Framework para facilitar el desarrollo de aplicaciones Java / Java EE

Francisco Javier Gil Gala. UO230948@uniovi.es. Universidad de Oviedo.

V3: Se continua con el framework de la versión 2 pero se le añadiedole más servicios. Dicha funcionalidad adicional es probada a través de dos nuevos ejemplos: ExampleArqJSFandSimpleJDBC y ExampleArqJSFandJPA

1 Introducción

La siguiente propuesta de arquitectura de aplicaciones esta basada en el uso de un modelo definido en 3 capas: lógica, persistencia y presentación. Este diseño intenta primar la organización de la aplicación en varios paquetes permitiendo, ente otras cosas, una gran escalabilidad en las aplicaciones construidas. Esta arquitectura no tiene solo un ámbito educativo o de investigación si no que pretende ser un completo framework para aplicaciones Java / Java EE.

La arquitectura define una serie de clases e interfaces con objeto de estructurar toda la aplicación. Estas clases e interfaces no deben ser modificadas por el programador, ya que el comportamiento de dichas clases se encuetra parametrizado a través de un archivo de propiedades o extendiendo la funcionalidad de las clases.

No estamos ante nada nuevo, simplemente es una arquitectura basada en el modelo de 3 capas pero aplicando los principios SOLID y algunos de los patrones de diseño propuestos en [1] y [2]

Además de incorporar todo el marco de trabajo también incopora clases de utilidad para generar bases de datos embebidas en la aplicación o para generar la estructura de paquetes y clases de ejemplo.

2 Requisitos y librerias necesarias

- Derby
- HSQLDB
- javax.faces o cualquier otra implementación de JSF 2.1 o superior.
- ojdbc6
- Una implementación de JPA, por ejemplo Hibernate 2.0 o superior.

3 Diseño propuesto

Respecto a la versión 1, en esta revisión del framework no se impone una estructura de paquetes si no que será el usuario el encargado de generarla a partir de las clases del paquete generadores. Tambien existe la opción de generar versiones demo del framework.

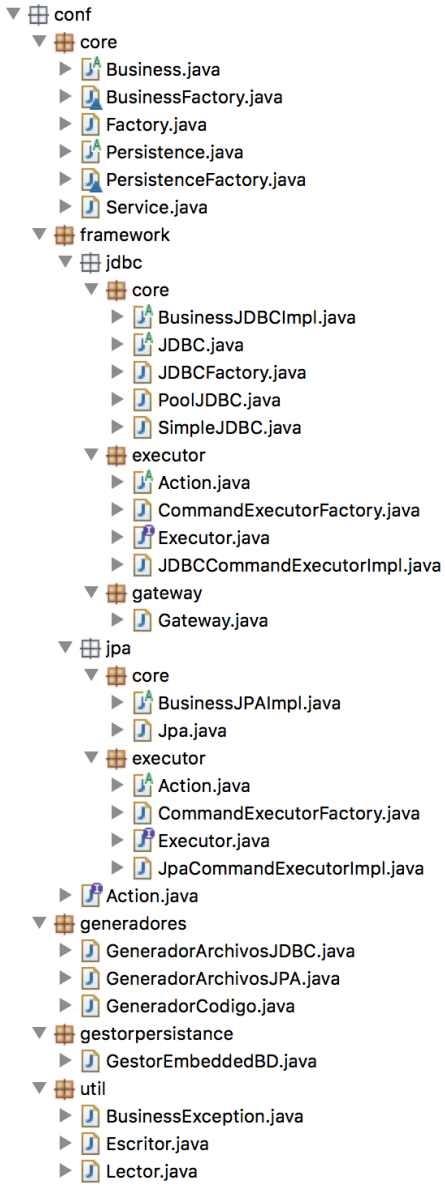

Ilustración 1

El corazón de esta arquitectura se articula en el paquete conf. Como podemos observar en la ilustración 1 principalmente tendremos dos bloques estructurados en dos paquetes: el paquete core y el paquete framework, además de los paquetes util, gestorpersistance y generadores que sirven de apoyo a los otros dos paquetes.

