Proyecto de Integracion

Índice

1. Caso de Estudio

1.1. Introducción

A partir de un supuesto básico de la gestión de una biblioteca, vamos a crear un caso de estudio completo que evolucionará conforme estudiemos las diferentes tecnologías de la plataforma Java Enterprise.

El objetivo de esta sesión es introducir el caso de estudio que vamos a desarrollar, obtener una visión global del proyecto, fijando los casos de uso y requisitos principales y definiendo el esqueleto inicial del problema.

1.2. Ingeniería de Requisitos

El Instituto de Educación Secundaria "jUA" nos ha encargado que desarrollemos una aplicación para la gestión de los préstamos realizados en la biblioteca del centro, lo que implica tanto una gestión de los libros como de los alumnos y profesores que realizan estos prestamos.

1.2.1. Requisitos de Información (IRQ)

Tras una serie de entrevistas y reuniones con diferente personal del centro, hemos llegado a recopilar los siguientes requisitos de información:

- Respecto a un **usuario**, nos interesa almacenar:
  - *Nombre y apellidos*
  - *Login y password*
  - *Tipo de usuario / Rol*: administrador, bibliotecario, profesor, socio

    Un usuario *administrador* será el encargado de configurar la aplicación y gestionar los usuarios, mientras que un *bibliotecario* se encargará de la gestión de préstamos y libros, siendo el único tipo que pueda realizar préstamos.

    Tanto los *profesores* como los *socios* van a poder realizar reservas sobre los libros, pero con diferentes permisos, tanto en número de libros como en duración del préstamo. Para realizar un préstamo, deberán personarse en la biblioteca, y el bibliotecario formalizará el préstamo en nombre del profesor o socio.
  - *Estado de un usuario*: activo o moroso

    Los usuarios, al crearlos tendrá un estado activo. Cuando un usuario se retrase en la devolución de un libro pasará a un estado de moroso, mediante el cual se le impide la reserva y préstamo de más libros.
  - *Correo electrónico*

    - Datos referentes a su dirección, como son *calle*, *número*, *piso*, *ciudad* y *código postal*.
- Cuando un usuario se retrase en la devolución de un libro, se le creará una **multa**, de la cual nos interesa saber la *fecha de inicio* y de *finalización* de la misma. Nos han comunicado que quieren mantener un histórico de las multas que ha tenido un usuario.
- Respecto a un **libro**, nos interesa almacenar:
    - *Título y autor*
    - *ISBN*
    - *Número de páginas*
    - *Fecha de alta del libro*
- Respecto a los **préstamos**, tras muchas entrevistas deducimos que tanto las reservas, como los prestamos tienen características comunes, como son:
    - *Fecha de inicio y finalización*
    - *Usuario de la operación*
    - *Libro de la operación*

    Además, nos informan que desean mantener un histórico con las reservas y préstamos realizados tanto por los profesores como por lo socios.

1.2.2. Casos de Uso

En un principio, el número de casos de uso es muy limitado. A lo largo del proyecto nos vamos a centrar en el desarrollo de los siguientes casos de uso:

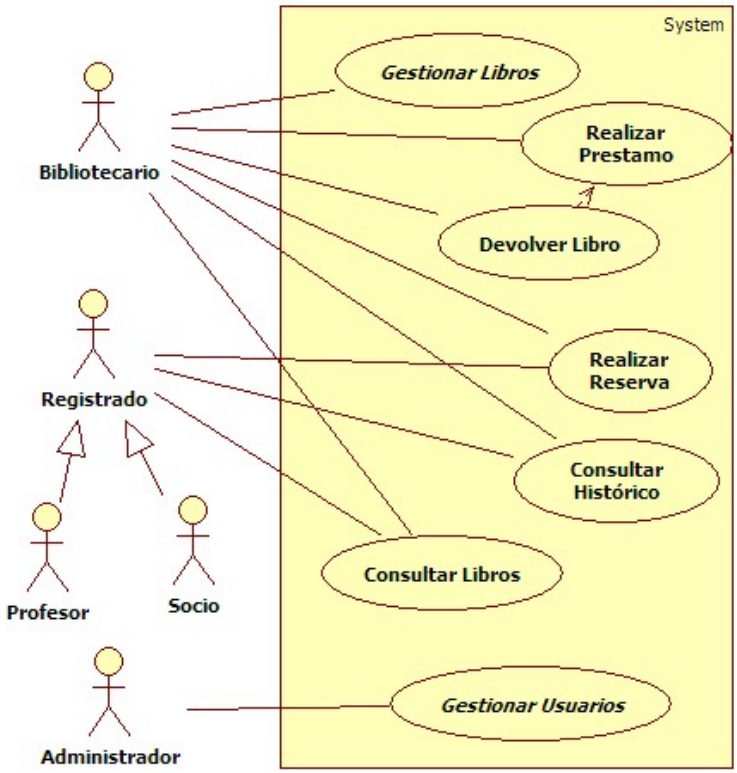


Diagrama de Casos de Uso Principal

Destacar que la **validación de usuarios** para todos los actores se considera una precondición que deben cumplir todos los casos de uso, y por lo tanto no se muestra en el diagrama.

A continuación vamos a mostrar en mayor detalle los casos de uso separados por el tipo de usuario, de modo que quede más claro cuales son las operaciones que debe soportar la aplicación.

Un poco de UML...
Recordar que las relaciones entre casos de uso con estereotipo **‹include›** representan que el caso de uso incluido se realiza siempre que se realiza el caso base. En cambio, el estereotipo **‹extend›** representa que el caso de uso extendido puede realizarse cuando se realiza el caso de uso padre (o puede que no).

1.2.2.1. Administrador

El siguiente diagrama muestra en mayor detalle las operaciones que puede realizar un usuario administrador:

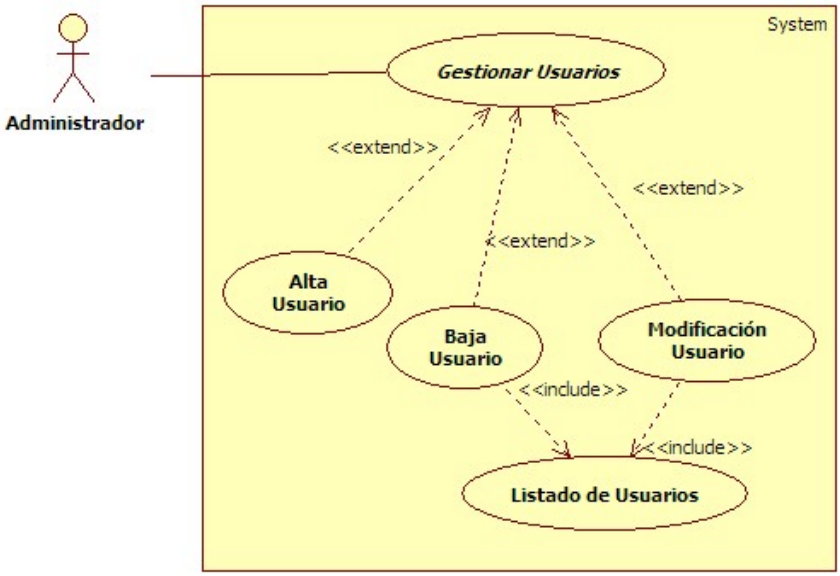


Diagrama de Casos de Uso detallado para el Administrador

Podemos observar que tanto para modificar como para eliminar un usuario, previamente debe haber realizado un listado de usuarios para poder seleccionar sobre que usuario va a realizar la modificación o borrado.

1.2.2.2. Bibliotecario

El tipo de usuario bibliotecario es el más complejo de nuestra aplicación. Como se puede observar, es el que va a poder realizar un mayor número de casos de uso:

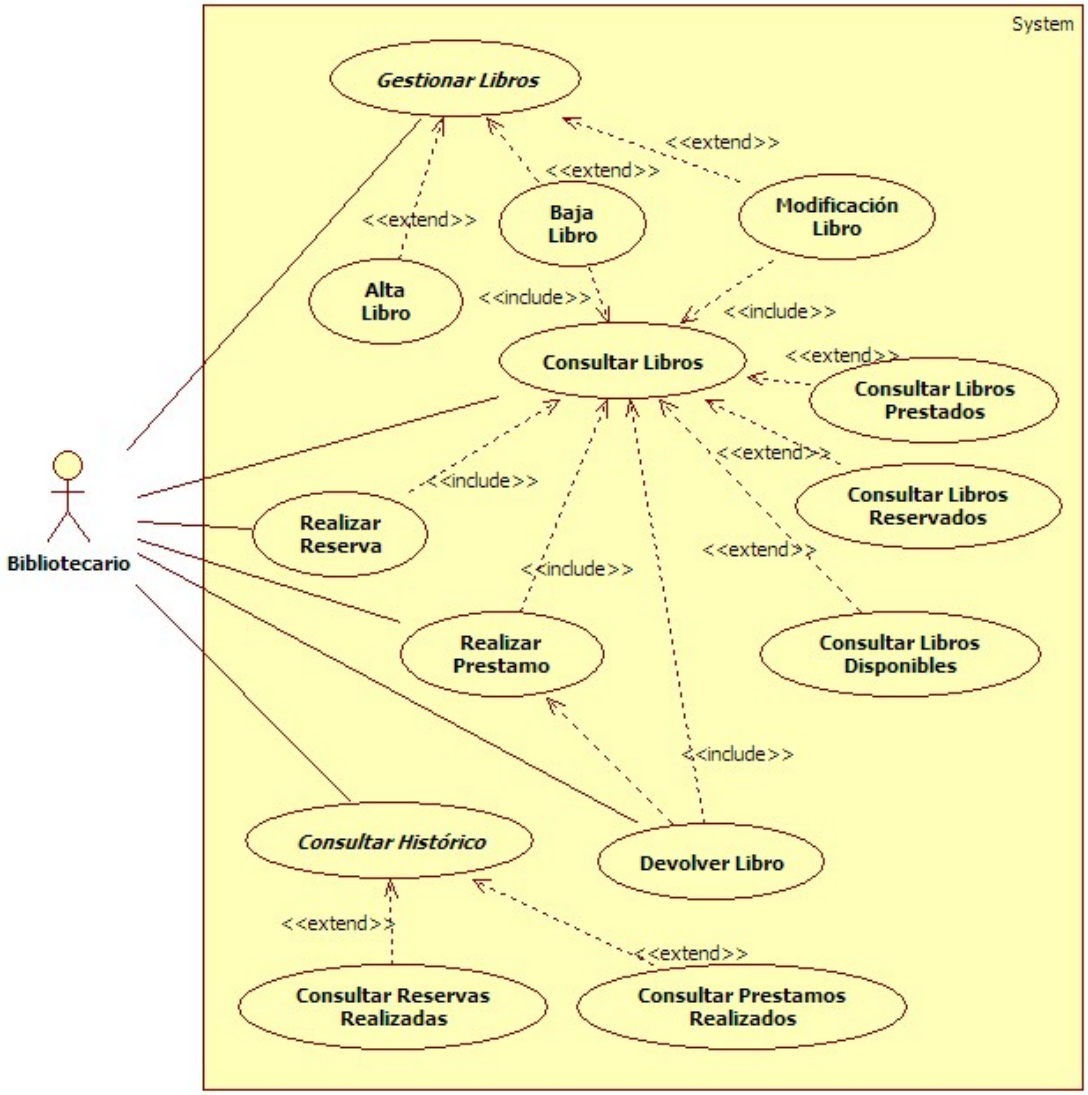


Diagrama de Casos de Uso detallado para el Bibliotecario

Claramente en el centro podemos observar como el caso "*Consultar Libros*" es el eje de la aplicación. A partir del listado de libros, ya sean disponibles, reservados o prestados, este tipo de usuario podrá gestionar los libros (alta, baja o modificación) y realizar las operaciones de reserva, préstamo o devolución (en nombre de un determinado usuario registrado). Finalmente, también tiene la posibilidad de consultar un histórico de operaciones, tanto de reservas como de prestamos realizados.

1.2.2.3. Socio o Profesor

En cuanto a un usuario cuyo tipo sea socio o profesor, y por tanto, sea un usuario

registrado en el sistema, las operaciones que puede realizar son las siguientes:

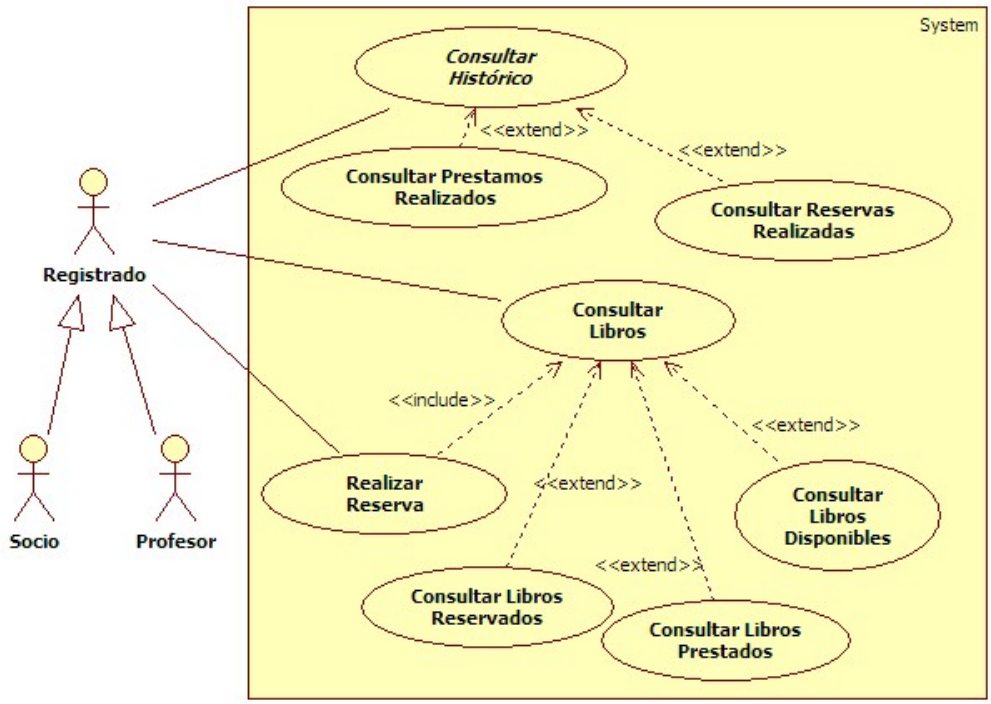


Diagrama de Casos de Uso detallado para el Usuario Registrado

En cierta medida, este tipo de usuario va a poder realizar las mismas consultas que un bibliotecario, pero sólo desde su punto de vista (sus reservas, sus prestamos), sin poder visualizar que usuarios tienen reservado/prestado un determinado libro.

Además, solamente podrá reservar libros. Para formalizar el prestamo, deberá acudir a la biblioteca, donde el bibliotecario realizará el préstamo en su nombre.

Más adelante, conforme cambien los requisitos del cliente (que siempre cambian), puede que el sistema permita renovar los prestamos a los usuarios registrados, que éstos puedan modificar su información de usuario, que los bibliotecarios obtengan informes sobre libros más prestados y/o reservados, etc...

1.2.3. Requisitos de Restricción (CRQ)

Respecto a las restricciones que se aplicarán tanto a los casos de uso como a los requisitos de información, y que concretan las reglas de negocio, hemos averiguado:

- Diferencias a la hora de realizar una operación tanto por parte de un profesor como de un socio:

	Número Máximo	Días de Reserva	Días de Prestamo

	de Libros		
Socio	3	5	7
Profesor	10	10	30

Según esta información, un socio solo puede reservar un máximo de 3 libros, teniendo 5 días para recoger el libro. Si a los 5 días de realizar la reserva no ha acudido a la biblioteca a formalizar el préstamo, se anulará la reserva, de modo que el libro quedará disponible. En cuanto a los prestamos, deberá devolver el libro como mucho una semana después.

- En el momento que un usuario tenga una demora en la devolución de un préstamo, se considerará al usuario moroso y se le impondrá una penalización del doble de días de desfase durante los cuales no podrá ni reservar ni realizar préstamos de libros.

El equipo directivo del "jUA" nos ha confirmado que para la primera versión de la aplicación se van a cumplir las siguientes premisas:

- No se permite más de una existencia de un libro. Es decir, sólo hay un ejemplar de cada libro, y la aplicación no permite libros repetidos.
- Un libro sólo tiene un autor.

1.3. Análisis y Diseño OO

A partir de esta captura de requisitos inicial, vamos a plantear los elementos que van a formar parte de la aplicación, comenzado por el modelo de clases.

