward(
              request, response);
53        break;
54
55        case "add":
56        // Recogemos los parametros y comprobamos si son validos
57        nombre = request.getParameter("nombre");
58        apellidos = request.getParameter("apellidos");
59        puesto = request.getParameter("puesto");
60        ano = request.getParameter("ano");
61        mes = request.getParameter("mes");
62        dia =  request.getParameter("dia");
63        date = Date.valueOf(ano + "-" + mes + "-" + dia);
64        try {
65          nivelEd = Short.parseShort(request.getParameter("nivelEducacion"));
66          sueldo = Float.parseFloat(request.getParameter("sueldo"));
67          complemento = Float.parseFloat(request.getParameter("complemento"));
68          depto = Integer.parseInt(request.getParameter("depto"));
69        }
70        catch (NumberFormatException e) {
71          error = true;
72        }
73        if ((nombre == null) ||
74        (apellidos == null) ||
75        (puesto == null) || error) {
76          System.out.println("Error en el empleado...");
77          request.getRequestDispatcher("/empleados/errorEmpleado.jsp").forward(
                request, response);
78        }
79        else {
80          // Construimos y salvamos el nuevo empleado
81          empleadoBO.newEmpleado(nombre, apellidos, puesto, date, nivelEd, sueldo
                , complemento, depto);
82          request.getRequestDispatcher("/empleados/creadoEmpleado.jsp").forward(
                request, response);
83        }
```

```
84          break;
85
86        default:
87          throw new ServletException("óAccin no reconocida o no especificada");
88      }
89    }
90
91    /**
92    * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse
          response)
93    */
94    protected void doGet(HttpServletRequest request, HttpServletResponse response)
95    throws ServletException, IOException {
96      processRequest(request, response);
97    }
98
99    /**
100   * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse
          response)
101   */
102   protected void doPost(HttpServletRequest request, HttpServletResponse response)
103   throws ServletException, IOException {
104     processRequest(request, response);
105   }
106 }
```

### 5.1.3 Buscar empleados por apellidos

La figura 14 muestra el diagrama de secuencia del caso de uso añadir un nuevo empleado. En el listado 12 se proporciona el código del servlet que coordina la acción.

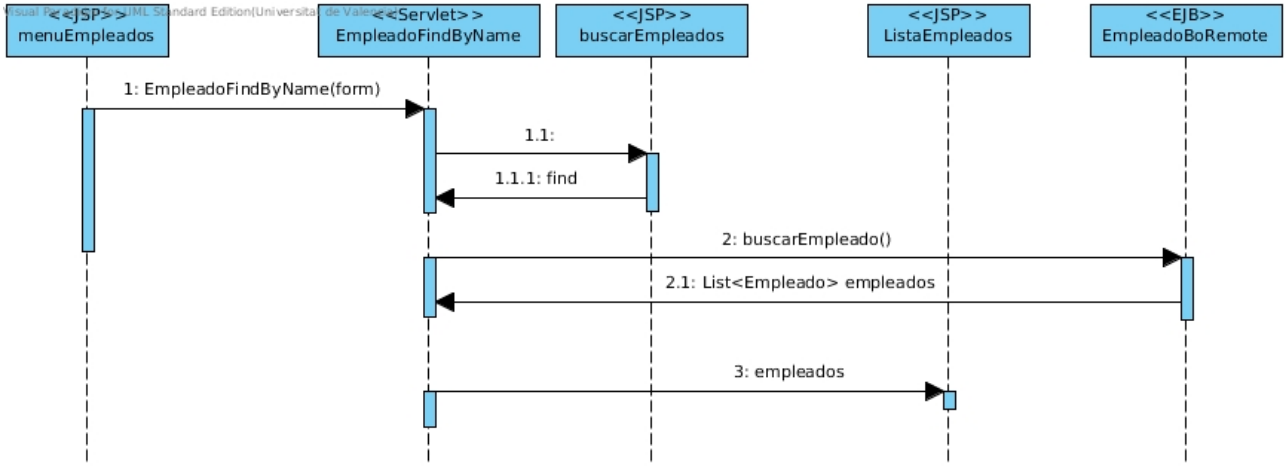


Figura 14: Diagrama de secuencia del caso de que permite buscar empleados por sus apellidos.

Listado 12: Código del *servlet* `EmpleadoFindByName.java`.