3.1 Paquete core

Este paquete apenas a sufrido cambios respecto a la versión 1. En la ilustración 2 se muestra la relación entre las distintas clases:

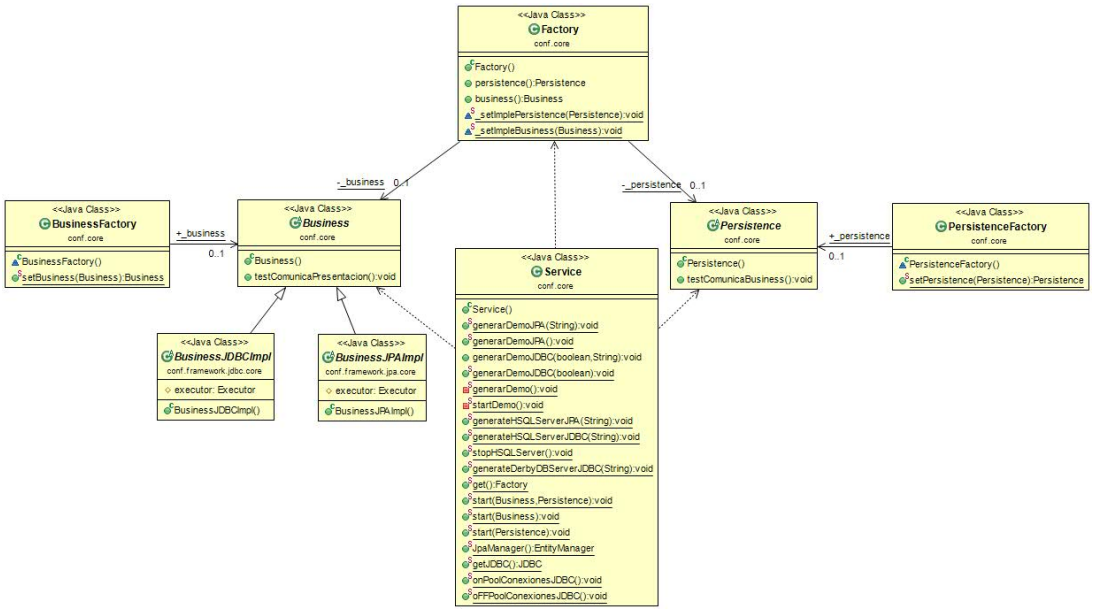

Ilustración 2

Se observa como la clase de entrada será la clase Service que será la encargada de proporcionar todos los servicios básicos del framework:

- Iniciar el framework.
- Proporcionar y gestionar el acceso entre las distintas capas.
- Proporcionar conexión a la base de datos, bien a través de JDBC "a pelo" o bien a través de un ORM como JPA.
- Iniciar y parar una base de datos embebida.
- Generar clases de ejemplo para JPA o para JDBC.

Las demás clases que sirven de apoyo a la clase Service son:

- Factory será una factoria de factorias encargada de crear las factorias de persistencia y de negocio.
- BusinessFactory y PersistenceFactory serán las encargadas de crear las implementaciones concretas del usuario de Business y Persistence.
- Business y Persistence serán las clases que el usuario deberá extender para utilizar el framework, dependiendo de como se extienda (por BusinessJDBCImpl o BusinessJPAImpl) el framework utilizará las clases de JPA o las de JDBC.

3.2 Paquete framework

3.2.1 Paquete jdbc

POR HACER, pero similar a JPA.