1.3.1. Modelo de Clases Conceptual

A partir de los requisitos y tras unas sesiones de modelado, hemos llegado al siguiente modelo de clases conceptual representado mediante el siguiente diagrama UML:

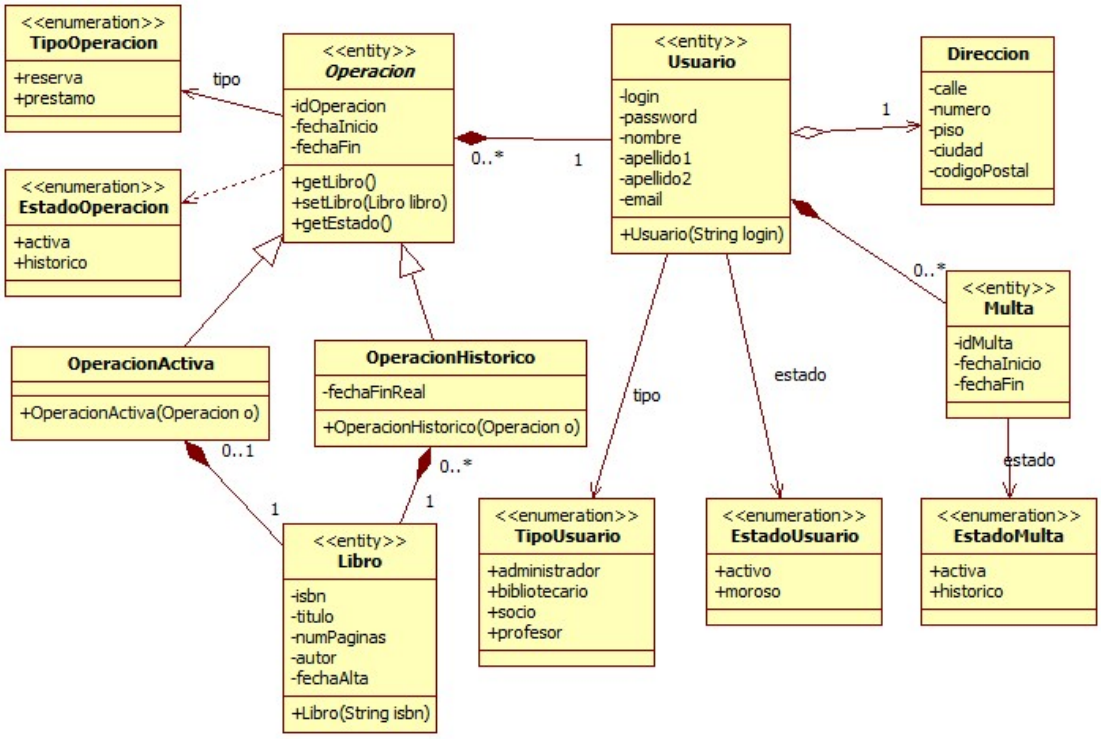


Diagrama de Clases

El elemento central de nuestro sistema va a ser la clase `Operacion`, el cual, dependiendo de un atributo enumerado, nos indicará si el objeto es un préstamo o una reserva. Además, cada operación se compone de un `Libro` y de un `Usuario`.

Alternativa

Otra forma de modelar este problema sería mediante una relación de herencia entre una clase abstracta `Operacion`, del cual tuviésemos las clases extendidas de `Reserva` y `Prestamo`, pero realmente para la lógica de nuestra aplicación, no nos interesa esta separación.
La decisión de desnormalizar esta jerarquía se debe a que realmente estas dos entidades no tienen ningún atributo diferenciador, ni se prevé que vayamos a necesitar almacenar algún dato específico de alguna entidad que lo haga diferenciarse una de la otra.

La separación de la operación en activa e histórico viene dada por querer almacenar las reservas y los préstamos que ya han sido realizados. Como podemos observar, un libro va a tener como máximo una operación activa, en cambio, puede tener muchas operaciones históricas.

1.3.2. Modelo de Datos Conceptual

Siguiendo las mismas pautas que en el modelado de clases, llegamos a un modelo

conceptual de datos muy similar al anterior. Podemos observar que ambos modelos representan el mismo problema y son casi semejantes.

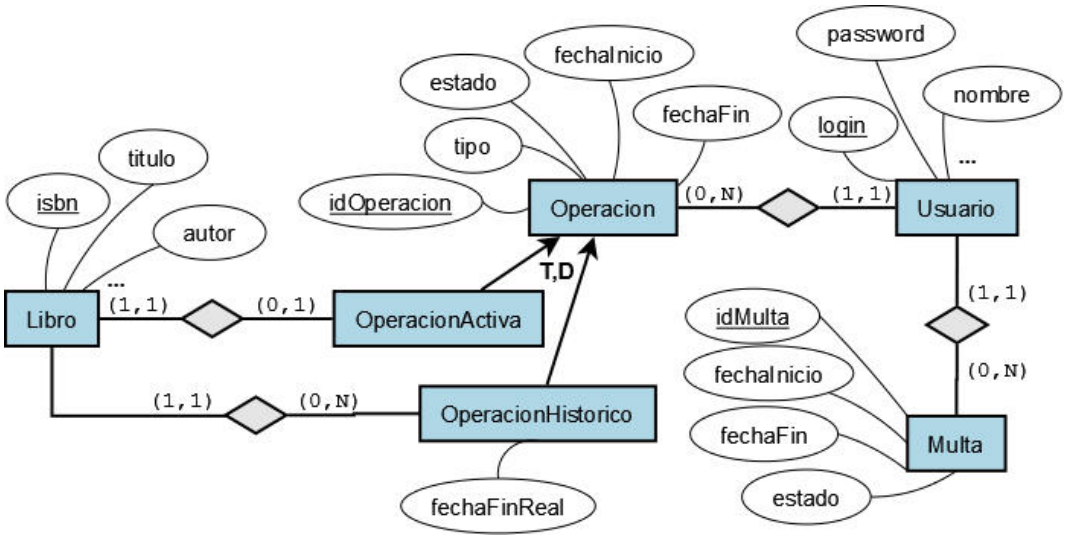


Diagrama Conceptual de Datos

Por donde empezamos

¿ Datos o clases ? Podemos empezar modelando los datos o las clases, y ambos modelos serán casi semejantes. Normalmente, la elección viene dada por la destreza del analista, si se siente más seguro comenzando por los datos, o con el modelo conceptual de clases. Otra opción es el modelado en paralelo, de modo que al finalizar ambos modelos, podamos compararlos y validar si hemos comprobado todas las restricciones.

1.3.3. Diagramas de Estado

A continuación podemos visualizar un par de diagramas que representan los diferentes estados que puede tomar tanto los libros (por medio de las operaciones) como los usuarios (mediante las acciones que conllevan algunas operaciones).

1.3.3.1. Estados de un libro

En el caso de un libro, los estados viene definidos por las operaciones que se pueden realizar. Para poder saber el estado actual de un libro, tendremos que consultar en la base de datos si hay alguna operación existente (activa) que lo relacione.

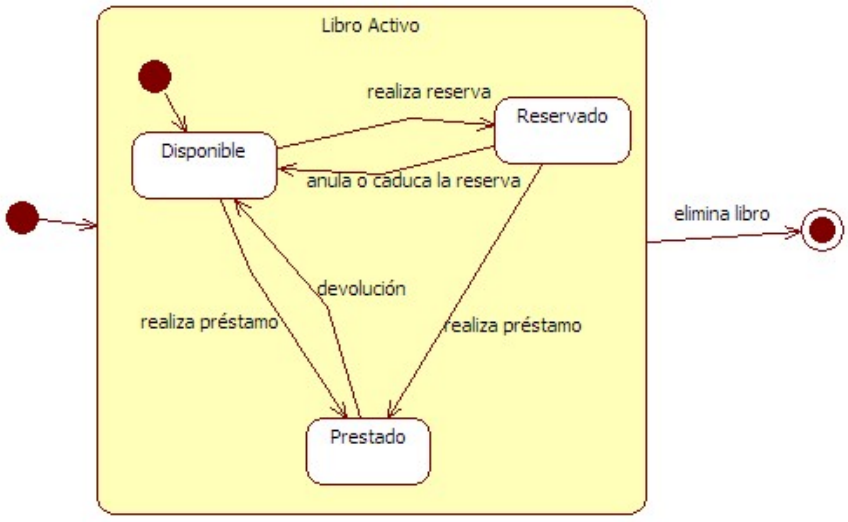


Diagrama de Estados de un Libro

1.3.3.2. Estados de un usuario

Del mismo modo, los estados de un usuario dependen de si el usuario realiza sus operaciones dentro de las reglas de negocio permitidas. Si un usuario devuelve un libro con fecha caducada, se le multará y pasará a ser moroso. Destacar que si un usuario posee un libro con fecha de devolución caducada, hasta que el usuario no devuelve el libro, no se le considera moroso, ya que no podemos calcular la cuantía de la multa.

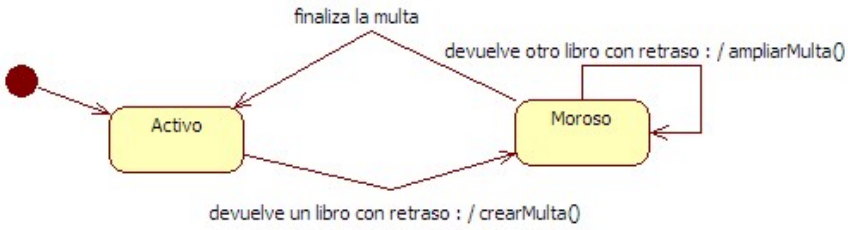


Diagrama de Estados de un Usuario

Esta lógica se puede tratar de diferentes modos, pero por ahora, como veremos a continuación, cuando un usuario entra al sistema, comprobaremos tanto si es moroso como si tiene libros pendientes de devolución cuya fecha ya ha caducado.

1.3.4. Validación de Usuarios

Dados los diferentes tipos de usuarios y la compleja lógica de entrada a la aplicación, hemos dedicado una sesión de modelado a estudiar cual debe ser el comportamiento de la

aplicación cuando un usuario se valida en el sistema.

El siguiente diagrama de actividad representa toda la posible casuística:

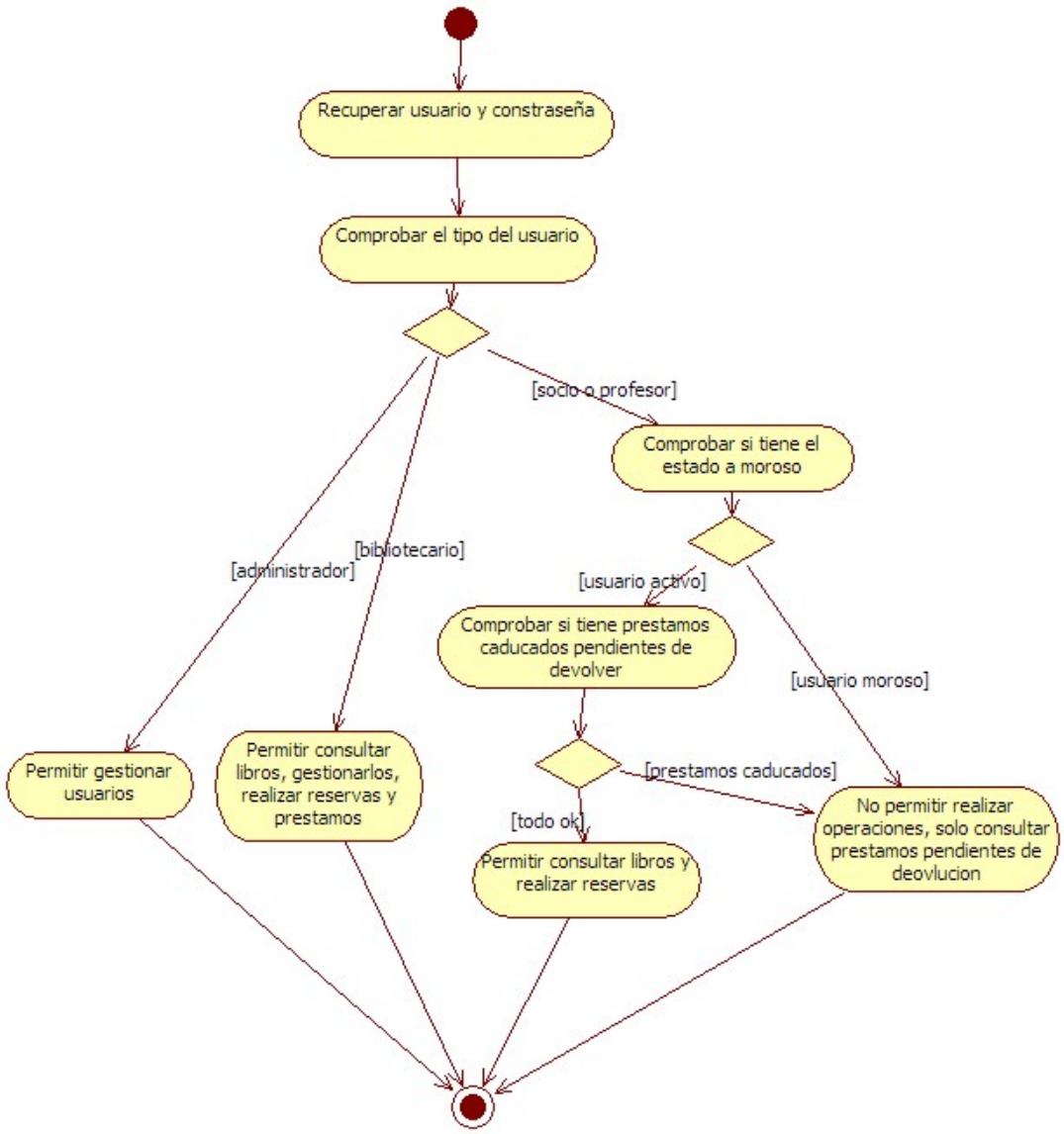


Diagrama de Actividad para la entrada al sistema

1.3.5. Reserva/Prestamo de un Libro

Tanto las operaciones de reserva como de préstamo de un libro conllevan una serie de comprobaciones que hemos definido en el siguiente diagrama:

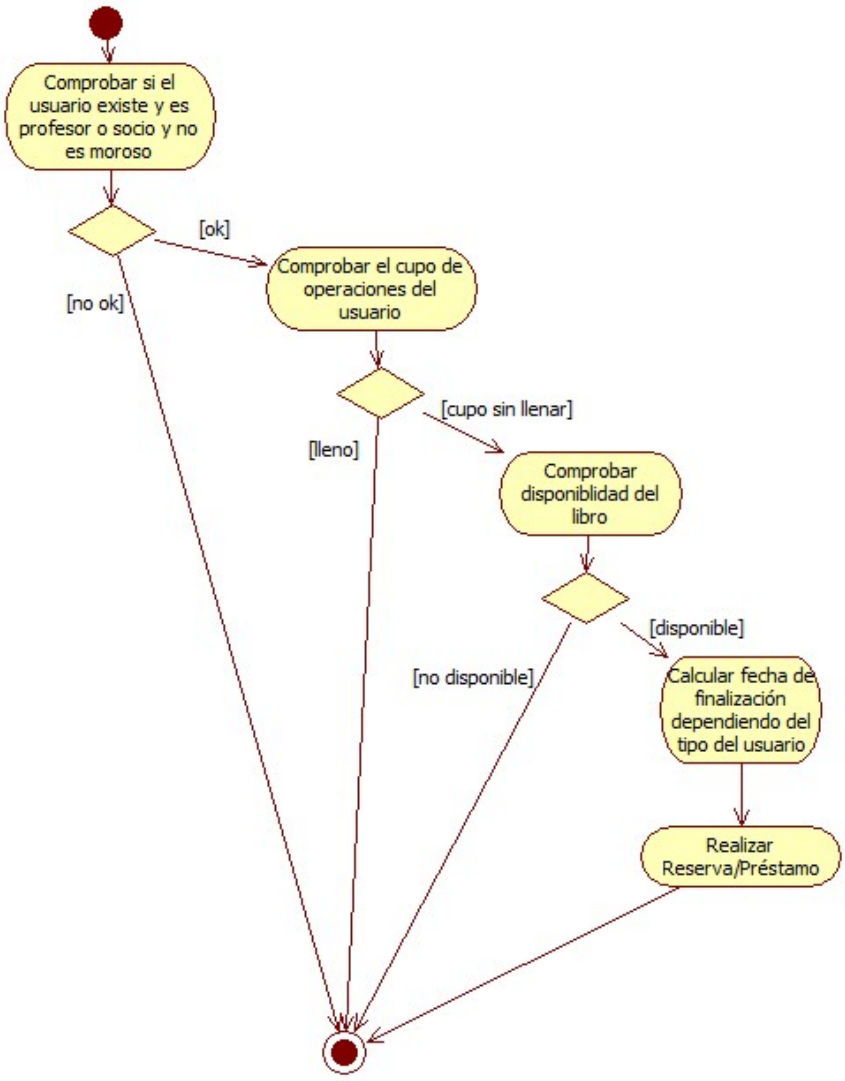


Diagrama de Actividad para la reserva de un libro

1.4. Implementación

Antes de comenzar a implementar el proyecto, vamos a crear la estructura del proyecto en Eclipse.