```
1  package sssi.tasi.personal.servlet;
2
3  import java.io.IOException;
4  import java.util.List;
5
6  import javax.ejb.EJB;
7  import javax.servlet.ServletException;
8  import javax.servlet.annotation.WebServlet;
9  import javax.servlet.http.HttpServlet;
10 import javax.servlet.http.HttpServletRequest;
11 import javax.servlet.http.HttpServletResponse;
12
13 import sssi.tasi.personal.bo.DepartamentoBoRemote;
```

```
14 import sssi.tasi.personal.bo.EmpleadoBoRemote;
15 import sssi.tasi.personal.entity.Empleado;
16
17 /**
18 * Servlet implementation class EmpleadoFindByName
19 */
20 @WebServlet("/EmpleadoFindByName")
21 public class EmpleadoFindByName extends HttpServlet {
22   private static final long serialVersionUID = 1L;
23   @EJB
24   private EmpleadoBoRemote empleadoBO;
25   @EJB
26   private DepartamentoBoRemote departamentoBO;
27
28   /**
29   * @see HttpServlet#HttpServlet()
30   */
31   public EmpleadoFindByName() {
32     super();
33   }
34   protected void processRequest(HttpServletRequest request, HttpServletResponse
         response) throws ServletException, IOException {
35     List<Empleado> empleados;
36     String apellidos;
37
38     String accion = request.getParameter("action");
39     // Bucle de óseleccin de la óaccin
40     switch (accion) {
41
42         case "form":
43         request.getRequestDispatcher("/empleados/buscarEmpleados.jsp").forward(
               request, response);
44         break;
45
46         case "find":
47         apellidos = request.getParameter("apellidos");
48         empleados = empleadoBO.buscarEmpleado(apellidos);
49         request.setAttribute("empleados", empleados);
50         request.getRequestDispatcher("/empleados/listaEmpleados.jsp").forward(
               request, response);
51         break;
52
53         default:
54         throw new ServletException("óAccin no reconocida o no especificada");
55     }
56   }
57
58   /**
59   * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse
         response)
60   */
61   protected void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {
62     processRequest(request, response);
63   }
64
65   /**
66   * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse
         response)
67   */
68   protected void doPost(HttpServletRequest request, HttpServletResponse response)
          throws ServletException, IOException {
69     processRequest(request, response);
70   }
71 }
```

### 5.1.4 Buscar empleados por departamento

La figura 15 muestra el diagrama de secuencia del caso de uso añadir un nuevo empleado. En el listado 12 se proporciona el código del servlet que coordina la acción.

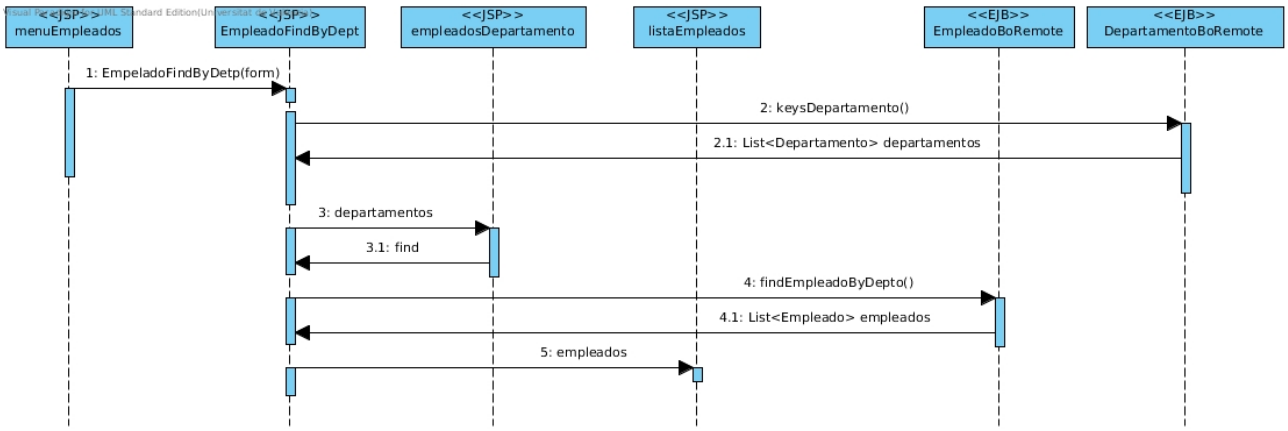


Figura 15: Diagrama de secuencia del caso de que permite buscar empleados adscritos a un departamento.

Listado 13: Código del *servlet* `EmpleadoFindByDept.java`.