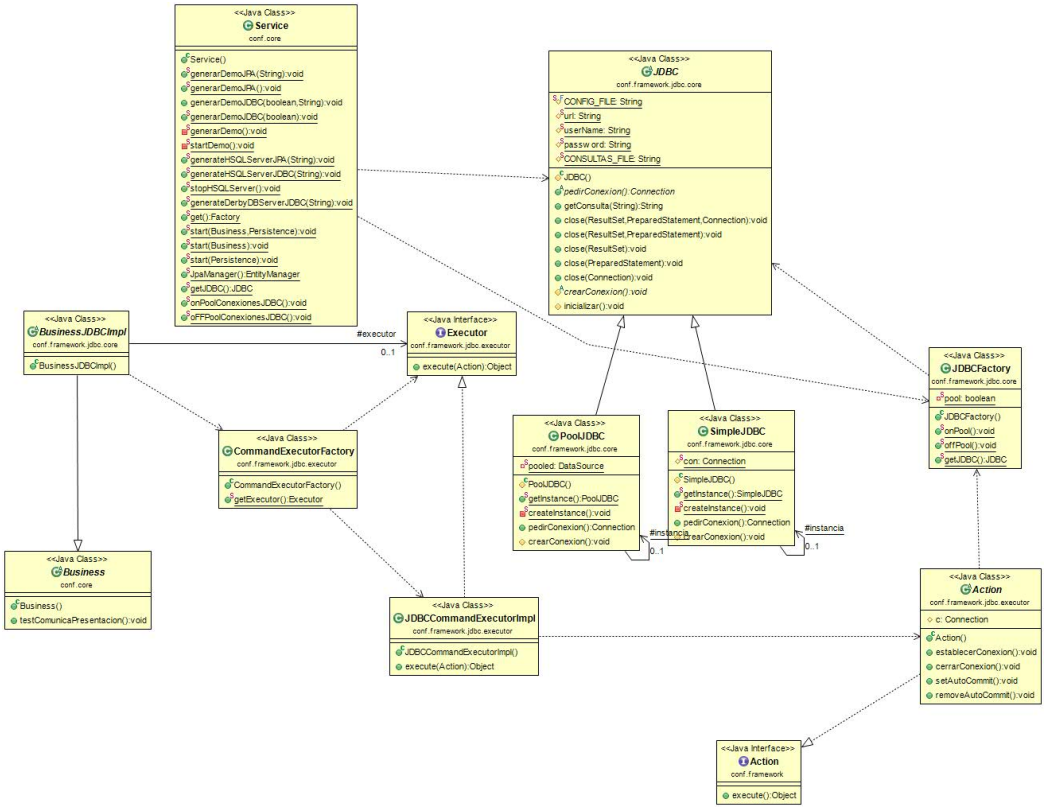

Ilustración 3

3.2.2 Paquete jpa

El paquete executor es la implementación concreta de JPA sin especificar aun que implementación concreta se esta utilizando. La idea de este conjunto de código es mantener la máxima independencia entre la implementación concreta de JPA y la capa business.

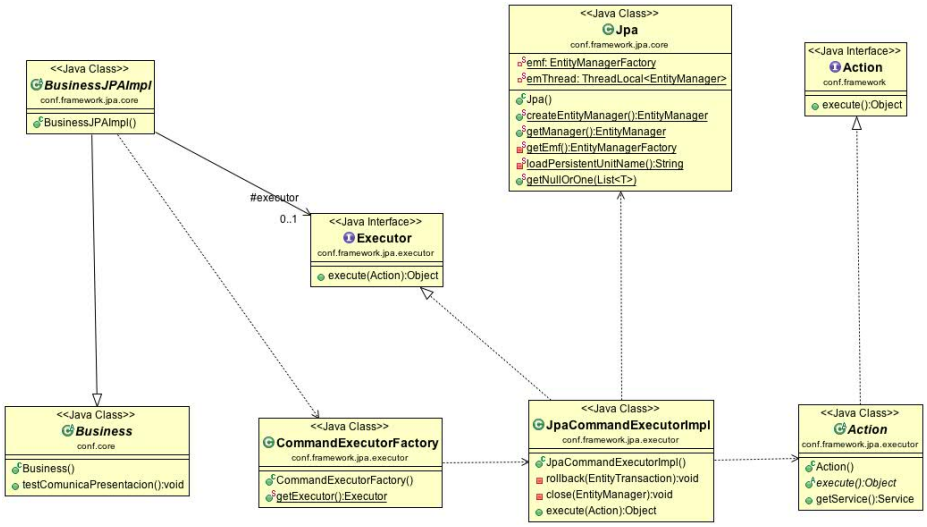

Ilustración 4

Como se puede observar en la ilustración 4 JPACommandExecutorImpl será una implementación de la interfaz Executor que recibirá una implementación concreta de la interfaz Action. Como se verá en los siguientes apartados cada una de las operaciones de negocio estará representada por una implementación de Action que será pasada por parámetro a la implementación concreta de Executor.

```
private Executor executor = CommandExecutorFactory.getExecutor();
```

Ilustración 5

Como se puede observar en la ilustración 5 en algun lugar de la capa business deberá existir una llamada a la factoria de Executor, en este caso CommandExecutorFactory, que devuelva la implementación concreta de Executor, en este caso JPACommandExecutorImpl. El framework ya lo proprociona a través de la clase BusinessJPAImpl que deberá ser extendida por la clase

Por otro lado comentar que, aunque aparezca en el diseño final, la clase JPA unicamente se encarga en gran medida de buscar la EntityManager y de crearla a partir de una la unidad de persistencia, por lo que su diseño no tiene gran interés en el diseño de esta arquitectura.

3.3 Otros aspectos relevantes

3.3.1 Generadores

POR HACER.