1.4.1. Empezamos Desde Cero

Vamos a crear un nuevo workspace al cual llamaremos **`proy-int`**, dentro de `/home/especialista`.

Una vez creado el workspace, crearemos un proyecto Java que contendrá nuestras clases que van a representar el modelo de objetos, así como las reglas de negocio, excepciones, configuración del log, etc... Este proyecto se llamará **`proy-int-comun`**.

La estructura del proyecto es la siguiente:

Estructura de Proyecto

- **src**: carpeta fuente que contiene las clases Java comunes tanto para una aplicación web como para una aplicación Swing.
  Destacar el prefijo de todos los paquetes: `es.ua.jtech.proyint`
- **test**: carpeta fuente con clases de Java basadas en *JUnit* para realizar las pruebas de la aplicación.
- **resources**: carpeta fuente con los ficheros de recursos que al desplegar la aplicación deben estar en el classpath
- **db**: scripts para la creación de la BD
- **lib**: librerías auxiliares utilizadas por el proyecto

1.4.2. Implementación del Modelo de Clases

1.4.2.1. Entidades

Las entidades implementarán el patrón *Transfer Objects* (TOs) dentro del sistema, ya que realizan funciones de meros contenedores, los cuales viajarán por las capas de la aplicación.

Además, para asegurarnos que todas nuestras entidades sean `Serializables` (y poder ampliarlas en un futuro) vamos a crear una clase abstracta, que será la clase padre de todas nuestras entidades.

```
package es.ua.jtech.proyint.entity;

import java.io.Serializable;

/**
 * Padre de todos nuestros Entities
 * Si abrimos la jerarquia de tipos (F4), podemos ver todos los
 * entities que tenemos.
 * Todos nuestros EntityObject realizan el patrón TransferObject
 *
 * @author $Author: amedrano $
 * @version $Revision: 1.5 $
 */
public abstract class EntityObject implements Serializable {

        private static final long serialVersionUID = 1L;

        // Aqui anyadiriamos los métodos que quisieramos compartir
entre todas las entidades
}
```

CVS Keywords

Es muy útil incrustar las palabra clave `$Author: amedrano $` y `$Revision: 1.5 $` de cvs dentro de nuestro código fuente. De este modo, siempre sabremos la última persona que ha subido una revisión al cvs, y el número de ésta revisión.

Todas las entidades las vamos a definir dentro del paquete `es.ua.jtech.proyint.entity`, y utilizaremos el sufijo `Entity` a modo de nomenclatura para indicar que un objeto de la aplicación es una entidad.

Cada entidad se compone de sus atributos, relaciones y de todos los getter/setter que encapsulan al objeto. Así pues, por ejemplo, la representación del *Entity Object* `LibroEntity` sería:

```
package es.ua.jtech.proyint.entity;

import java.util.Date;
import java.util.List;

/**
 * Clase que representa un libro.
 * Destacar que la app no permite 2 ejemplares de un mismo libro
 *
 * @author $Author: amedrano $
 * @version $Revision: 1.5 $
 */

public class LibroEntity extends EntityObject {

        private static final long serialVersionUID =
3957186972437085430L;

        private String isbn;
        private String titulo;
        private String autor;
        private int numPaginas;
        private Date fechaAlta;

        private List<OperacionHistoricoEntity> operaciones;

        private OperacionActivaEntity activa;

        public LibroEntity() {}

        public LibroEntity(String isbn) {
                this.isbn = isbn;
        }

        public String getAutor() {
                return autor;
        }

        public void setAutor(String autor) {
                this.autor = autor;
        }
```

```
        // resto de getters/setters
}
```

Para implementar nuestras clases, tendremos que codificar todas las clases y relaciones expuestas en el Modelo de Clases Conceptual

1.4.2.2. Enumeraciones

En cuanto a las enumeraciones, Java (desde su versión 5.0) permite su creación mediante la clase `java.lang.Enum`. En nuestro caso, por ejemplo, la enumeración de `EstadoUsuario` quedaría del siguiente modo:

```
package es.ua.jtech.proyint.entity;

/**
 * Enumeracion con los posibles estados de un usuario
 * @author $Author: amedrano $
 * @version $Revision: 1.5 $
 */
public enum EstadoUsuario {
        activo, moroso
}
```

1.4.3. Gestión de las Excepciones

Todas las aplicaciones empresariales definen una política de gestión de las excepciones de aplicación.

En nuestro caso, conforme crezca el proyecto, iremos creando nuevas excepciones. Como punto de partida, y como buena norma de programación, vamos a definir una excepción genérica de tipo *checked* (`BibliotecaException`), que será la excepción padre de todas las excepciones de aplicación de la biblioteca.

```
package es.ua.jtech.proyint;

/**
 * Excepcion padre de todas las excepciones de la aplicación
 *
 * @author $Author: amedrano $
 * @version $Revision: 1.5 $
 */
public class BibliotecaException extends Exception {

        private static final long serialVersionUID =
-5438451334294984028L;

        public BibliotecaException() {
                super();
        }

        public BibliotecaException(String message) {
```

```
                super(message);
        }

        public BibliotecaException(String message, Throwable cause)
{
                super(message, cause);
        }
}
```

Podemos observar como, al sobrecargar el constructor con los parámetros `{String, Throwable}`, nuestra excepción permitirá su uso como *Nested Exception*.

1.4.4. Implementación de las Reglas de Negocio

Es común agrupar las reglas de negocio de una aplicación en una o más clases (dependiendo de los diferentes subsistemas de la aplicación), para evitar que estén dispersas por la aplicación y acopladas a un gran número de clases.

En nuestro caso, vamos a crear un *Singleton*, al que llamaremos `BibliotecaBR` (BR = *Business Rules*). En principio, los valores estarán escritos directamente sobre la clase, pero en un futuro podríamos querer leer los valores de las reglas de negocio de un fichero de configuración).

El esqueleto de nuestras reglas de negocio será el siguiente:

```
package es.ua.jtech.proyint;

// faltan los imports

/**
 * Reglas de Negocio de la Biblioteca
 * BR = Business Rules
 *
 * Lo implementamos como un singleton por si algun dia queremos
leer
 * las constantes desde un fichero de configuración, lo podemos
 * hacer desde el constructor del singleton
 *
 * @author $Author: amedrano $
 * @version $Revision: 1.5 $
 */
public class BibliotecaBR {

        private static BibliotecaBR me = new BibliotecaBR();

        private BibliotecaBR() {
        }

        public static BibliotecaBR getInstance() {
                return me;
        }

        /**
         * Calcula el número de dias de plazo que tienen un usuario
```

```
para
         * realizar una reserva (Socio = 5 , Profesor = 10)
         * @param tipo tipo del usuario
         * @return número de dias de reserva
         * @throws BibliotecaException el tipo del usuario no es
válido
         */
        public int calculaNumDiasReserva(TipoUsuario tipo)
                        throws BibliotecaException {
                // TODO Completar
        }

        /**
         * Calcula el número de dias de plazo que tienen un usuario
para
         * realizar un prestamo (Socio = 7 , Profesor = 30)
         * @param tipo tipo del usuario
         * @return número de dias del prestamo
         * @throws BibliotecaException el tipo del usuario no es
válido
         */
        public int calculaNumDiasPrestamo(TipoUsuario tipo)
                        throws BibliotecaException {
                // TODO Completar
        }

        /**
         * Valida que el número de operaciones realizadas por
         * un determinado tipo de usuario se inferior al cupo
definido
         * @param tipo tipo del usuario
         * @param numOp número de operación que ya tiene realizadas
         * @throws BibliotecaException el cupo de operacion esta
lleno,
         *  o el tipo del usuario no es el esperado
         */
        public void compruebaCupoOperaciones(TipoUsuario tipo, int
numOp)
                        throws BibliotecaException {
                // TODO Completar
        }
}
```

Podemos observar que vamos a tener un método por cada regla de negocio, y que estos métodos lanzarán excepciones de aplicación en el caso de un comportamiento anómalo.

Para facilitar la implementación de estas reglas, hemos escrito una clase de prueba JUnit. Esta clase se encuentra dentro de la carpeta test, en el mismo paquete que la clase a probar.

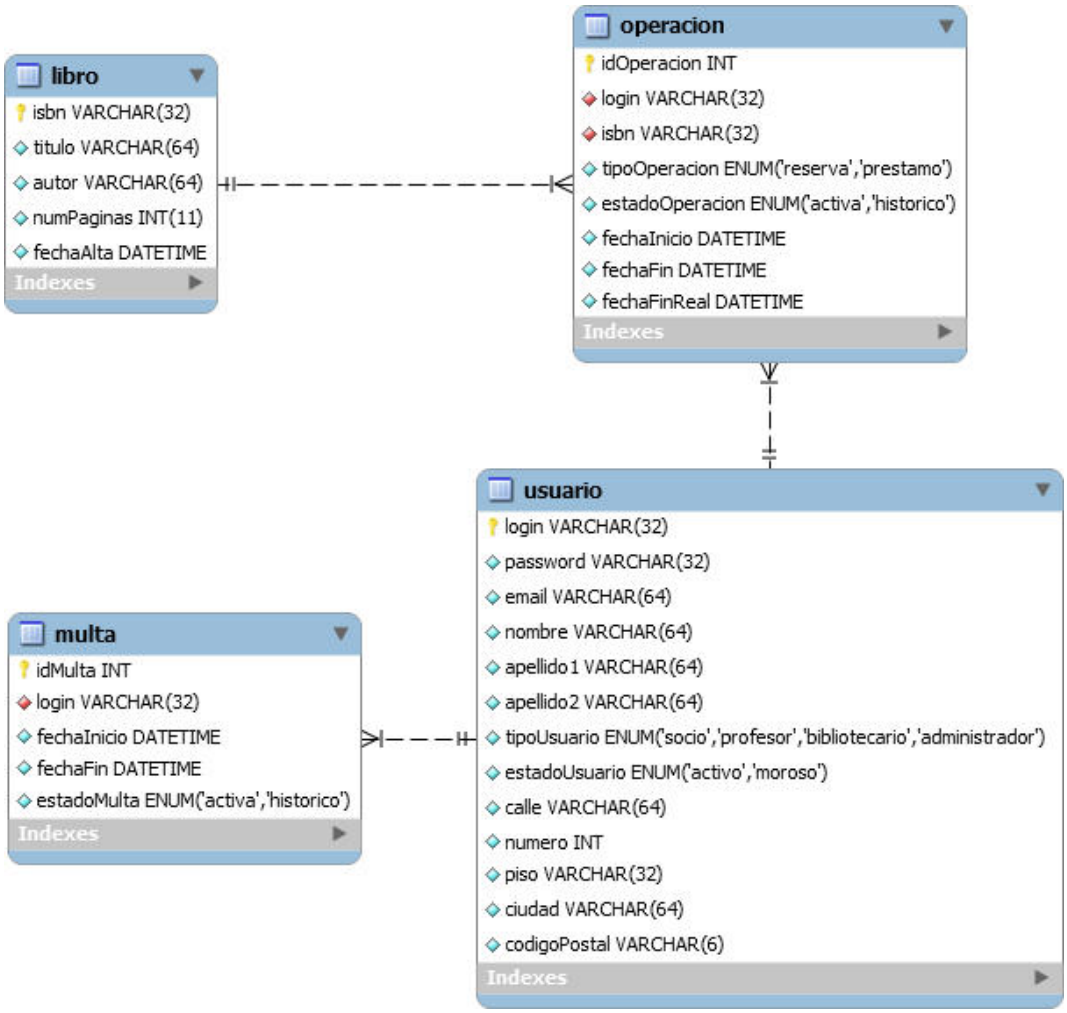
```
package es.ua.jtech.proyint;

// faltan los imports

/**
 * Prueba JUnit sobre las reglas de negocio
 *
```

```
 * @author $Author: amedrano $
 * @version $Revision: 1.5 $
 */
public class BibliotecaBRTest {

  @Test
  public void testCalculaNumDiasReserva() throws
BibliotecaException {

    int diasProfesor =
BibliotecaBR.getInstance().calculaNumDiasReserva(
        TipoUsuario.profesor);
    assertEquals(diasProfesor, 10);

    int diasAlumno =
BibliotecaBR.getInstance().calculaNumDiasReserva(
        TipoUsuario.socio);
    assertEquals(diasAlumno, 5);

    try {
      BibliotecaBR.getInstance().calculaNumDiasReserva(
          TipoUsuario.administrador);
      fail("Un administrador no puede hacer reservas");
    } catch (BibliotecaException e) {
    }

    try {
      BibliotecaBR.getInstance().calculaNumDiasReserva(
          TipoUsuario.bibliotecario);
      fail("Un bibliotecario no puede hacer reservas");
    } catch (BibliotecaException e) {
    }

  }

  @Test
  public void testCalculaNumDiasPrestamo() throws Exception {

    int diasProfesor =
BibliotecaBR.getInstance().calculaNumDiasPrestamo(
        TipoUsuario.profesor);
    assertEquals(diasProfesor, 30);

    int diasAlumno =
BibliotecaBR.getInstance().calculaNumDiasPrestamo(
        TipoUsuario.socio);
    assertEquals(diasAlumno, 7);

    try {
      BibliotecaBR.getInstance().calculaNumDiasPrestamo(
          TipoUsuario.administrador);
      fail("Un administrador no puede hacer prestamos");
    } catch (BibliotecaException e) {
    }

    try {
      BibliotecaBR.getInstance().calculaNumDiasPrestamo(
          TipoUsuario.bibliotecario);
      fail("Un bibliotecario no puede hacer prestamos");
```

```
    } catch (BibliotecaException e) {
    }

  }

  @Test
  public void testCompruebaCupoOperaciones() {

    try {
      BibliotecaBR.getInstance().compruebaCupoOperaciones(
          TipoUsuario.socio, 2);
      BibliotecaBR.getInstance().compruebaCupoOperaciones(
          TipoUsuario.socio, 0);
    } catch (BibliotecaException e) {
      fail("No deberia fallar el socio");
    }

    try {
      BibliotecaBR.getInstance().compruebaCupoOperaciones(
          TipoUsuario.socio, 4);
      fail("Debería evitar que un socio pudiera hacer más de 3
operaciones");
    } catch (BibliotecaException e) {
    }

  }
```

1.4.5. Modelo de Datos Físico

A partir del modelo de datos conceptual, y conjunto al modelo de clases conceptual, la representación que vamos a utilizar del modelo EER es la siguiente:



Modelo Físico de Datos

La principal diferencia con el modelo de datos conceptual y con el modelo de clases es que la relación de generalización/herencia entre `Operacion` y `(OperacionActiva, OperacionHistorico)` ha desaparecido. Así pues, hemos desnormalizado la base de datos. Esta decisión se debe a los pocos campos que diferencian a un hijo de otro, y que los hijos no son muy diferentes de los padres. Así pues, hemos creado un nuevo campo enumerado, `estadoOperacion`, que nos indica si la operación esta activa o en histórico. En este último caso, el valor del campo `fechaFinReal` debe estar rellenado.

En segundo lugar, para permitir relaciones de muchos a muchos (M:N) entre `usuario` y `libro` hemos creado un campo autoincrementable en la tabla `operacion`. Mediante este mecanismo vamos a permitir que un usuario realice un préstamo del mismo libro en fechas distintas.

Si quisiéramos evitar este campo, pondríamos como clave primaria el conjunto de campos `{login, isbn, fechaInicio}`.

Cuidado

Mucho cuidado con prestar el mismo libro 2 veces a 2 usuarios distintos en fechas solapadas. Tal como hemos definido el modelo, nuestra BD permite esta casuística.

Por último, todos los atributos de la clase `Direccion` quedan embebidos dentro de la tabla `usuario`.