```
1  package sssi.tasi.personal.servlet;
2
3  import java.io.IOException;
4  import java.util.List;
5  import java.util.TreeMap;
6
7  import javax.ejb.EJB;
8  import javax.servlet.ServletException;
9  import javax.servlet.annotation.WebServlet;
10 import javax.servlet.http.HttpServlet;
11 import javax.servlet.http.HttpServletRequest;
12 import javax.servlet.http.HttpServletResponse;
13
14 import sssi.tasi.personal.bo.DepartamentoBoRemote;
15 import sssi.tasi.personal.bo.EmpleadoBoRemote;
16 import sssi.tasi.personal.entity.Empleado;
17
18 /**
19 * Servlet implementation class EmpleadoFindByDept
20 */
21 @WebServlet("/EmpleadoFindByDept")
22 public class EmpleadoFindByDept extends HttpServlet {
23   private static final long serialVersionUID = 1L;
24   @EJB
25   private EmpleadoBoRemote empleadoBO;
26   @EJB
27   private DepartamentoBoRemote departamentoBO;
28
29   /**
30   * @see HttpServlet#HttpServlet()
31   */
32   public EmpleadoFindByDept() {
33     super();
34   }
35
36   protected void processRequest(HttpServletRequest request, HttpServletResponse
       response) throws ServletException, IOException {
37     List<Empleado> empleados;
38     int depto = 0;
39     TreeMap<String, Integer> departamentos;
40
41     String accion = request.getParameter("action");
```

```
42      switch (accion) {
43
44          case "form":
45          departamentos = departamentoBO.keysDepartamento();
46          request.setAttribute("departamentos", departamentos);
47          request.getRequestDispatcher("/empleados/empleadosDepartamento.jsp").
                forward(request, response);
48          break;
49
50          case "find":
51          depto = Integer.parseInt(request.getParameter("depto"));
52          empleados = departamentoBO.findEmpleadosByDepto(depto);
53          request.setAttribute("empleados", empleados);
54          request.getRequestDispatcher("/empleados/listaEmpleados.jsp").forward(
                request, response);
55          break;
56
57          default:
58          throw new ServletException("óAccin no reconocida o no especificada");
59      }
60    }
61
62    /**
63    * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse
          response)
64    */
65    protected void doGet(HttpServletRequest request, HttpServletResponse response)
          throws ServletException, IOException {
66      processRequest(request, response);
67    }
68
69    /**
70    * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse
          response)
71    */
72    protected void doPost(HttpServletRequest request, HttpServletResponse response)
           throws ServletException, IOException {
73      processRequest(request, response);
74    }
75  }
```

## 5.2 El módulo de gestión de departamentos

También crearemos los *servlet* necesarios para gestionar los departamentos. Siguiendo el mismo procedimiento explicado en el caso anterior, cree los *servlets* cuyo código se muestra a continuación y cuya misión es gestionar las operaciones que nuestra aplicación realiza sobre los departamentos. El código es bastante más sencillo ya que el número de operaciones implementadas es menor:

- Mostrar la lista de todos los departamentos.
- Añadir un nuevo departamento.

### 5.2.1 Mostrar la lista de todos los departamentos

Este caso de uso es muy similar al caso de uso que muestra la lista de todos los empleados. En el listado 14 se muestra el código del servlet que controla los elementos implicados.

Listado 14: Código del *servlet* `DepartamentoList.java`.

```
1  package sssi.tasi.personal.servlet;
2
3  import java.io.IOException;
4  import java.util.List;
5
6  import javax.ejb.EJB;
7  import javax.servlet.ServletException;
8  import javax.servlet.annotation.WebServlet;
```

```java
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import sssi.tasi.personal.bo.DepartamentoBoRemote;
import sssi.tasi.personal.entity.Departamento;

/**
 * Servlet implementation class DepartamentoList
 */
@WebServlet("/DepartamentoList")
public class DepartamentoList extends HttpServlet {
  private static final long serialVersionUID = 1L;
  @EJB
  private DepartamentoBoRemote departamentoBO;

  /**
   * @see HttpServlet#HttpServlet()
   */
  public DepartamentoList() {
    super();
  }

  protected void processRequest(HttpServletRequest request, HttpServletResponse
      response) throws ServletException, IOException {
    List<Departamento> departamentos;
    departamentos = departamentoBO.listaDepartamentos();
    request.setAttribute("departamentos", departamentos);
    request.getRequestDispatcher("/departamentos/listaDepartamentos.jsp").forward
        (request, response);
  }

  /**
   * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse
      response)
   */
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    processRequest(request, response);
  }

  /**
   * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse
      response)
   */
  protected void doPost(HttpServletRequest request, HttpServletResponse response)
       throws ServletException, IOException {
    processRequest(request, response);
  }
}
```

### 5.2.2 Añadir un nuevo departamento

En el listado 15 se muestra el código del servlet que controla la creación de un nuevo departamento. La secuencia de acciones es similar a la mostrada para el caso de uso añadir empleado, por lo que como ejercicio adicional, puede intentar dibujar el diagrama de secuencia asociado a este caso de uso.

Listado 15: Código del *servlet* `DepartamentoNew.java`.