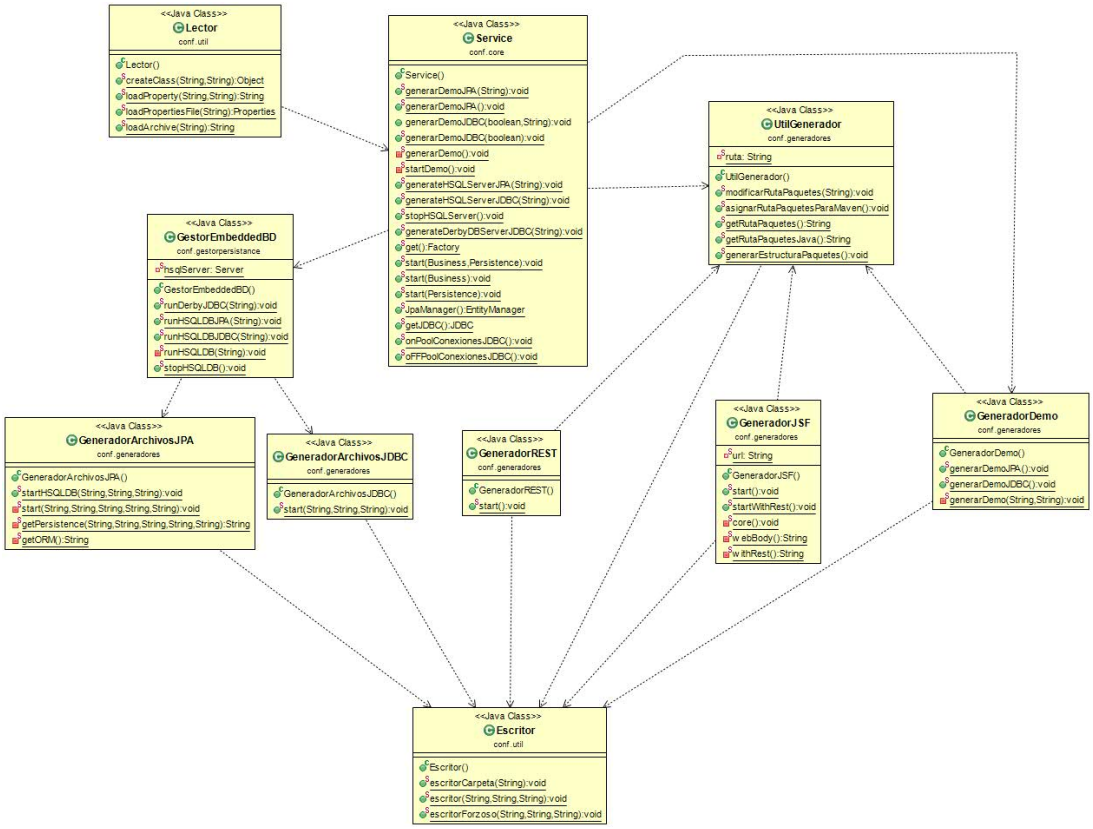

Ilustración 6

3.3.2 Gestor de persistencia

3.3.3 Paquete util

3.4 Ejemplos

Se basa en un simple ejemplo para crear y listar libros mediante una aplicación de consola (para los ejemplos de JDBC y JPA) o bien una aplicación web dinámica (para los ejemplos JSF con JDBC y JSF con JPA)