Destacar que las enumeraciones del modelo de objeto se han transformado en enumeraciones de la base de datos (ésta es una característica propia de MySQL). Estas enumeraciones se tratan como campos `varchar` con reglas de integridad que comprueban que la información introducida siempre sea de uno de los posibles tipos definidos en la enumeración.

Los scripts SQL de creación de la Base de Datos y carga de los datos los podemos descargar desde aquí: (biblioteca.sql y datos.sql). Actualmente, la base de datos ya está creada en la instancia de MySQL instalada en la máquina virtual.

Estos scripts los vamos a colocar dentro de la carpeta **`db`** del proyecto *Eclipse*. Para desplegar ambos scripts, podemos hacerlo via la herramienta *MySQL Query Browser*.

1.5. Entorno de Desarrollo

A lo largo del proyecto, vamos a basarnos en la plataforma *Eclipse Ganymede + MySQL 5.0* (entregados en el DVD de inicio de curso). Más adelante, ya entrarán en juego el servidor web (*Apache Tomcat*) y el servidor de Aplicaciones (*Glassfish*).

Dentro de las prácticas comunes a lo largo del proyecto tenemos:

- una estructura de proyecto limpia y estándar
- uso de logs para todo tipo de mensajes de debug y error
- uso de convenciones de código Java

1.5.1. Gestión de Logs

Para la gestión de logs, pese a saber que *log4j* es la mejor oferta que existe en el mercado actual, para evitar problemas futuros de versiones, vamos a utilizar el envoltorio ofrecido por las librerías *commons-logging* de *Jakarta*, de modo que envolvamos a *log4j*. Ambas librerías se encuentran en la página de software de jTech.

La gestión de logs debe permitir que el fichero de log (`proy-int-comun.log`) cambie cada día (`DailyRollingFileAppender`), y que cada mensaje de log muestre:

- timestamp
- nivel
- mensaje

Javadocs de librerías

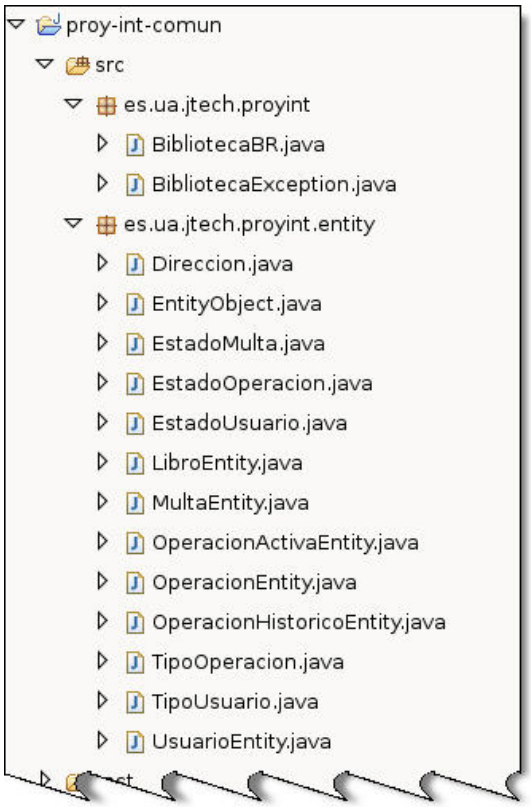
Es muy cómodo configurar donde se encuentra los javadocs de las librerías para tener la documentación sensible al contexto.

1.6. A Entregar

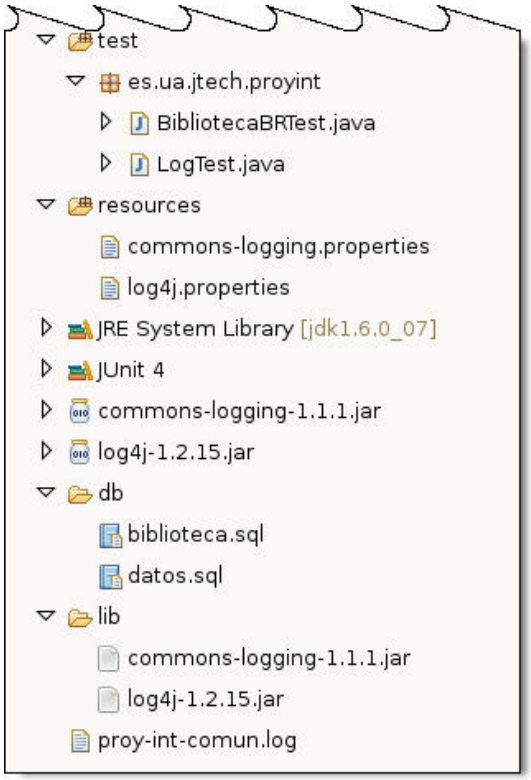
En esta sesión vamos a preparar la base para el resto de sesiones. Por ellos, debemos crear un proyecto Eclipse, al que llamaremos **proy-int-comun**, el cual contenga:

1. estructura de proyecto, siguiendo la nomenclatura definida
2. modelo de objetos dentro del paquete `es.ua.jtech.proyint.entity`, implementando cada entidad con los atributos representados en el diagrama UML, sus relaciones, y teniendo en cuenta la relación de herencia con la clase `EntityObject` y los constructores necesarios.
3. clase `BibliotecaBR` con las reglas de negocio, implementada como un singleton, la cual debe pasar las pruebas JUnit aportadas. Para implementar esta clase, es necesaria la clase `BibliotecaException`.
4. gestión de logs (`proy-int-comun.log`) con una política de cambios diarios
5. estructura y despliegue de la base de datos

Las siguientes imágenes muestran todos los archivos que debe contener el proyecto Eclipse:



Estructura de Proyecto I



Estructura de Proyecto II

El trabajo a realizar se debe efectuar durante la sesión actual, por lo cual no debería ser necesario emplear tiempo extra. De todos modos, el **plazo final** de entrega será el jueves de la semana que viene (*jueves 13 de Noviembre*).

2. Acceso a la base de datos

2.1. Introducción

El objetivo de esta sesión de integración será implementar la capa de acceso a datos de nuestra aplicación, para lo que utilizaremos el patrón DAO. Dado que el acceso a datos es un elemento crítico, sobre el que se apoyarán el resto de capas de la aplicación, y que además es probable que sufra cambios en el futuro, deberemos definir esta capa de forma que los posibles cambios en la implementación del acceso a datos no afecten a las capas superiores. Para ello haremos uso de todos aquellos elementos y patrones que nos permitan aislar a quien use nuestra capa de acceso a datos de su implementació concreta (interfaces, factoría, *nested exceptions*).

2.2. Preparación del entorno

Dado que vamos a acceder a una base de datos MySQL mediante JDBC, necesitaremos insertar el driver correspondiente entre las librerías de la aplicación. Copiaremos el driver al directorio `lib` del proyecto `proy-int-comun`, y añadiremos dicha librería al *build path* del proyecto.

En el directorio `db` tenemos los scripts `biblioteca.sql` y `datos.sql`, para crear las tablas de la base de datos e insertar una serie de datos de prueba, respectivamente. Podemos añadir a este directorio un fichero `build.xml` con las tareas de Ant que automatizan la ejecución de estos scripts. Este fichero será como se muestra a continuación:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="proy-int-db" default="initBD" basedir=".">

  <property name="driver" value="com.mysql.jdbc.Driver" />
  <property name="url" value="jdbc:mysql://localhost:3306/" />
  <property name="schema" value="biblioteca" />

  <path id="project.classpath">
    <pathelement
        location="../lib/mysql-connector-java-5.0.3-bin.jar" />
  </path>

  <target name="initBD">
    <sql driver="${driver}" url="${url}" userid="root"
        password="especialista" src="biblioteca.sql">
      <classpath refid="project.classpath" />
    </sql>
  </target>

  <target name="dataBD" depends="initBD">
    <sql driver="${driver}" url="${url}${schema}" userid="root"
        password="especialista" src="datos.sql">
```

```
      <classpath refid="project.classpath" />
    </sql>
  </target>

</project>
```

2.3. Operaciones de acceso a datos

En primer lugar deberemos decidir la lista de operaciones que ofrecerán nuestros DAOs. Tras un minucioso análisis de los requerimientos funcionales de nuestra aplicación, determinamos que las operaciones de acceso a datos que necesitaremos para implementar estas funcionalidades son las siguientes:

- Libros (operaciones CRUD)
  - *addLibro(libro)*: Crea un libro.
  - *delLibro(isbn)*: Elimina un libro.
  - *updateLibro(libro)*: Actualiza un libro.
  - *getLibro(isbn)*: Obtiene los datos de un libro.
  - *getLibros()*: Obtiene el listado de todos los libros.
- Usuarios (operaciones CRUD para usuarios y multas)
  - *addUsuario(usuario)*: Crea un usuario.
  - *delUsuario(login)*: Borra un usuario.
  - *updateUsuario(usuario)*: Actualiza un usuario.
  - *getUsuario(login)*: Obtiene los datos de un usuario.
  - *getUsuarios()*: Obtiene el listado de todos los usuarios.
  - *addMulta(multa)*: Crea una multa.
  - *delMulta(idMulta)*: Borra una multa.
  - *updateMulta(multa)*: Actualiza una multa.
  - *getMulta(idMulta)*: Obtiene los datos de una multa.
  - *getMultas(login, estado [activa | historico])*: Obtiene el listado de multas activas o históricas de un usuario dado.
- Operaciones (operaciones CRUD, listados y comprobaciones)
  - *addOperacion(operacion)*: Crea una operación
  - *delOperacion(idOperacion)*: Borra una operación.
  - *updateOperacion(operacion)*: Actualiza una operación.
  - *getOperacion(idOperacion)*: Obtiene los datos de una operación.
  - *getLibrosPrestados()*: Obtiene el listado de libros prestados.
  - *getLibrosReservados()*: Obtiene el listado de libros reservados.
  - *getLibrosDisponibles()*: Obtiene el listado de libros disponibles para reserva o préstamo.
  - *getLibros()*: Obtiene lista de todos los libros, junto a su operación activa, si la hubiera.
  - *isLibroDisponible(isbn)*: Comprueba si un libro está disponible para el préstamo o reserva.

- *countOperacionesActivas(login)*: Devuelve el número de operaciones activas de un usuario.

Nota

Podemos observar que no es necesario tener un DAO por cada tabla de la base de datos. En nuestro caso por ejemplo accederemos a las operaciones de las multas mediante el DAO de usuarios.

Es posible que cualquiera de estas operaciones produzca algún fallo, por ejemplo si el servidor de base de datos no se encuentra activo. Por lo tanto, deberemos tratar estos posibles errores mediante excepciones. Utilizaremos una excepción propia para nuestro DAO, a la que llamaremos `DAOException`, que heredará de la excepción base de nuestra aplicación (`BibliotecaException`).

El DAO estará en un paquete `es.ua.jtech.proyint.dao`. Por lo tanto, será en este paquete en el que ubicaremos la excepción para nuestro DAO. Crearemos en primer lugar esta excepción:

```
package es.ua.jtech.proyint.dao;

import es.ua.jtech.proyint.BibliotecaException;

/**
 * Manejo de excepciones para los DAO
 *
 * @author $Author$
 * @version $Revision$
 */
public class DAOException extends BibliotecaException {

  private static final long serialVersionUID =
                                    6158132279964132104L;

  /**
   * Constructor de la clase
   */
  public DAOException() {
    super();
  }

  /**
   * Constructor de la clase con mensaje
   * @param message Mensaje con la excepcion
   */
  public DAOException(String message) {
    super(message);
  }

  /**
   * Constructor de la clase con mensaje y causa
   * @param message Mensaje con la excepcion
   * @param cause Causa de la excepcion
   */
```

```
  public DAOException(String message, Throwable cause) {
    super(message, cause);
  }
}
```

De esta forma, nuestro DAO siempre devolverá `DAOException`, independientemente de la implementación concreta que estemos utilizando para acceder a los datos. Además, podemos ver que ofrecemos constructores de esta excepción a los que se proporciona como parámetro la excepción causante, con lo que podremos utilizar *nested exceptions*.

2.4. Interfaces para el DAO

Ahora deberemos crear las interfaces de cada DAO de nuestro sistema, con las operaciones que hemos determinado para cada uno de ellos. Dentro del paquete `es.ua.jtech.proyint.dao`, crearemos un subpaquete para cada DAO concreto (`libro`, `usuario`, y `operacion`), y en él situaremos la interfaz que dará acceso a este DAO.

Crearemos en primer lugar la interfaz para acceder al DAO de libros (`ILibroDAO`):

```
package es.ua.jtech.proyint.dao.libro;

import java.util.List;

import es.ua.jtech.proyint.dao.DAOException;
import es.ua.jtech.proyint.entity.LibroEntity;

/**
 * Interface para el DAO Libro
 *
 * @author $Author$
 * @version $Revision$
 */
public interface ILibroDAO {

  /**
   * Selecciona un libro de la BD
   * @param isbn Isbn del libro a seleccionar
   * @return Libro seleccionado
   * @throws DAOException
   */
  LibroEntity getLibro(String isbn) throws DAOException;

  /**
   * Añade un libro en la BD
   * @param libro Libro a añadir.
   * @throws DAOException
   */
  void addLibro(LibroEntity libro) throws DAOException;

  /**
   * El libro pasado por parámetro es borrado de la BD
   * @param isbn Libro a eliminar
   * @return Número de registros afectados por el borrado
   * @throws DAOException
```

```
   */
  int delLibro(String isbn) throws DAOException;

  /**
   * Devuelve una lista con todos los libros en la BD
   * @return Lista con todos los libros
   * @throws DAOException
   */
  List<LibroEntity> getLibros() throws DAOException;

  /**
   * Actualiza el valor de un libro
   * @param libro Libro a modificar.
   * @return Numero de libros actualizados
   * @throws DAOException
   */
  int updateLibro(LibroEntity libro) throws DAOException;
}
```

A continuación crearemos la interfaz `IUsuarioDAO` para dar acceso a los usuarios (y a sus multas):

```
package es.ua.jtech.proyint.dao.usuario;

import es.ua.jtech.proyint.dao.DAOException;
import es.ua.jtech.proyint.entity.UsuarioEntity;

/**
 * Interface para el DAO Usuario
 *
 * @author $Author$
 * @version $Revision$
 */
public interface IUsuarioDAO {

  /**
   * Recupera un usuario a partir de su PK
   * @param login Login del usuario a seleccionar
   * @return El usuario seleccionado o null si no existe
   * @throws DAOException
   */
  UsuarioEntity getUsuario(String login) throws DAOException ;

  /**
   * Añade un usuario a la BD
   * @param usuario Usuario a añadir en la BD
   * @throws DAOException
   */
  //void addUsuario(UsuarioEntity usuario) throws DAOException;

  /**
   * Borra el usuario de la BD
   * @param login Usuario a eliminar
   * @return Número de registros afectados en el borrado
   * @throws DAOException
   */
  //int delUsuario(String login) throws DAOException;
```

```
  /**
   * Modifica los datos de un usario de la BD
   * @param usuario Usuario a modificar
   * @return Número de registros afectados en la actualizacion
   * @throws DAOException
   */
  //int updateUsuario(UsuarioEntity usuario) throws DAOException;

  /**
   * Devuelve una lista con todos los usuarios en la BD
   * @return Lista con todos los usuarios
   * @throws DAOException
   */
  //List<UsuarioEntity> getUsuarios() throws DAOException;

  /**
   * Añade una multa al usuario relacionado con la multa
   * @param multa
   * @throws DAOException
   */
  //void addMulta(MultaEntity multa) throws DAOException;

  /**
   * Elimina una multa
   * @param idMulta Identificador de la multa a eliminar
   * @return Numero de registros afectados por el borrado
   * @throws DAOException
   */
  //int delMulta(String idMulta) throws DAOException;

  /**
   * Obtiene los datos de una multa
   * @param idMulta Identificador de la multa a consultar
   * @return Datos de la multa
   * @throws DAOException
   */
  //MultaEntity getMulta(String idMulta) throws DAOException;

  /**
   * Modifica una multa
   * @param multa Datos de la multa a modificar
   * @return Numero de registros afectados por la modificacion
   * @throws DAOException
   */
  //int updateMulta(MultaEntity multa) throws DAOException;