```java
package sssi.tasi.personal.servlet;

import java.io.IOException;
import java.util.TreeMap;

import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
```

```
9  import javax.servlet.http.HttpServlet;
10 import javax.servlet.http.HttpServletRequest;
11 import javax.servlet.http.HttpServletResponse;
12
13 import sssi.tasi.personal.bo.DepartamentoBoRemote;
14 import sssi.tasi.personal.bo.EmpleadoBoRemote;
15
16 /**
17 * Servlet implementation class DepartamentoNew
18 */
19 @WebServlet("/DepartamentoNew")
20 public class DepartamentoNew extends HttpServlet {
21   private static final long serialVersionUID = 1L;
22   @EJB
23   private EmpleadoBoRemote empleadoBO;
24   @EJB
25   private DepartamentoBoRemote departamentoBO;
26
27   /**
28   * @see HttpServlet#HttpServlet()
29   */
30   public DepartamentoNew() {
31     super();
32   }
33   protected void processRequest(HttpServletRequest request, HttpServletResponse
         response)
34   throws ServletException, IOException {
35
36     TreeMap<String, Integer> empleados;
37     String nombre;
38     int manager;
39
40     String accion = request.getParameter("action");
41     switch (accion) {
42         case "form":
43         empleados = empleadoBO.keysEmpleado();
44         request.setAttribute("empleados", empleados);
45         request.getRequestDispatcher("/departamentos/nuevoDepartamento.jsp").
               forward(request, response);
46         break;
47
48         case "add":
49         nombre = request.getParameter("nombre");
50         manager = Integer.parseInt(request.getParameter("manager"));
51         if (nombre == null) {
52           System.out.println("Error en el departamento...");
53           request.getRequestDispatcher("/departamentos/errorDepartamento.jsp").
                 forward(request, response);
54         }
55         else {
56           departamentoBO.newDepartamento(nombre, manager);
57           request.getRequestDispatcher("/departamentos/creadoDepartamento.jsp").
                 forward(request, response);
58         }
59         break;
60
61       default:
62         throw new ServletException("DepartamentoNew: óaccin desconocida o no
               especificada.");
63     }
64   }
65
66   /**
67   * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse
         response)
68   */
69   protected void doGet(HttpServletRequest request, HttpServletResponse response)
```

```
        throws ServletException, IOException {
70      processRequest(request, response);
71    }
72
73    /**
74    * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse
          response)
75    */
76    protected void doPost(HttpServletRequest request, HttpServletResponse response)
          throws ServletException, IOException {
77      processRequest(request, response);
78    }
79  }
```

**Nota:** ¡Enhorabuena! Ha completado el ejercicio casi en su totalidad. Prepárese para disfrutar del resultado...

## 5.3 Desplegar y ejecutar la aplicación

Ahora sólo queda desplegar la aplicación en el servidor de aplicaciones y ejecutar el servlet. Para ello hacemos click con el botón derecho sobre el fichero `index.jsp` en la ventana del *Project Explorer*. En el menú emergente selecciones **Run As** → **Run on Server** . En el panel de configuración del servidor de aplicaciones, seleccione su servidor Glassfish local. En las figuras 16 y 17 se muestran dos ejemplos de ejecución de la aplicación.


Figura 16: Lista de empleados tal y como se presenta a través del navegador.

# 6 Ejercicio

El objetivo de este ejercicio es ampliar nuestra aplicación web con los siguientes casos de uso:

- **Obtener detalles de empleado**, que implica:
  1. El sistema solicitará al usuario un código de empleado.
  2. El sistema buscará el empleado con el código introducido.


Figura 17: Vista desde el navegador de la lista de departamentos

3. Si el usuario existe, se mostrará un panel con toda la información asociada al usuario.
4. Si el usuario no existe, se mostrará una pantalla informativa notificando la incidencia.

- **Editar empleado**:
  1. El sistema solicitará al usuario un código de empleado.
  2. El sistema buscará el empleado con el código introducido.
  3. Si el usuario existe:
     (a) El sistema mostrará un formulario con toda la información asociada al usuario.
     (b) El usuario podrá modificar cualquier dato asociado al empleado.
     (c) Si es necesario, el sistema deberá reflejar todos los cambios en la base de datos.
  4. Si el usuario no existe, se mostrará una pantalla informativa notificando la incidencia.
- **Cambiar manager**, cuyo proceso es:
  1. El usuario selecciona un departamento de una lista desplegable.
  2. El sistema muestra los datos del manager actual y la posibilidad de seleccionar un nuevo manager a partir de una lista desplegable.
  3. Cuando se modifica el mánafer, el sistema debe reflejar todos los cambios en la base de datos. Analice el código del método `DepartamentoBo:newDepartamento` para determinar qué acciones es necesario realizar para implementar el caso de uso.