3.4.1 Ejemplo JDBC

POR HACER, pero similar a JPA

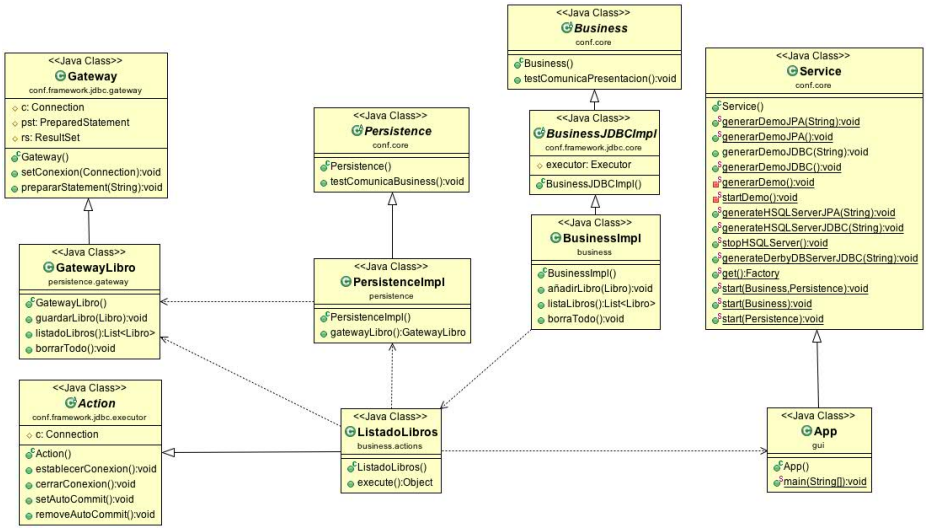

Ilustración 7

3.4.2 Ejemplo JPA

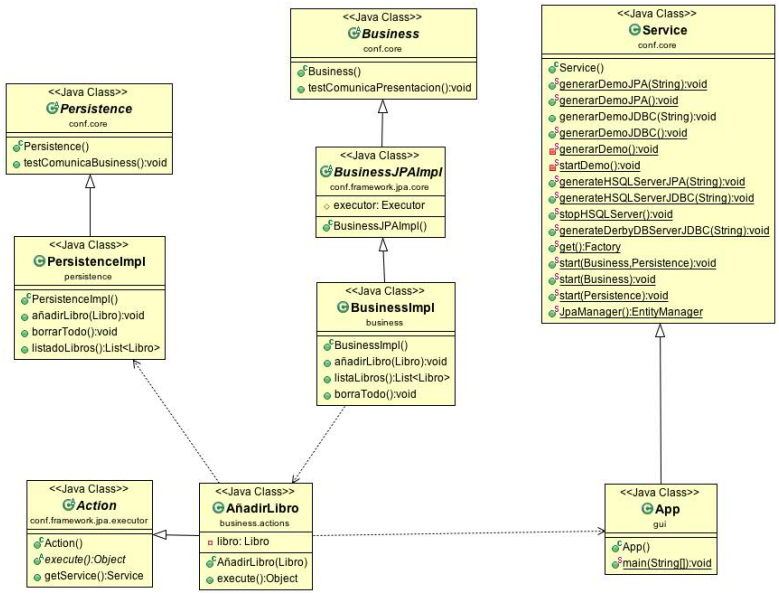

Ilustración 8

En la ilustración 8 se observa el funcionamiento del framework. Unicamente se debe extender las clases especificas del framework en las clases concretas del usuario, es decir, la clase BusinessImpl, creada por el usuario, extenderá la clase BusinessJPAImpl proporcionada por el framework. De forma similar pasará con las clases concretas del usuario: PersistenceImpl y App, que extenderán de Persistence y de Service respectivamente.

El usuario podrá acceder de esta forma a los servicios de la clase Service a través de su propia implementación, la clase App, y si tiene que modificarlos podrá modificarlos libremente.

En la ilustración 9 se muestra la distribución de paquetes del ejemplo.

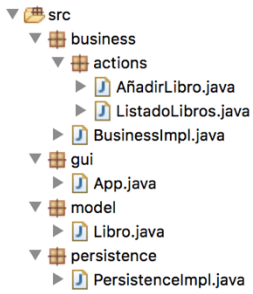

Ilustración 9

```
public class App extends Service {

    public static void main(String[] args) throws BusinessException {

        // Inicia servicios de persistencia; no es necesario para iniciar el
        // framework
        String databasename = "internal";
        generateHSQLServerJPA(databasename);

        // Inicia el framework
        start(new BusinessImpl(), new PersistenceImpl());

        // App
        BusinessImpl b = (BusinessImpl) get().business();
        List<Libro> libros = b.listaLibros();
        for (Libro l : libros)
            System.out.println(l);
        b.añadirLibro(new Libro("telecadas varias"));
        b.añadirLibro(new Libro("patrones de diseño"));
        libros = b.listaLibros();
        for (Libro l : libros)
            System.out.println(l);

        // necesario forzar la detención o finalizar la ejecución
        stopHSQLServer();
    }
}
```

Ilustración 10

Como se observa en la ilustración 10, la clase principal de la aplicación PUEDE extender de la clase Service. Para la inicialización del framework es necesario llamar al metodo start(…) y pasarle las implementaciones concretas de las clases Business y Persistance del usuario. Será dependiendo de la implementación que estas clases usen (de que clase extiendan) la que determine como funcionará el framework, si estará basado en JPA o en JDBC.

Cada una de las operaciones de negocio asociadas com servicio deberán ser implementada en una clase que extienda de Action, como se puede observar en la ilustración 11.

```
public class BusinessImpl extends BusinessJPAImpl {

    public void añadirLibro(Libro libro) throws BusinessException {
        executor.execute(new AñadirLibro(libro));
    }

    @SuppressWarnings("unchecked")
    public List<Libro> listaLibros() throws BusinessException {
        return (List<Libro>) executor.execute(new ListadoLibros());
    }
}
```

Ilustración 11

En este ejemplo la operacion de añadir libro se implementará mediante la creación de una clase AñadirLibro que extienda de la clase Action del paquete conf.framework.jpa.executor

```
public class AñadirLibro extends Action {

    private Libro libro;

    public AñadirLibro(Libro libro) {
        this.libro = libro;
    }

    @Override
    public Object execute() throws BusinessException {
        ((PersistenceImpl) App.get().persistence()).añadirLibro(libro);
        return null;
    }
}
```