  /**
   * Obtiene las multas, activas o históricas, de un
   * determinado usuario
   * @param login Login del usuario
   * @param estado Estado de las multas a obtener (activa
   * o histórica)
   * @return Lista de multas
   * @throws DAOException
   */
  //List<MultaEntity> getMultas(String login, EstadoMulta estado)
throws DAOException;
}
```

Nota

Dejamos varias operaciones comentadas ya que por el momento no vamos a implementarlas. Todas ellas se implementarán más adelante.

Por último, introduciremos la interfaz `IOperacionDAO`, para acceder a los datos sobre las operaciones:

```
package es.ua.jtech.proyint.dao.operacion;

import java.util.List;

import es.ua.jtech.proyint.dao.DAOException;
import es.ua.jtech.proyint.entity.LibroEntity;
import es.ua.jtech.proyint.entity.OperacionActivaEntity;
import es.ua.jtech.proyint.entity.OperacionEntity;

/**
 * Interface para el DAO Operacion
 *
 * @author $Author$
 * @version $Revision$
 */
public interface IOperacionDAO {

  /**
   * Comprueba si un libro esta diponible para ser reservado o
   * prestado
   * @param isbn ISBN del libro a consultar
   * @return true si el libro está disponible, false en caso
   * contrario
   * @throws DAOException
   */
  boolean isLibroDisponible(String isbn) throws DAOException;

  /**
   * Obtiene una operación proporcionando su identificador
   * @param idOperacion Identificador de la operación a obtener
   * @return la operación solicitada, o null si no hay ninguna
   * operación con el identificador proporcionado
   * @throws DAOException
   */
  OperacionEntity getOperacion(String idOperacion)
                                              throws DAOException;

  /**
   * Obtiene el numero de operaciones que tiene activo un
   * determinado usuario
   * @param login Login del usuario
   * @return numero de operaciones activas
   * @throws DAOException
   */
  int countOperacionesActivas(String login) throws DAOException;

  /**
   * Añade una nueva operación a la BD.
   * @param operacion Operacion a insertar
```

```
    * @return El identificador de operacion generado. Este número
    * es autogenerado.
    * @throws DAOException
    */
   String addOperacion(OperacionEntity operacion)
                                                throws DAOException;

   /**
    * Elimina la operación cuyo id es el pasado por parámetro
    * @param idOperacion Id de la operación a eliminar.
    * @return Numero de registros afectados
    * @throws DAOException
    */
   int delOperacion(String idOperacion) throws DAOException;

   /**
    * Modifica la operación pasada por parámetro
    * @param operacion Datos de la operación a modificar.
    * @return Numero de registros afectados
    * @throws DAOException
    */
   int updateOperacion(OperacionEntity operacion)
                                                throws DAOException;

   /**
    * Devuelve una lista de libros prestados, con los atributos
    * del prestamo
    * @return Lista de libros prestados
    * @throws DAOException
    */
   List<OperacionActivaEntity> getLibrosPrestados()
                                                throws DAOException;

   /**
    * Devuelve una lista de libros reservados, con los atributos
    * de la reserva
    * @return Lista de libros reservados
    * @throws DAOException
    */
   List<OperacionActivaEntity> getLibrosReservados()
                                                throws DAOException;

   /**
    * Devuelve una lista de libros disponibles
    * @return Lista de libros disponibles
    * @throws DAOException
    */
   List<LibroEntity> getLibrosDisponibles() throws DAOException;

   /**
    * Devuelve una lista de libros, y en el caso de tener alguna
    * operacion asociada, con los atributos de la operacion
    * @return Lista de todos los libros, junto a sus operaciones
    * activas (si las hay)
    * @throws DAOException
    */
   List<LibroEntity> getLibros() throws DAOException;

}
```

Nota

Por el momento hemos determinado que estas serán las operaciones de acceso a datos que necesitará nuestra aplicación. Sin embargo, puede que más adelante durante el desarrollo de la misma necesitemos añadir nuevas operaciones a estos DAOs.

2.5. Implementación de los DAO

Una vez definidas las interfaces que nos darán acceso a nuestros DAO, deberemos crear una implementación de las mismas. En esta primera sesión crearemos una implementación de los DAO que nos dé acceso a los datos mediante JDBC.

En todas las operaciones de la implementación JDBC de nuestros DAOs será necesario obtener una conexión a la base de datos (objeto `Connection` de la API JDBC). Para evitar tener código repetido, crearemos una clase `FuenteDatosJDBC` que nos proporcione estas conexiones:

```
package es.ua.jtech.proyint.dao;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
 * Implementación JDBC del patrón singleton para acceder a la
 * base de datos
 *
 * @author $Author$
 */
public class FuenteDatosJDBC {

  private static Log logger =
              LogFactory.getLog(FuenteDatosJDBC.class);
  private static FuenteDatosJDBC me = new FuenteDatosJDBC();

  private FuenteDatosJDBC() {
    try {
      Class.forName("com.mysql.jdbc.Driver");
    } catch (ClassNotFoundException cnfe) {
      logger.fatal("No se encuentra el Driver de MySQL ", cnfe);
    }
  }

  public static FuenteDatosJDBC getInstance() {
    return me;
  }

  /**
   * Crea la conexion a la BD
   *
```

```
   * @return Conexion creada
   */
  public Connection createConnection() {
    Connection conn = null;

    try {
      conn = DriverManager
          .getConnection("jdbc:mysql://localhost/biblioteca",
                         "root", "especialista");
    } catch (SQLException sqle) {
      logger.fatal("No se ha podido crear la conexion", sqle);
    }

    return conn;
  }

}
```

Esta clase nos permitirá, además, cambiar posteriormente la forma en la que se obtienen las conexiones de forma sencilla. Si en lugar de crear nosotros una nueva conexión, queremos obtener las conexiones de una fuente de datos de nuestro servidor de aplicaciones, simplemente deberemos modificar `FuenteDatosJDBC`.

Comenzaremos viendo como ejemplo una implementación de `ILibroDAO`, que recibirá el nombre `LibroJDBCDAO`:

```
package es.ua.jtech.proyint.dao.libro;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import es.ua.jtech.proyint.dao.DAOException;
import es.ua.jtech.proyint.dao.FuenteDatosJDBC;
import es.ua.jtech.proyint.entity.LibroEntity;

/**
 * Implementación JDBC del DAO Libro
 *
 * @author $Author$
 */
public class LibroJDBCDAO implements ILibroDAO {

  private static Log logger =
LogFactory.getLog(LibroJDBCDAO.class.getName());

  /**
   * Método para seleccionar el libro cuyo Isbn se pasa por
   * parámetro.
```

```
   *
   * @see
es.ua.jtech.proyint.dao.libro.ILibroDAO#getLibro(java.lang.String)
   */
  public LibroEntity getLibro(String isbn) throws DAOException {
    LibroEntity libro = null;

    Connection conn = null;
    PreparedStatement st = null;
    ResultSet rs = null;

    try {
      String query = "select * from libro where isbn = ?";

      if (logger.isDebugEnabled()) {
        logger.debug(query + " [isbn=" + isbn + "]");
      }

      conn = FuenteDatosJDBC.getInstance().createConnection();
      st = conn.prepareStatement(query);
      st.setString(1, isbn);
      rs = st.executeQuery();

      if (rs.next()) {
        libro = new LibroEntity();
        libro.setIsbn(rs.getString("isbn"));
        libro.setTitulo(rs.getString("titulo"));
        libro.setAutor(rs.getString("autor"));
        libro.setNumPaginas(rs.getInt("numPaginas"));
        libro.setFechaAlta(rs.getDate("fechaAlta"));
      }
    } catch (SQLException sqle) {
      throw new DAOException("Error en el select de un libro",
          sqle);
    } finally {
      try {
        if (rs != null) {
          rs.close();
          rs = null;
        }
        if (st != null) {
          st.close();
          st = null;
        }
        if (conn != null) {
          conn.close();
          conn = null;
        }
      } catch (SQLException sqlError) {
        throw new RuntimeException("Error cerrando las conexiones",
            sqlError);
      }
    }

    return libro;
  }

  public void addLibro(LibroEntity libro) throws DAOException {
    Connection conn = null;
```

```
    PreparedStatement st = null;
    Date fechaAlta = null;

    try {
      String insert = "insert into libro(isbn, titulo, "
          + "autor, numPaginas, fechaAlta) values (?, ?, ?, ?, ?)";

      conn = FuenteDatosJDBC.getInstance().createConnection();
      st = conn.prepareStatement(insert);
      st.setString(1, libro.getIsbn());
      st.setString(2, libro.getTitulo());
      st.setString(3, libro.getAutor());
      st.setInt(4, libro.getNumPaginas());
      fechaAlta = libro.getFechaAlta();
      if (fechaAlta == null)
        st.setDate(5, null);
      else
        st.setDate(5, new java.sql.Date(fechaAlta.getTime()));
      st.executeUpdate();
    } catch (SQLException sqle) {
      throw new DAOException("Error en el insert de libro", sqle);
    } finally {
      try {
        if (st != null) {
          st.close();
          st = null;
        }
        if (conn != null) {
          conn.close();
          conn = null;
        }
      } catch (SQLException sqlError) {
        throw new RuntimeException("Error cerrando las conexiones",
            sqlError);
      }
    }
  }

  public int delLibro(String isbn) throws DAOException {
    Connection conn = null;
    PreparedStatement st = null;
    int numDel = 0;

    try {
      String delete = "delete from libro where isbn=?";

      conn = FuenteDatosJDBC.getInstance().createConnection();
      st = conn.prepareStatement(delete);
      st.setString(1, isbn);
      numDel = st.executeUpdate();
    } catch (SQLException sqle) {
      throw new DAOException("Error en el delete de libro", sqle);
    } finally {
      try {
        if (st != null) {
          st.close();
          st = null;
        }
        if (conn != null) {
```

```
        conn.close();
        conn = null;
      }
    } catch (SQLException sqlError) {
      throw new RuntimeException("Error cerrando las conexiones",
          sqlError);
    }
  }
  return numDel;
}

public List<LibroEntity> getLibros() throws DAOException {
  Connection conn = null;
  Statement st = null;
  ResultSet rs = null;
  List<LibroEntity> al = null;

  try {
    conn = FuenteDatosJDBC.getInstance().createConnection();
    st = conn.createStatement();
    rs = st.executeQuery("select * from libro");
    al = new ArrayList<LibroEntity>();
    LibroEntity libro = null;
    while (rs.next()) {
      libro = new LibroEntity();
      libro.setIsbn(rs.getString("isbn"));
      libro.setTitulo(rs.getString("titulo"));
      libro.setAutor(rs.getString("autor"));
      libro.setNumPaginas(rs.getInt("numPaginas"));
      libro.setFechaAlta(rs.getDate("fechaAlta"));
      al.add(libro);
    }
  } catch (SQLException sqle) {
    throw new DAOException("Error en el select de libros", sqle);
  } finally {
    try {
      if (rs != null) {
        rs.close();
        rs = null;
      }
      if (st != null) {
        st.close();
        st = null;
      }
      if (conn != null) {
        conn.close();
        conn = null;
      }
    } catch (SQLException sqlError) {
      throw new RuntimeException("Error cerrando las conexiones",
          sqlError);
    }
  }
  return al;
}

public int updateLibro(LibroEntity libro) throws DAOException {
  Connection conn = null;
  PreparedStatement st = null;
```

```
    int numUpdate = 0;

    try {
      String update = "update libro set isbn=?, titulo=?, "
          + "autor=?, numPaginas=?, fechaAlta=? where isbn=?";

      conn = FuenteDatosJDBC.getInstance().createConnection();
      st = conn.prepareStatement(update);
      st.setString(1, libro.getIsbn());
      st.setString(2, libro.getTitulo());
      st.setString(3, libro.getAutor());
      st.setInt(4, libro.getNumPaginas());
      st.setDate(5, new
                java.sql.Date(libro.getFechaAlta().getTime()));
      st.setString(6, libro.getIsbn());
      numUpdate = st.executeUpdate();
    } catch (SQLException sqle) {
      throw new DAOException("Error en el update de libro", sqle);
    } finally {
      try {
        if (st != null) {
          st.close();
          st = null;
        }
        if (conn != null) {
          conn.close();
          conn = null;
        }
      } catch (SQLException sqlError) {
        throw new RuntimeException("Error cerrando las conexiones",
            sqlError);
      }
    }
    return numUpdate;
  }

}
```

De esta clase destacamos el uso de:

- El *singleton* `FuenteDatosJDBC` para obtener las conexiones.
- *Nested exceptions* cada vez que ocurre un error en el acceso a los datos. Capturamos la excepción `SQLException` propia de JDBC, y devolvemos la nuestra propia (`DAOException`).
- `PreparedStatements` para así evitar el SQL *injection*.
- Log4J para escribir mensajes de error o depuración.

También podemos observar que hay ciertos aspectos mejorables en el código. Por ejemplo, cuando recuperamos los datos de un libro, tanto en `getLibro()` como en `getLibros()` utilizamos el mismo bloque de código:

```
libro = new LibroEntity();
libro.setIsbn(rs.getString("isbn"));
libro.setTitulo(rs.getString("titulo"));
libro.setAutor(rs.getString("autor"));
libro.setNumPaginas(rs.getInt("numPaginas"));
```

```
libro.setFechaAlta(rs.getDate("fechaAlta"));
```

Para evitar tener código repetido, podríamos refactorizar el código para extraer este bloque a un método independiente, de forma que desde ambas operaciones podamos reutilizar dicho método. Podemos utilizar el ménu *Refactor* de Eclipse para hacer esto. Seleccionaremos el bloque de operaciones que queremos extraer, pulsamos sobre ellas con el botón derecho, y en el menú *Refactor* seleccionamos la opción *Extract Method* .... Extraremos estas operaciones a un método privado `getLibro`, que a partir de un objeto `ResultSet` leerá los datos del libro y nos los devolverá dentro de un `LibroEntity`:

```
private LibroEntity getLibro(ResultSet rs) throws SQLException {
  LibroEntity libro;
  libro = new LibroEntity();
  libro.setIsbn(rs.getString("isbn"));
  libro.setTitulo(rs.getString("titulo"));
  libro.setAutor(rs.getString("autor"));
  libro.setNumPaginas(rs.getInt("numPaginas"));
  libro.setFechaAlta(rs.getDate("fechaAlta"));
  return libro;
}
```

Veremos que Eclipse ha modificado automáticamente todos los lugares del código en los que se realizaban las operaciones seleccionadas, y las ha sustituido por la llamada al nuevo método. Así, en `getLibro` tendremos:

```
      if (rs.next()) {
        libro = this.getLibro(rs);
      }
```

Y de la misma forma, en `getLibros` tendremos:

```
      while (rs.next()) {
        libro = this.getLibro(rs);
        al.add(libro);
      }
```

Ahora deberemos crear el resto de DAOs JDBC, que implementen todas las operaciones (que no estén comentadas) de las interfaces `IUsuarioDAO` e `IOperacionDAO`. Podemos utilizar `LibroJDBCDAO` como ejemplo a seguir.

Nota

Para la implementació de `OperacionJDBCDAO`, en muchas ocasiones necesitaremos especificar en la sentencia SQL el tipo o el estado de una operación. Para ello podemos servirnos de los elementos de las enumeraciones `TipoOperacion` y `EstadoOperacion` que tenemos definidas, evitando de esa forma introducir las cadenas con estos nombres directamente en el código.

2.6. Factorías de DAOs

Para independizar al resto de capas de nuestra aplicación de la instanciación de la

implementación concreta del DAO que se vaya a utilizar, realizaremos esta instanciación siempre mediante una factoría.

Crearemos la siguiente factoría en el paquete `es.ua.jtech.proyint.dao`:

```
package es.ua.jtech.proyint.dao;

import es.ua.jtech.proyint.dao.libro.ILibroDAO;
import es.ua.jtech.proyint.dao.libro.LibroJDBCDAO;
import es.ua.jtech.proyint.dao.operacion.IOperacionDAO;
import es.ua.jtech.proyint.dao.operacion.OperacionJDBCDAO;
import es.ua.jtech.proyint.dao.usuario.IUsuarioDAO;
import es.ua.jtech.proyint.dao.usuario.UsuarioJDBCDAO;

/**
 * Factoria que desacopla la implementación de los datos
 *
 * @author $Author$
 * @version $Revision$
 */
public class FactoriaDAOs {
  private static FactoriaDAOs me = new FactoriaDAOs();

  private FactoriaDAOs() {
  }

  public static FactoriaDAOs getInstance() {
    return me;
  }

  public ILibroDAO getLibroDAO() {
    return new LibroJDBCDAO();
  }

  public IOperacionDAO getOperacionDAO() {
    return new OperacionJDBCDAO();
  }

  public IUsuarioDAO getUsuarioDAO() {
    return new UsuarioJDBCDAO();
  }

}
```

Esta factoría, al igual que `FuenteDatosJDBC`, utiliza el patrón *singleton*, y nos permitirá obtener instancias de nuestros diferentes DAOs. Si posteriormente queremos cambiar el tipo de acceso a datos, simplemente deberemos crear una nueva implementación de los DAOs y modificar la factoría para que instancie esta nueva implementación.

2.7. Casos de prueba

Una vez hayamos implementado los DAOs, deberemos probarlos para comprobar que funcionan correctamente. Para ello podríamos definir una serie de casos de prueba de JUnit como los siguientes:

```
package es.ua.jtech.proyint.dao.libro;

import static org.junit.Assert.*;

import java.util.List;

import org.junit.Test;

import es.ua.jtech.proyint.dao.DAOException;
import es.ua.jtech.proyint.dao.FactoriaDAOs;
import es.ua.jtech.proyint.entity.LibroEntity;

public class ILibroDAOTest {

  @Test
  public void testGetLibro() throws DAOException {
    FactoriaDAOs factoriaDao = FactoriaDAOs.getInstance();
    ILibroDAO libroDao = factoriaDao.getLibroDAO();

    LibroEntity libro = libroDao.getLibro("0321180860");
    assertEquals(465, libro.getNumPaginas());
    assertEquals("Understanding SOA with Web Services",
        libro.getTitulo());
    assertEquals("Eric Newcomer and Greg Lomow", libro.getAutor());
  }

  @Test
  public void testGetLibroNulo() throws DAOException {
    FactoriaDAOs factoriaDao = FactoriaDAOs.getInstance();
    ILibroDAO libroDao = factoriaDao.getLibroDAO();

    LibroEntity libro = libroDao.getLibro("1321180860");
    assertNull(libro);
  }

  @Test
  public void testGetLibros() throws DAOException {
    FactoriaDAOs factoriaDao = FactoriaDAOs.getInstance();
    ILibroDAO libroDao = factoriaDao.getLibroDAO();

    List<LibroEntity> libros = libroDao.getLibros();
    assertEquals(12, libros.size());
  }

}
```

Sin embargo, aquí debemos tener en cuenta que todas estas pruebas dependen del estado actual de la base de datos (es una entrada más para nuestras operaciones), lo cual dificulta la realización de estas pruebas de unidad. En este caso, las pruebas se han implementado basándonos en el estado de la base de datos definido en el script `datos.sql`. Por este motivo, antes de ejecutar las pruebas deberemos asegurarnos de que la base se encuentra en este estado ejecutando dicho script.

Pero aquí no acaban todos los posibles problemas, ya que si estamos probando por ejemplo a hacer una inserción, el estado en el que quedará la base de datos tras hacer esto dependerá de si la inserción ha funcionado bien o mal, lo cual afectaría a las siguientes

pruebas. Para evitar que ocurra esto, deberíamos recuperar el estado inicial de la base de datos tras realizar cada prueba. Esto podemos hacerlo por ejemplo con librerías como DBUnit (primeros pasos con DBUnit, artículo sobre DBUnit).

Para simplificar, vamos a limitarnos a utilizar JUnit para operaciones que no modifican la base de datos, y que por lo tanto no producirán problemas de este tipo.

Al igual que hemos realizado pruebas con los métodos de consulta de `ILibroDAO`, vamos ahora a implementar casos de prueba para `IUsuarioDAO`, concretamente para su método `getUsuario(login)`, que es el único implementado por el momento.

Recuerda

Los casos de prueba de JUnit deben situarse en el mismo paquete que la clase probada, pero en diferente directorio de fuentes (en `test` en lugar de `src`).

2.8. Prueba de métodos de negocio

Una vez probados algunos de los métodos individuales de nuestros DAO, vamos a realizar una aplicación de prueba que implemente algún caso de uso de nuestra aplicación, apoyándonos en los diferentes DAO que tenemos. Esta aplicación se ejecutará en la consola, y nos mostrará un menú con dos opciones:

1. Obtener listado de libros
2. Realizar reserva

La primera de ellas se basará en la operación `getLibros()` de `IOperacionDAO` para obtener todos los libros de la base de datos, y mostrar los datos del libro junto a su estado actual (reservado, prestado o disponible).

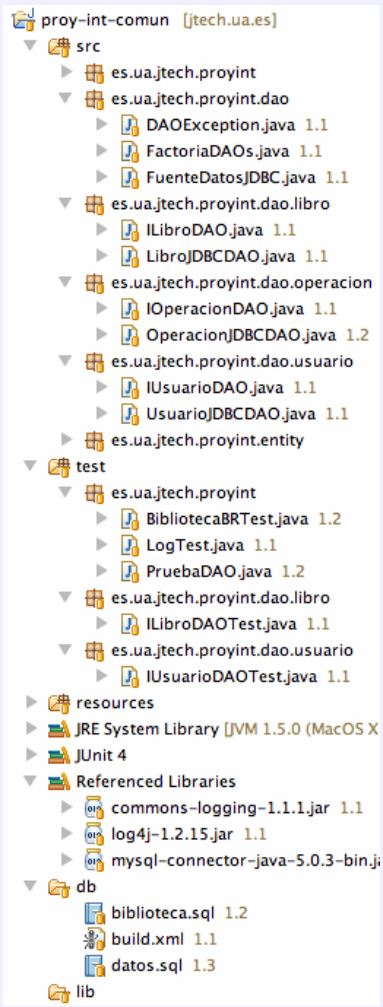
La segunda necesitará combinar diferentes llamadas a los DAO y a las reglas de negocio definidas, para implementar así la funcionalidad de este método de negocio.

Podemos descargar la implementación de esta aplicación `PruebaDAO` y añadirla a nuestra aplicación. De especial interés es observar su método `realizaReserva(isbn, login)`, en el que se implementa la lógica de negocio de esta operación, utilizando para ello las reglas de negocio y los diferentes DAOs definidos.

Nota

Dado que la clase `PruebaDAO` no pertenecerá a la aplicación final, y simplemente se utiliza como *driver* para realizar una prueba de la aplicación, la ubicaremos en el directorio de fuentes `test`. No obstante, más adelante podríamos coger de aquí el método `realizaReserva` para implementar la lógica de negocio de la aplicación.

2.9. A entregar



Proyecto a entregar

En esta sesión deberemos entregar el proyecto `proy-int-comun`, al que habremos añadido los DAO para acceder a la base de datos mediante JDBC. Sobre el proyecto de la anterior sesión, deberemos añadir:

- Driver JDBC de MySQL, y fichero `build.xml` para automatizar la ejecución de los *scripts* para crear la base de datos e insertar datos en ella.
- Excepción `DAOException` para tratar los errores ocurridos en los DAO de la aplicación.
- Interfaces de los diferentes DAOs: `ILibroDAO`, `IUsuarioDAO`, e `IOperacionDAO`.
- *Singleton* `FuenteDatosJDBC` para obtener la conexión a la base de datos.
- Implementaciones JDBC de los DAOs: `LibroJDBCDAO`, `UsuarioJDBCDAO`, y `OperacionJDBCDAO`.
- Factoría `FactoriaDAO` para instaciar los diferentes DAOs.
- Casos de prueba JUnit `ILibroDAOTest` e `IUsuarioDAOTest`.
- *Driver* `PruebaDAO` para probar la funcionalidad de reserva de libros.

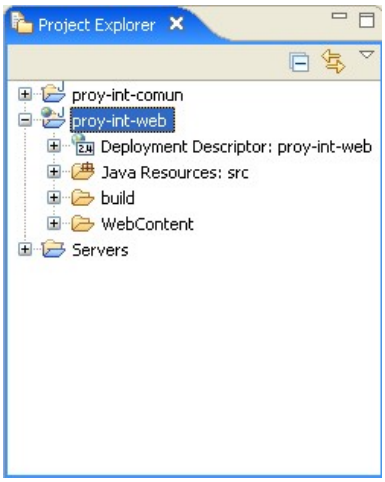
3. Aplicación web con servlets

3.1. Introducción

En esta sesión de integración incorporaremos la estructura de clases y los *DAOs* implementados en las sesiones anteriores, a una aplicación web dinámica que, mediante servlets que utilicen estas clases previas, aportarán todas las funcionalidades de los DAOs a la aplicación web.

3.2. Creación del proyecto web

En primer lugar, vamos a crear un proyecto web dinámico (*New - Dynamic Web Project*), llamado **`proy-int-web`**. Dejamos que cree las carpetas por defecto `WebContent` (donde pondremos nuestras páginas HTML y JSP, aparte de librerías, `web.xml` y demás), `src` (donde pondremos nuestros servlets y otras clases), y `build` (para compilar y unir toda la aplicación antes de desplegar o empaquetar). Como nombre de contexto, le pondremos también **`proy-int-web`**.

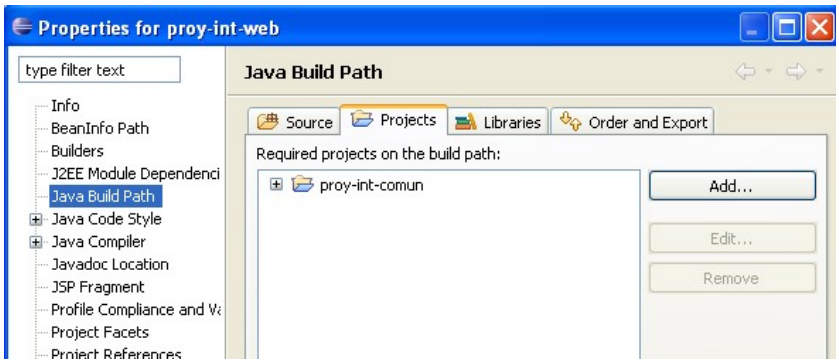


Los dos proyectos coexistiendo

Aparte, creamos **dos carpetas fuentes**, que utilizaremos más adelante, **`resources`** y **`test`**.

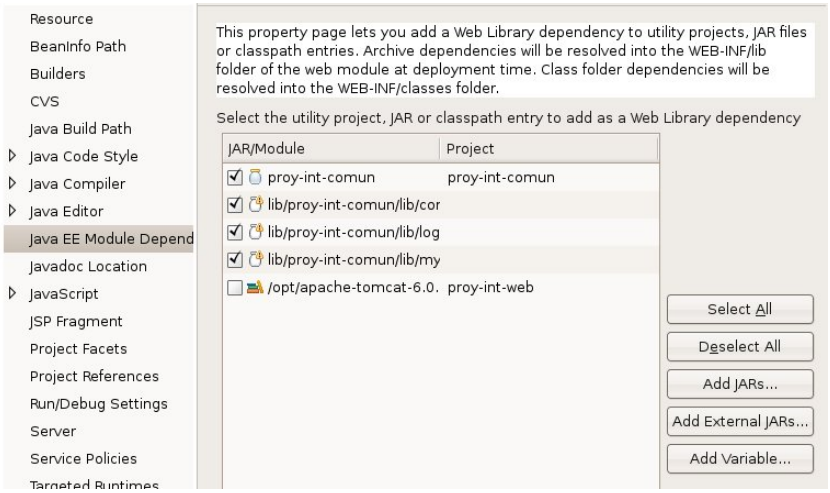
Después, tenemos que enlazar este proyecto web con el proyecto `proy-int-comun` que tenemos de sesiones anteriores. Para eso, seguimos los pasos:

- Vamos a las *Properties* del nuevo proyecto, y en su *Build Path* añadimos el proyecto `proy-int-comun` en la pestaña *Projects*.



Relacionar el proyecto Java con el proyecto web

- Volvemos a *Properties* del proyecto web, y en *Java EE Module Dependencies* marcamos el proyecto anterior, `proy-int-comun`.



Dependencias entre módulos

También debemos elegir la opción *Add JARs...*, y elegir los ficheros JAR de `proy-int-comun` (las librerías de logging y MySQL). De esta forma se copiarán automáticamente a `WEB-INF/lib` al desplegar la aplicación.

Paso 1

Crear el proyecto web según las instrucciones indicadas, enlazando con el proyecto anterior y creando las carpetas necesarias.

3.3. Configurar el pooling de conexiones

En las sesiones de integración anteriores obteníamos una `Connection` simple, a través de un `DriverManager`, en el método `createConnection` de la clase

es.ua.jtech.proyint.dao.FuenteDatosJDBC. Ahora que vamos a pasar a aplicación web, vamos a mejorar el rendimiento, accediendo al pooling de conexiones que nos ofrece Tomcat. Para ello, introduciremos los siguientes cambios:

- Creamos un fichero context.xml en la carpeta META-INF de nuestro proyecto web, donde definiremos las características del pooling:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Context>
    <Resource
        name="jdbc/biblioteca"
        type="javax.sql.DataSource"
        auth="Container"
        username="root"
        password="especialista"
        driverClassName="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost:3306/biblioteca?autoReconnect=true"/>

        <ResourceParams name="jdbc/biblioteca">

            <parameter>
                <name>maxActive</name>
                <value>20</value>
            </parameter>

            <parameter>
                <name>maxIdle</name>
                <value>5</value>
            </parameter>

            <parameter>
                <name>maxWait</name>
                <value>10000</value>
            </parameter>

        </ResourceParams>

</Context>
```

- Modificar el fichero web.xml de la aplicación para que referencie al recurso creado en el paso anterior. Para eso, añadimos estas líneas:

```
<resource-ref>
        <res-ref-name>jdbc/biblioteca</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
</resource-ref>
```

- Finalmente, ponemos entre comentarios el método createConnection de la clase es.ua.jtech.proyint.dao.FuenteDatosJDBC del proyecto proy-int-comun anterior. En su lugar, creamos un nuevo createConnection que utilizará el pooling de conexiones de Tomcat::

```
public Connection createConnection()
{
        Connection conn = null;
        try {
                Context initCtx = new InitialContext();
```

```
                Context envCtx = (Context)
initCtx.lookup("java:comp/env");
                DataSource ds =
(DataSource)envCtx.lookup("jdbc/biblioteca");
                conn = ds.getConnection();
        } catch (Exception e) {
                logger.fatal("No se ha podido crear la conexion",
e);
        }
        return conn;
}
```

Paso 2

Crear el pooling de conexiones, tocando los ficheros **context.xml**, **web.xml** y **FuenteDatosJDBC** como se ha indicado previamente.

3.4. Montando la web

3.4.1. Acciones a realizar

De la sesión anterior, tenemos unos objetos DAO que realizan una serie de acciones sobre usuarios (`IUsuarioDAO` y su implementación `UsuarioJDBCDAO`), sobre libros (`ILibroDAO` y su implementación `LibroJDBCDAO`), y sobre operaciones de usuarios con libros (`IOperacionDAO` y su implementación `OperacionJDBCDAO`). Estos objetos se apoyan para trabajar en sus respectivas entidades (*entities*) para ciertas operaciones, como por ejemplo, dar de alta un nuevo usuario, libro u operación, dado su objeto *entity* correspondiente (por ejemplo, pasamos un objeto `LibroEntity` para dar de alta un libro).

Lo que vamos a hacer en esta sesión es incorporar estas clases a la estructura de una aplicación web donde, mediante páginas HTML/JSP, indicaremos al servidor la operación que queremos realizar (por ejemplo, dar de alta un nuevo usuario, o listar todos los libros). Estas operaciones irán a parar al servlet encargado de atenderlas (veremos luego qué servlets implementar). Dicho servlet utilizará las clases auxiliares anteriores (los DAOs y las *entities*) para realizar la operación, y luego mostrar un resultado (una página HTML generada por él mismo, con el resultado de la operación).

Es **IMPORTANTE** tener en cuenta que, para que todo esto funcione de la forma más modular posible, necesitamos tener las clases de manejo de DAOs de la sesión anterior, ya que serán las que nos proporcionen el "puente" para acceder desde los servlets a las operaciones que queramos realizar. Por ejemplo, para sacar un listado de todos los libros de la biblioteca, haríamos, desde el servlet, lo siguiente:

```
ILibroDAO il = FactoriaDAOs.getInstance().getLibroDAO();
List<LibroEntity> lista = il.getLibros();
...// y luego recorrer la lista y mostrarla
```

Como puede verse, el esquema a seguir es que el servlet obtenga en un objeto el DAO (o DAOs) apropiados, y éstos le proporcionen los interfaces DAO que necesite (`ILibroDAO`, en este caso), y con dicho interfaz, llamar al método o métodos necesarios para completar la operación.

En esta sesión vamos a centrarnos en mostrar un listado de libros, y sobre el mismo, poder realizar reservas de los libros que puedan reservarse.