Ilustración 12

La forma de realizar las operaciones de persistencia no se especifica, unicamente se especifica que debe existir una clase que extienda de Persistence, en este caso PersistenceImpl, que implemente las operaciones que se hagan necesarias desde la capa de negocio.

```
public class PersistenceImpl extends Persistence {

    public void añadirLibro(Libro libro) {
        App.JpaManager().persist(libro);
    }

    @SuppressWarnings("unchecked")
    public List<Libro> listadoLibros() {
        return App.JpaManager().createNamedQuery("Libro.findAll").getResultList();
    }
}
```

Ilustración 13

Será habitual ver código como el que se muestra en la ilustración 13 donde se accede al controlador de JPA a través de la clase cocnreta del usuario App que está extendiendo la funcionalidad de la clase Server del framework.

3.4.3 Ejemplo JSF y JDBC

POR HACER, pero similar a JPA

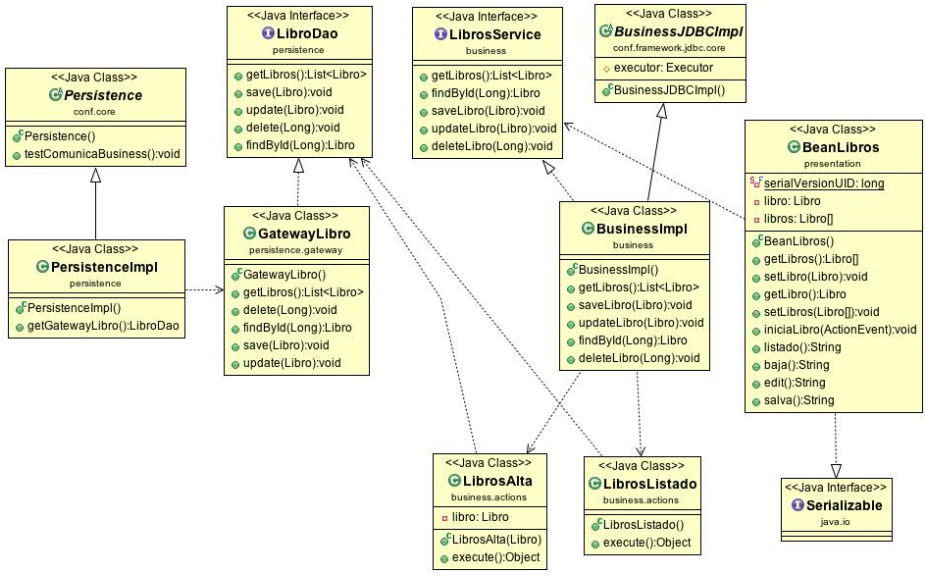

Ilustración 14

3.4.4 Ejemplo JSF y JPA

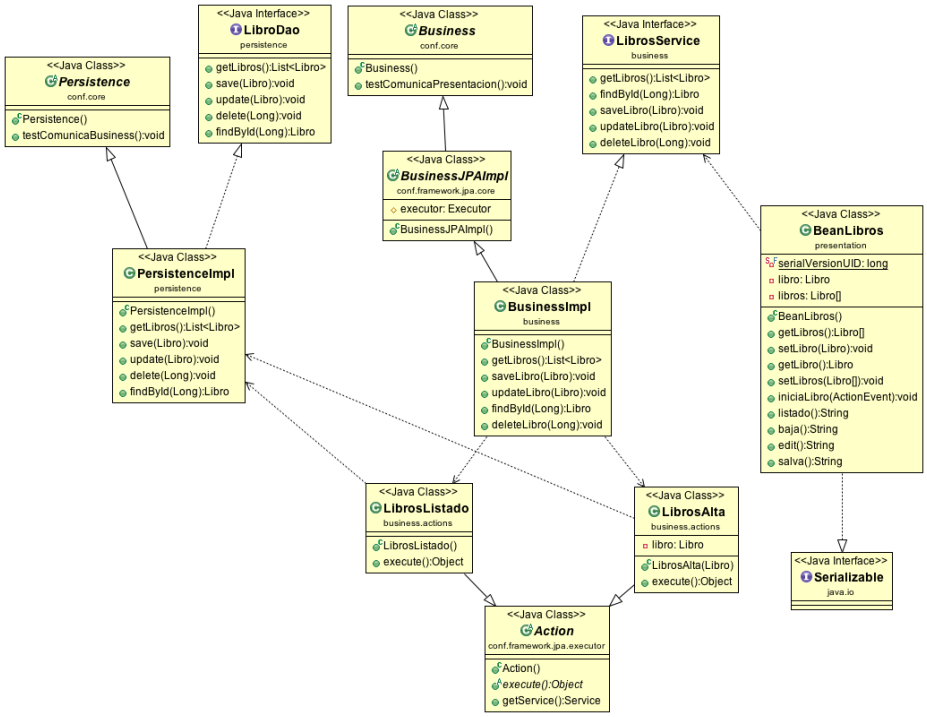

Ilustración 15

Como en los ejemplos anteriores es necesario extender las clases de Persistence y Business e inicializarlas en la clase principal a través del método start(…) de la clase Service. Como se puede observar no existe una clase que extienda de la clase Service, no es necesario, aunque se podría hacer.

```
public BeanLibros() throws SQLException, BusinessException {
    Service.start(new BusinessImpl(), new PersistenceImpl());
    iniciaLibro(null);
}
```

Ilustración 16

Como se observa en la ilustración 16, el método Service.start(…) es llamado cuando se crea el BeanLibros.

```
public String listado() {
    LibrosService service;
    try {
        service = (LibrosService) Service.get().business();
        libros = (Libro[]) service.getLibros().toArray(new Libro[0]);
        return "exito"; // Nos vamos a la vista listado.xhtml
    } catch (Exception e) {
        FacesContext.getCurrentInstance().getExternalContext().getSessionMap().put("ERROR",
                Reportador.error("errorListado"));
        return "error"; // Nos vamos la vista de error
    }
}
```

Ilustración 17

Como se observa en la ilustración 17 la funcionalidad de la clase BeanLibros es la de cualquier Bean, ejercer de controlador entre la vista y la capa de negocio, a través de los métodos: listado(), baja(),… También se observa que para la gestión de errores se hará uso de la clase conf.framework.jsf.Reportador que se encarga de acceder al bundle "msgs" definido en el archivo faces-config.xml de nuestro ejemplo.