3.4.2. Listado de libros

En esta sesión todo va a partir de una página principal que generará un servlet `SeleccionarLibroServlet`, con un listado de libros. Dicho servlet lo tenéis parcialmente implementado aquí. Copiadlo en el paquete correspondiente (crear un paquete `es.ua.jtech.proyint.servlet.libro` en la carpeta `src` del proyecto web):

Listado de libros

Titulo	Operaciones
Data Access Patterns	Reservar
Patterns Of Enterprise Application Architecture	Reservar
Understanding SOA with Web Services	Reservar
Extreme Programming Explained - Embrace Change	Reservar
Agile Software Development	Reservar
Service-Oriented Architecture (SOA)	Reservar
Expert One-On-One J2EE Development Without EJB	Reservar
Practices of an Agile Developer	Reservar
Agile Retrospectives	Reservar
Beginning GIMP	Reservar
Java Persistence with Hibernate	Reservar
EJB 3 In Action	Reservar

Listado básico de libros

Deberéis:

- Mapear adecuadamente el servlet en el fichero `web.xml` (podríais, además de su mapeo, incluir temporalmente un mapeo adicional para que el servlet se cargue como página principal "/").
- Modificar el listado de libros para que, además del título, saque también el autor y el número de páginas, antes de la columna "Operaciones".
- Modificar la columna "Operaciones" para que sólo muestre el enlace "Reservar" en aquellos libros que puedan reservarse (los disponibles, según `IOperacionDAO.isLibroDisponible(...)`).
- Implementar la reserva de libros como se comenta a continuación

3.4.2.1. Probando los servlets

Cuando tengas modificado y listo el servlet `SeleccionarLibroServlet`, puedes probarlo con Cactus. Para ello aquí tienes una prueba ya hecha para incorporar al proyecto. Sigue estos pasos:

- En la carpeta de fuentes `test`, crea un paquete igual que el que hay en la carpeta `src` (`es.ua.jtech.proyint.servlet.libro`), y dentro crea un nuevo `Servlet Test Case` (*New - Java - JUnit - Servlet Test Case*, es importante que lo hagas así para que Eclipse importe él solo las librerías de Cactus al proyecto web).
- Una vez tengas la clase hecha, copia y pega dentro el código del caso de prueba ya hecho (borra el código fuente que genere Eclipse al crear la clase).
- Crea en la carpeta de fuentes `resources` un fichero `cactus.properties` y pega dentro este texto.
- Finalmente, añade al fichero `web.xml` del proyecto web este texto.

3.4.3. Reserva de libros

Para la reserva de libros, definiremos 2 servlets:

- Un primer servlet **`PrepararReservaServlet`**, que será llamado desde los enlaces de "Reservar" del listado de libros anterior (mapear el servlet adecuadamente). Recogerá como parámetro el ISBN del libro, y mostrará una página con un formulario para indicar el login del usuario a nombre del cual se realizará la reserva:
- Un segundo servlet **`RealizarReservaServlet`** que será llamado desde el formulario del paso anterior, y tomará como parámetros el ISBN del libro y el login del usuario que reserva. Tras realizar la reserva, mostrará una página con el resultado de la operación, y un enlace para poder volver al listado de libros.

Paso 3

Reimplementar el servlet `SeleccionarLibroServlet` (incorporando su caso de prueba ya hecho como se explica antes) e implementar los servlets `PrepararReservaServlet` y `RealizarReservaServlet` como se ha explicado anteriormente.

3.5. Configurar la seguridad

Como último paso, vamos a configurar seguridad declarativa en nuestra aplicación. Para eso seguiremos estos pasos:

1. Vamos a utilizar la propia tabla `usuario` para validar quién entra. En principio dejaremos pasar a cualquiera que esté registrado en esa tabla, y en futuras sesiones de integración veremos cómo, dependiendo de su perfil (rol) le podemos dejar hacer unas u otras cosas.

Para poder utilizar nuestra base de datos, debemos añadir un bloque `Realm` en el contexto de nuestra aplicación, que especifique que utilizaremos un `JDBCRealm` conectado a nuestra base de datos. Para ello, añadimos este bloque dentro de la etiqueta `Context` de nuestro fichero `META-INF/context.xml`:

```
<Realm  className="org.apache.catalina.realm.JDBCRealm"
driverName="com.mysql.jdbc.Driver"
connectionURL="jdbc:mysql://localhost:3306/biblioteca?autoReconnect=true"
connectionName="root" connectionPassword="especialista"
userTable="usuario" userNameCol="login" userCredCol="password"
userRoleTable="usuario" roleNameCol="tipoUsuario" />
```

2. En segundo lugar, debemos definir las páginas **login.jsp** y **error.jsp** en nuestra carpeta `WebContent`, con el formulario de validación y la página de error, como se vio en el módulo de Servidores Web.

Sugerencia

Es importante que ambas páginas tengan extensión `.jsp`, ya que de lo contrario Eclipse parece no admitirlas como páginas de login y error válidas. Además, en futuras sesiones será mejor tener páginas JSP a las que se pueda incorporar contenido dinámico.

3. Finalmente, queda añadir las líneas de configuración de seguridad al fichero `web.xml`.

   El bloque `security-constraint`:

```
<security-constraint>
        <web-resource-collection>
                <web-resource-name>...</web-resource-name>
                <url-pattern>...</url-pattern>
        </web-resource-collection>
        <auth-constraint>
                <role-name>...</role-name>
                <role-name>...</role-name>
                ...
        </auth-constraint>
</security-constraint>
```

   Hemos dejado en blanco (...) el contenido de cada etiqueta para que pongáis lo que corresponde. Deberemos especificar por separado los 4 roles para los que damos acceso (administrador, socio, profesor y bibliotecario). Para ello podemos añadir varias etiquetas `role-name` dentro del `auth-constraint`, como puede verse en el código anterior

   Bajo el bloque anterior, colocamos el bloque `login-config`:

```
<login-config>
        <auth-method>...</auth-method>
        ...
</login-config>
```

Se debe permitir la entrada a usuarios de los 4 roles, pero **sólo los usuarios de rol bibliotecario podrán**:

- Ver los enlaces de "Reservar" del listado de libros
- Reservar libros a través de los formularios de reserva

Es decir, los usuarios no-bibliotecarios sólo podrán ver el listado de libros con la información.

Para eso podéis ayudaros del método `request.isUserInRole(...)`, y mediante bloques `if...` definir qué parte de los servlets mostrar y cuáles no, o redirigir a otro servlet si el usuario no tiene permisos para usar el actual.

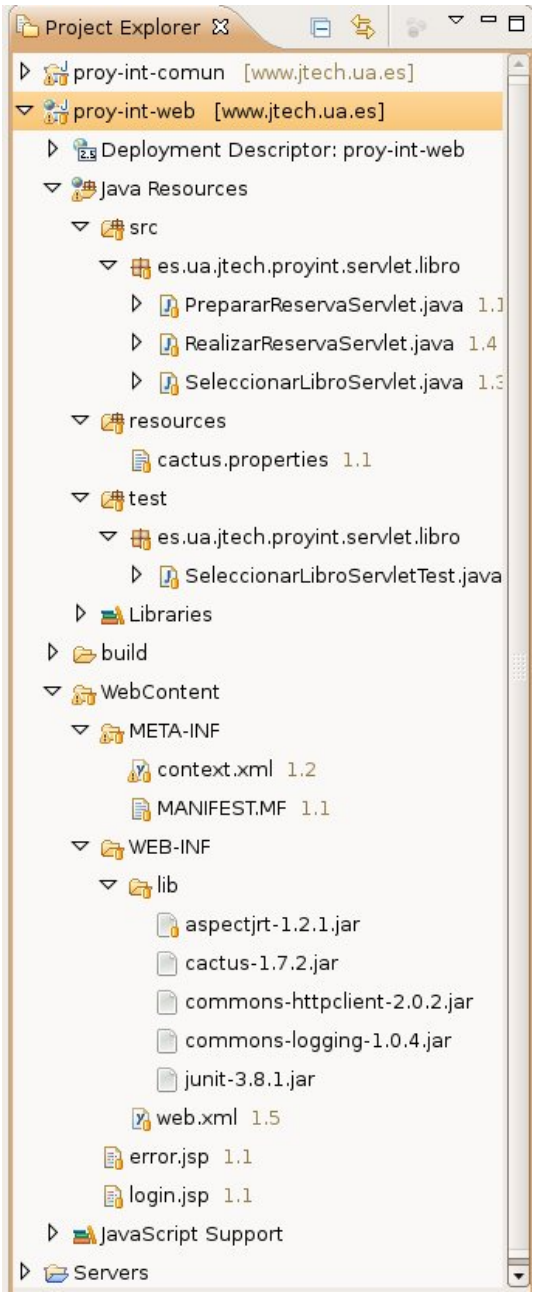
Paso 4
Configurar la seguridad de la aplicación modificando los ficheros **context.xml** y **web.xml** como se indica, y añadiendo las páginas **login.html** y **error.html**.

3.6. Sumario de tareas a realizar

En resumen, las tareas que deberéis completar para esta sesión de integración son:

1. Paso 1: configuración de la aplicación web
2. Paso 2: configuración del pooling de conexiones
3. Paso 3: servlets a implementar (`SeleccionarLibroServlet`, `PrepararReservaServlet`, `RealizarReservaServlet`)
4. Paso 4: configuración de seguridad mediante base de datos y formularios

Aquí tenéis un pantallazo de cómo os debería quedar el proyecto web:



Contenido del proyecto web

Como **ENTREGA**, exportad a un fichero ZIP los dos proyectos (`proy-int-comun` y `proy-int-web`) con los cambios realizados.

4. Aplicación web con servlets y JSPs

4.1. Introducción

En esta sesión de integración vamos a refactorizar la aplicación web dinámica desarrollada en la sesión anterior separando la presentación (JSPs) del procesamiento (servlets). Además, añadiremos la funcionalidad de *logout*, y una serie de funcionalidades para la gestión de los libros por parte de los bibliotecarios (alta, edición y borrado). También mejoraremos la interfaz de las diferentes páginas mediante la introducción de una serie de elementos comunes a todas ellas (cabecera, menú de opciones, y pie).

4.2. Separación en servlets y JSPs

En la sesión anterior implementamos una aplicación web con una serie de servlets que se encargaban tanto de procesar la información necesaria como de generar un documento HTML en el que se presenta el resultado que se le mostrará al usuario. Hemos visto que si bien los servlets son adecuados para procesar la información, a la hora de generar la presentación es más conveniente utilizar JSPs. El patrón *Modelo-Vista-Controlador* (MVC) proporciona esta separación, dejando la vista (presentación) en los JSP separada del controlador y del modelo, que estarán implementados mediante servlets y otros componentes y clases Java.

En esta sesión vamos a refactorizar la aplicación web separando la lógica de negocio (modelo-controlador) y presentación (vista), de forma que los servlets simplemente procesarán la información necesaria, y proporcionarán el resultado de este procesamiento a un JSP que se encargará de generar la presentación de dicho resultado.

Vamos a comenzar viendo como separar en servlets y JSPs las funcionalidades de obtener el listado de libros y realizar reserva implementadas en la sesión anterior. Para ello crearemos dos JSPs llamados `listadoLibros.jsp` y `preparaReserva.jsp`. El primero nos mostrará un listado de todos los libros (que habrá recibido desde el servlet `SeleccionarLibroServlet` como atributo de la petición) y junto a cada uno de ellos indicará si está reservado, prestado, o disponible. En caso de estar disponible nos mostrará un enlace para realizar la reserva del mismo, y al pulsar sobre este enlace se invocará el servlet `PrepararReservaServlet` que nos devolverá el contenido generado por `preparaReserva.jsp`, lo cual nos mostrará un formulario en el que indicar el login del socio para el cual queremos reservar el libro.

Nota

El listado de libros que estamos empleando se trata de un listado para usuarios bibliotecarios, en el que además de reservar libros, más adelante podrán darlos de alta, editarlos, o borrarlos. Por lo tanto, sólo este tipo de usuarios tendrá permiso para acceder a este listado.

Esta es exactamente la misma funcionalidad que implementamos en la sesión anterior, pero en este caso, al realizar la presentación mediante JSPs, será mucho más cómodo y limpio definir la apariencia del documento web. Dado que lo deseable es separar lógica de negocio y presentación, será conveniente evitar introducir código Java directamente en los JSPs (*scriptlets*) siempre que sea posible. Para ello utilizaremos librerías de *tags* como JSTL.

Para poder utilizar JSTL, deberemos descargar la librería de *tags Standard 1.1* de Jakarta y copiar los ficheros `jstl.jar` y `standard.jar` a la carpeta `/WEB-INF/lib` del proyecto web.

Debemos tener en cuenta que los JSP que creemos no deben poder ser accedidos nunca directamente desde el navegador en el cliente, ya que necesitarán ser invocados por un servlet que procese la información necesaria y se la pase al JSP mediante atributo de la petición. Para evitar que esto pudiese ocurrir, lo que haremos es guardar todos los JSP es un mismo directorio protegido mediante seguridad declarativa, al que nadie podrá acceder desde el lado del cliente (sólo se podrá acceder a él mediante `include` y `forward` desde dentro del servidor). Este directorio se llamará `jsp` e impediremos el acceso a su contenido introduciendo las siguientes etiquetas en el descriptor de despliegue (`web.xml`):

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>JSPs</web-resource-name>
        <url-pattern>/jsp/*</url-pattern>
    </web-resource-collection>
    <auth-constraint></auth-constraint>
</security-constraint>
```

4.3. Modificación de los servlets

Lo primero que deberemos hacer es modificar los servlets implementados en la sesión anterior, para que simplemente obtengan y procesen la información necesaria, pasándole el resultado generado a un JSP que será quien se encargue de realizar la presentación.

En primer lugar tenemos el servlet `SeleccionarLibroServlet`. Este servlet obtendrá el listado de todos los libros mediante el DAO, y le pasará un atributo `listaLibros` en el ámbito de la petición al JSP `/jsp/biblio/listadoLibros.jsp`. El código de este servlet quedará como se muestra a continuación:

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response)
             throws ServletException, IOException {
  try {
    if (!request.isUserInRole(TipoUsuario.bibliotecario.name())) {
      throw new BibliotecaException(
        "Solo los bibliotecarios pueden realizar esta operacion.");
    }
    IOperacionDAO iod =
        FactoriaDAOs.getInstance().getOperacionDAO();
    List<LibroEntity> lista = iod.getLibros();
```

```
    request.setAttribute("listaLibros", lista);

    RequestDispatcher rd = this.getServletContext()
        .getRequestDispatcher("/jsp/biblio/listadoLibros.jsp");
    rd.forward(request, response);
  } catch (BibliotecaException ex) {
    request.setAttribute("error",
        "Error obteniendo el listado de libros. " + ex);
    logger.error("Error obteniendo el listado de libros. " + ex);

    RequestDispatcher rd = this.getServletContext()
        .getRequestDispatcher("/jsp/error.jsp");
    rd.forward(request, response);
  }
}
```

Podemos observar que ahora el servlet ya no se encarga de imprimir el listado de libros, sino que simplemente hace un `forward` al JSP que se encargará de hacerlo. Lo mismo deberemos hacer con el servlet `PrepararReservaServlet`, que ya no deberá mostrar el formulario para introducir los datos de la reserva, sino que únicamente obtendrá los datos del libro a reservar y se los pasará al JSP `preparaReserva.jsp` que será el encargado de mostrar el formulario:

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response)
          throws ServletException, IOException {
  try {
    ILibroDAO ild = FactoriaDAOs.getInstance().getLibroDAO();
    LibroEntity libro = null;

    if (!request.isUserInRole(TipoUsuario.bibliotecario.name())) {
      throw new BibliotecaException(
        "Solo los bibliotecarios pueden realizar esta operacion.");
    } else if (request.getParameter("isbn") == null) {
      throw new BibliotecaException(
          "Debe proporcionarse un ISBN.");
    } else {
      libro = ild.getLibro(request.getParameter("isbn"));
      request.setAttribute("libro", libro);

      RequestDispatcher rd = this.getServletContext()
          .getRequestDispatcher("/jsp/biblio/preparaReserva.jsp");
      rd.forward(request, response);
    }
  } catch (BibliotecaException ex) {
    request.setAttribute("error",
      "Error al seleccionar libro para su reserva. " + ex);
    logger.error("Error al seleccionar libro para su reserva. " +
      ex);

    RequestDispatcher rd = this.getServletContext()
        .getRequestDispatcher("/jsp/error.jsp");
    rd.forward(request, response);
  }
}
```

Por último, nos queda el servlet `RealizarReservaServlet`. En este caso, si la reserva se

realiza con éxito, se hará una redirección a la página principal con el listado de libros, en la que el libro que acabamos de reservar ya aparecerá como reservado:

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response)
            throws ServletException, IOException {
  try {
    String isbn = request.getParameter("isbn");
    String login = request.getParameter("login");

    if (request.isUserInRole(TipoUsuario.socio.name())) {
      if(!request.getUserPrincipal().getName().equals(login)) {
        throw new BibliotecaException("Los socios solo pueden " +
            "realizar reservas para ellos mismos");
      }
    } else if (!request
          .isUserInRole(TipoUsuario.bibliotecario.name())) {
      throw new BibliotecaException("Solo los socios y " +
        "bibliotecarios pueden realizar una reserva.");
    }

    this.realizaReserva(login, isbn);
    response.sendRedirect(request.getContextPath());

  } catch (BibliotecaException ex) {
    request.setAttribute("error", "Error realizando la reserva. " +
      ex.getMessage());
    logger.error("Error realizando la reserva. " +
      ex.getMessage());

    RequestDispatcher rd = this.getServletContext()
      .getRequestDispatcher("/jsp/error.jsp");
    rd.forward(request, response);
  }
}
```

Destacamos también que en todos estos servlets, en caso de haber un error se hace un `forward` al JSP `error.jsp`. Este será un documento común que utilizaremos para mostrar cualquier error que se produzca en nuestra aplicación.