```
public class Reportador {

    public static String error(String error) {
        FacesContext facesContext = FacesContext.getCurrentInstance();
        ResourceBundle bundle = facesContext.getApplication().getResourceBundle(facesContext, "msgs");
        return bundle.getString(error);
    }
}
```

Ilustración 18

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd"
    version="2.2">

    <application>
        <resource-bundle>
            <base-name>messages</base-name>
            <var>msgs</var>
        </resource-bundle>
    </application>

    <managed-bean>
        <managed-bean-name>controller</managed-bean-name>
        <managed-bean-class>presentation.BeanLibros</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>

    <managed-bean>
        <managed-bean-name>mensaje</managed-bean-name>
        <managed-bean-class>presentation.BeanMensajes</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
    </managed-bean>
```

Ilustración 19

Como se observa en las dos anteriores ilustraciones tendremos un bean en el framework que podremos bien usar, o bien extender su funcionalidad para implementar uno propio, que se encarga de mostrar los mensajes en las vistas.

```java
package presentation;

import javax.faces.bean.ManagedBean;

@ManagedBean(name = "mensaje")
@SessionScoped
public class BeanMensajes extends GestorMensajes {

    private static final long serialVersionUID = 1L;

}
```

Ilustración 20

```java
package conf.framework.jsf;

import java.io.Serializable;
import java.util.Map;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import javax.faces.context.FacesContext;

@ManagedBean(name = "gestor")
@SessionScoped
public class GestorMensajes implements Serializable {

    private static final long serialVersionUID = 55556L;
    private String mensaje;

    public String getMensaje() {
        Map<String, Object> session = FacesContext.getCurrentInstance().getExternalContext().getSessionMap();
        mensaje = (String) session.get("ERROR");
        return mensaje;
    }

    public void setMensaje(String mensaje) {
        this.mensaje = mensaje;
    }
```

Ilustración 21

En las dos últimas ilustraciones se observa la implementación propia, la de la aplicación, y la clase del framework que ya incorpora toda la funcionalidad necesaria. En este ejemplo se extiende unicamente por comodidad a la hora de declarar el bean en faces-conf.xml

```xml
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html">
<h:head>
    <title>APP EJEMPLO</title>
</h:head>
<h:body>
    <center>
        <h1>Libreria!</h1>
        <br />
        <h2>Aplicación de gestión de libros</h2>
    </center>
    <br />
    <br />
    <h1>#{msgs.error}</h1>
    <br />
    <p>#{mensaje.mensaje}</p>
</h:body>
</html>
```

Ilustración 22

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core">
<h:head>
    <title>APP EJEMPLO</title>
</h:head>
<h:body>
    <center>
        <h1>Libreria!</h1>
        <br />
        <h2>Aplicación de gestión de libros</h2>
    </center>
    <br />
    <br />
Listado de libros:
<br />
    <h:form>
        <h:dataTable var="libro" value="#{controller.libros}" border="1">
            <h:column>
                <f:facet name="header">Titulo</f:facet>#{libro.titulo}</h:column>
            <h:column>
                <f:facet name="header">Autor</f:facet>#{libro.autor}</h:column>
            <h:column>
                <f:facet name="header">Baja</f:facet>
                <h:commandLink action="#{controller.baja}" type="submit"
                    value="BAJA" immediate="true">
                    <f:setPropertyActionListener target="#{controller.libro}"
                        value="#{libro}" />
                </h:commandLink>
            </h:column>
            <h:column>
                <f:facet name="header">Edición</f:facet>
                <h:commandLink action="editar" type="submit" value="EDITAR"
                    immediate="true">
                    <f:setPropertyActionListener target="#{controller.libro}"
                        value="#{libro}" />
                </h:commandLink>
            </h:column>
        </h:dataTable>
    </h:form>
</h:body>
</html>
```

Ilustración 23

Como se observa en las dos últimas ilustraciones el acceso al bundle de mensajes como a las propiedades de los beans será exactamente igual a cualquier otro proyecto que no utilice el framework, en este aspecto la capa de presentación es completamente igual, a excepción cel control de excepciones mostrado en la ilustración 17.

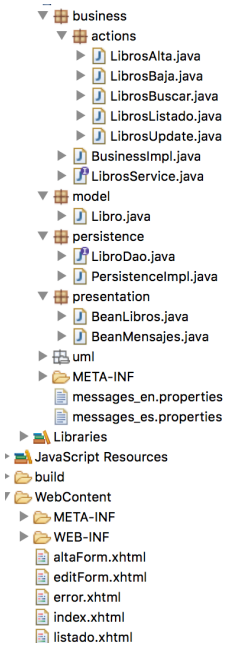

Ilustración 24

Respecto al resto de la aplicación, seguirá manteniendo un esquema de uso similar al visto en el ejemplo de JPA. En la ilustración 24 podemos observar la distribución de paquetes del ejemplo. Se observa como se tendrán las interfaces LibrosService y LibroDao, son opcionales, unicamente estructuran que operaciones deben implementar las clases BusinessImpl y PersistenceImpl y aumentar la independencia entre las capas. Por tanto, como podemos observar en las ilustraciones 25 y 26 cuando se acceda de una capa a otra se utilizará dichas interfaces ya que estas interfaces marcaran el tipo de las implementaciones concretas.

```
public String listado() {
    LibrosService service;
    try {
        service = (LibrosService) Service.get().business();
        libros = (Libro[]) service.getLibros().toArray(new Libro[0]);
        return "exito"; // Nos vamos a la vista listado.xhtml
    } catch (Exception e) {
        FacesContext.getCurrentInstance().getExternalContext().getSessionMap().put("ERROR",
                Reportador.error("errorListado"));
        return "error"; // Nos vamos la vista de error
    }
}

public String baja() {
    LibrosService service;
    try {
        service = (LibrosService) Service.get().business();
        service.deleteLibro(libro.getId());
        libros = (Libro[]) service.getLibros().toArray(new Libro[0]);
        return "exito"; // Nos vamos a la vista de listado.
    } catch (Exception e) {
        FacesContext.getCurrentInstance().getExternalContext().getSessionMap().put("ERROR",
                Reportador.error("errorBaja"));
        return "error"; // Nos vamos a la vista de error
    }
}

public String edit() {
    LibrosService service;
    try {
        service = (LibrosService) Service.get().business();
        libro = service.findById(libro.getId());
        return "exito"; // Nos vamos a la vista de Edición.
    } catch (Exception e) {
        FacesContext.getCurrentInstance().getExternalContext().getSessionMap().put("ERROR",
                Reportador.error("errorEdit"));
        return "error"; // Nos vamos a la vista de error.
    }
}
```

Ilustración 25

```
package business.actions;

import conf.core.Service;

public class LibrosListado extends Action {

    @Override
    public Object execute() throws BusinessException {
        LibroDao dao = ((PersistenceImpl) Service.get().persistence());
        return dao.getLibros();
    }
}
```

Ilustración 26

En las ilustraciones 25 y 26 se observa como se accede a los servicios de la capa de negocio y persistencia a través de las interfaces LibrosService y LibroDao respectivamente, aunque en realidad estaremos accediendo a BusinessImpl y PersistenceImple que son las clases que implementan dichas interfaces.

Referencias bibliográficas

[1] Design Patterns, Gang of Four
[2] Publicaciones de Martin Fowler