4.4. Elementos comunes

Antes de escribir los JSPs necesarios para nuestra aplicación, vamos a aprovechar para introducir una serie de elementos comunes a todas las páginas del sitio web. Estos elementos son la cabecera, el menú de opciones y el pie. Los definiremos en ficheros `jspf` independientes, y los incluiremos en todas las páginas mediante la directiva `include`.

El fragmento para la cabecera estará dentro del directorio `jsp` creado anteriormente y se llamará `cabecera.jspf`. Su contenido será el siguiente:

```
<div id="header">
<table width="100%">
<tr>
```

```
  <td><h1>Biblioteca jTech</h1></td>
</tr>
<tr>
  <td align="right">Usuario:
    ${pageContext.request.userPrincipal.name}</td>
</tr>
<tr>
  <td align="right"><a
    href="${pageContext.request.contextPath}/LogoutServlet">
      Cerrar sesión</a></td>
</tr>
</table>
</div>
```

Para el menú, crearemos el fragmento `jsp/biblio/menu.jspf` con el siguiente contenido:

```
<div id="sidebar">
<h2>Libros</h2>
<a href="${pageContext.request.contextPath}/LibroPreAddServlet">
  Alta</a><br/>
<a href=
  "${pageContext.request.contextPath}/SeleccionarLibroServlet">
  Listar</a>
</div>
```

En este caso lo ubicamos en el directorio `jsp/biblio` porque se trata del menú de opciones correspondiente a los usuarios de tipo bibliotecario.

Por último, el pie (`jsp/pie.jspf`) será como se muestra a continuación:

```
<div id="footer">
  <p>&copy; 2008-09 Especialista Universitario Java
    Enterprise</p>
</div>
```

4.5. Creación de los JSPs

Vamos a pasar ahora a crear los JSPs necesarios para las funcionalidades a implementar. En primer lugar crearemos el JSP `listadoLibros.jsp` dentro del directorio `jsp/biblio` (indicando así que pertenece a las funcionalidades de los bibliotecarios):

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
  pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">

<%@page import="es.ua.jtech.proyint.entity.LibroEntity"%>
<%@page import="es.ua.jtech.proyint.entity.TipoOperacion"%><html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Listado de libros</title>
<link rel="stylesheet" type="text/css" href="css/style.css"
        title="800px style" media="screen,projection" /></head>
```

```
</head>
<body>

<div id="wrap">

<%@include file="/jsp/cabecera.jspf" %>
<%@include file="/jsp/biblio/menu.jspf" %>

<div id="content">

<h2>Listado de libros</h2>

<c:choose>
<c:when test="${empty requestScope.listaLibros}">
  <p>No se han encontrado libros</p>
</c:when>
<c:otherwise>
  <table>
  <tr>
  <td><strong>Titulo</strong></td>
  <td><strong>Autor</strong></td>
  <td><strong>Paginas</strong></td>
  <td><strong>Estado</strong></td>
  <td><strong>Operaciones</strong></tr>

  <c:forEach items="${requestScope.listaLibros}" var="libro">

    <tr><td>${libro.titulo}</td>
    <td>${libro.autor}</td>
    <td>${libro.numPaginas}</td>

    <c:choose>
    <c:when test="${libro.activa==null}">
      <td>Disponible</td>
    </c:when>
    <c:when test="${libro.activa.tipo eq 'prestamo'}">
      <td>Prestado a ${libro.activa.usuario.login}</td>
    </c:when>
    <c:when test="${libro.activa.tipo eq 'reserva'}">
      <td>Reservado a ${libro.activa.usuario.login}</td>
    </c:when>
    </c:choose>

    <td>
      <c:if test="${libro.activa==null}"><a href=
"${pageContext.request.contextPath}/PrepararReservaServlet?isbn=${libro.isbn}">
        [Reservar]</a></c:if>
      <a href=
"${pageContext.request.contextPath}/LibroDelServlet?isbn=${libro.isbn}">
        [Borrar]</a>
      <a href =
"${pageContext.request.contextPath}/LibroPreUpdateServlet?isbn=${libro.isbn}">
        [Editar]</a>
    </td>
    </tr>

  </c:forEach>

  </table>
```

```
</c:otherwise>
</c:choose>

</div>

<%@include file="/jsp/pie.jspf" %>

</div>

</body>
</html>
```

En segundo lugar, dentro del mismo directorio (`jsp/biblio`) crearemos el JSP `preparaReserva.jsp`:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Solicitud de reserva</title>
<link rel="stylesheet" type="text/css" href="css/style.css"
        title="800px style" media="screen,projection" /></head>
</head>
<body>

<div id="wrap">

<%@include file="/jsp/cabecera.jspf" %>
<%@include file="/jsp/biblio/menu.jspf" %>

<div id="content">

<h2>Solicitud de reserva</h2>

<p><strong>Libro con '${requestScope.libro.titulo}'
  </strong></p>

<form
action="${pageContext.request.contextPath}/RealizarReservaServlet">
  <input type="hidden" name="isbn"
      value="${requestScope.libro.isbn}" />
  Realizar reserva a nombre de (introducir login):
  <input type="text" name="login"/><br/>

  <input type="submit" value="Reservar"/><br/>
</form>

</div>

<%@include file="/jsp/pie.jspf" %>

</div>

</body>
</html>
```

Por último, crearemos la página `error.jsp` dentro del directorio `jsp` (esta página será común para todas las funcionalidades):

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Error</title>
<link rel="stylesheet" type="text/css" href="css/style.css"
        title="800px style" media="screen,projection" /></head>
</head>
<body>

<div id="wrap">

<%@include file="/jsp/cabecera.jspf" %>
<%@include file="/jsp/biblio/menu.jspf" %>

<div id="content">

<h2>Error</h2>

<p>${requestScope.error}</p>

</div>

<%@include file="/jsp/pie.jspf" %>

</div>

</body>
</html>
```

Nota

En la sesión anterior se hizo una página `error.jsp` para tratar los errores que se produzcan en el login. Para evitar confusiones con la nueva página `error.jsp` que sirve para mostrar errores generales de la aplicación, renombraremos el fichero realizado en la anterior sesión a `errorLogin.jsp`, para que así quede claro que se utiliza sólo para el login.

Ahora ya podremos probar la aplicación, que deberá funcionar de forma similar a la anterior sesión, pero con la nueva interfaz que acabamos de crear.

4.6. Hoja de estilo

Podemos observar que los elementos de nuestros JSPs se encuentran organizados dentro de una serie de bloques `div` que representan los diferentes componentes de nuestros documentos web (cabecera, menú lateral, cuerpo, pie). Esta organización nos permitirá aplicar una hoja de estilo CSS para indicar el aspecto y la ubicación que tendrá cada uno de estos elementos. Podemos crear una hoja de estilo como la siguiente en un fichero

`/css/style.css` dentro de la carpeta web de nuestra aplicación:

```
/* 800px - An open source xhtml/css website template by Andreas
Viklund - http://andreasviklund.com . Free to use in any way and
for any purpose as long as the proper credits are given to the
original designer.

Version: 1.0, March 29, 2006 */

/******** General tags ********/
body{
font:76% Verdana,Tahoma,Arial,sans-serif;
color:#404040;
line-height:1.2em;
margin:0 auto;
padding:0;
}

a{
text-decoration:none;
color:#00344d;
font-weight:bold;
}

a:hover{text-decoration:underline;}
a img{border:0;}
p{margin:0 0 18px 10px;}
ul,ol,dl{font-size:1em; margin:2px 0 16px 16px;}
li {margin: 0px 0px 10px 0px;}
ul ul,ol ol{margin:4px 0 4px 35px;}
th {text-align:center; border:1px dotted #dadada;}


h1{
font-size:4.2em;
letter-spacing:-5px;
margin:0 0 30px 25px;
color:#00225A;
}

h1 a{text-transform:none; color:#00344d;}

h2{
font-size:1.4em;
color:#00344d;
border-bottom:4px solid #dadada;
padding:0 2px 2px 5px;
margin:0 0 10px 0;
letter-spacing:-1px;
}

h3{
font-size:1.2em;
font-weight:bold;
color:#00344d;
border-bottom:1px solid #dadada;
margin:10px 0 8px 0;
padding:1px 2px 2px 3px;
}
```

```
blockquote{
font-size:0.9em;
border:1px solid #dadada;
margin:20px 10px;
padding:8px;
}

/******** Main wrap ********/
#wrap{
color:#404040;
width:760px;
margin:10px auto;
padding:0;
}

#header{margin:0;}

#toplinks{text-align:right; padding:5px 2px 2px 3px;}

#slogan{
font-size:1.5em;
color:#808080;
font-weight:bold;
letter-spacing:-1px;
margin:15px 0px 20px 35px;
line-height:1.2em;
}

/******** sidebar ********/
#sidebar{
float:left;
width:130px;
margin:0 0 5px 0;
padding:1px 0 0 0;
}

#sidebar ul{
list-style:none;
font-size:0.9em;
margin:0;
padding:0 0 15px 10px;
}

#sidebar li{
list-style:none;
margin:0 0 1px 0;
padding:0;
}

#sidebar li a{
font-size:1.1em;
font-weight:bold;
padding:2px;
}

#sidebar ul ul{
margin:4px 0 3px 15px;
line-height:1.2em;
```

```
padding:0;
}

#sidebar ul ul li a{font-weight:normal;}
#sidebar h2{margin:3px 0px 8px 0px;}

/******** Content variations ********/
#content{
line-height:1.5em;
width:600px;
float:right;
text-align:left;
margin:0;
padding:0;
}

#contentalt{
line-height:1.5em;
width:600px;
float:left;
text-align:left;
padding:0;
margin-right:20px;
}

#content h3, #contentalt h3{margin:10px 0 8px;}

/******** Footer ********/
#footer{
clear:both;
text-align:right;
color:#808080;
font-size:0.9em;
border-top:4px solid #dadada;
margin:0 auto;
padding:8px 0;
line-height:1.6em;
}

#footer p{margin:0; padding:0;}
#footer a{color:#808080;}

/******** Various classes ********/
.box{
color:#ffffff;
font-size:0.9em;
background-color:#ff8e09;
border:1px solid #c8c8c8;
line-height:1.3em;
padding:5px 5px 5px 8px;
}

.box a{color:#f0f0f0;}
.left{float:left; margin:0 15px 4px 0;}
.right{float:right; margin:0 0 4px 15px;}
.textright{text-align:right;}
.readmore{text-align:right; margin:-10px 10px 12px 0;}

.compact{margin: 0 0 0 0;}
```

```
.center{text-align:center;}
.blue{color:#00344d;}
.big{font-size:1.3em;}
.small{font-size:0.8em;}
.bold{font-weight:bold;}

.clear{clear:both;}
.hide{display:none;}
.fade{color:#c8c8c8;}
.gray{color:#808080;}

.photo{
border:1px solid #bababa;
padding:2px;
background-color:#ffffff;
margin:6px 18px 2px 5px;
}
```

Si tenemos mapeado el servlet `SeleccionarLibroServlet` a la ruta `/` no se podrá acceder a la hoja de estilo, ya que el servlet estaría interceptando todas las peticiones al sitio web (siempre que no vayan a otro servlet mapeado a una URL). Por lo tanto, deberemos eliminar este mapeo, y en su lugar lo que haremos será incluir el servlet como *welcome file*, de forma que sea ejecutado por defecto al no especificar ninguna ruta dentro del contexto:

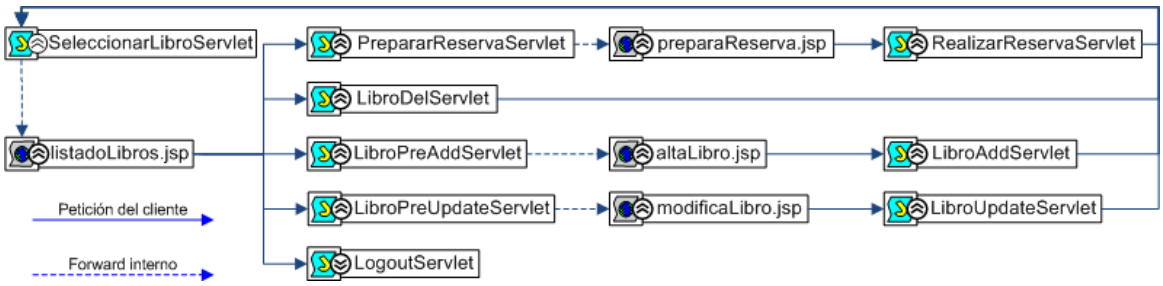
```
<welcome-file-list>
  <welcome-file>SeleccionarLibroServlet</welcome-file>
</welcome-file-list>
```

Listado de libros

4.7. Nuevas funcionalidades

Vamos ahora a añadir nuevas funcionalidades a la aplicación, siguiendo el mismo esquema de separación de responsabilidades entre servlets y JSPs que hemos visto para

las anteriores. Concretamente, añadiremos la posibilidad de dar de alta nuevos libros, o editar o borrar los libros ya existentes. Todas estas operaciones serán exclusivas de los usuarios de tipo bibliotecario.



Mapa de navegación de la web

Deberemos implementar los siguientes servlets en el paquete `es.ua.jtech.proyint.servlet.libro`:

- `LibroDelServlet`: Recibe un parámetro `isbn` con el ISBN del libro a borrar. Si el borrado se realiza de forma correcta, nos llevará de nuevo al listado de libros, donde veremos que el libro borrado ya no aparece en la lista.
- `LibroPreAddServlet`: Nos mostrará el formulario para introducir los datos del libro a dar de alta. Este formulario deberemos crearlo en un JSP de nombre `/jsp/biblio/altaLibro.jsp`.
- `LibroAddServlet`: Recibirá del formulario anterior los datos del libro a insertar, los introducirá en un objeto `LibroEntity`, y mediante el DAO los insertará en la base de datos. Una vez hecha la inserción, nos llevará a la página del listado de libros, donde aparecerá el nuevo libro.
- `LibroPreUpdateServlet`: Recibe un parámetro `isbn` con el ISBN del libro a editar. Obtendrá los datos de este libro de la base de datos y se los pasará mediante un atributo `libro` en el ámbito de la petición al JSP `modificaLibro.jsp` que nos mostrará un formulario que tendrá como valores por defecto los datos actuales del libro.
- `LibroUpdateServlet`: Este será el servlet invocado por el formulario anterior. Recibirá los nuevos datos del libro de dicho formulario y los actualizará en la base de datos. Una vez hecha la actualización nos llevará al listado principal donde veremos el libro modificado.

Si se produjese algún error en cualquiera de los servlets anteriores, se realizará un `forward` a `error.jsp` para mostrar la información del error producido.

Hemos visto que nuestro servlets necesitan dos JSPs con los formularios para el alta y la modificación de los libros. Estos JSPs son:

- `/jsp/biblio/altaLibro.jsp`: Contiene un formulario en el que deberemos introducir los datos del nuevo libro (isbn, titulo, autor, y número de páginas). Este formulario llamará al servlet `LibroAddServlet`.

Alta de libros

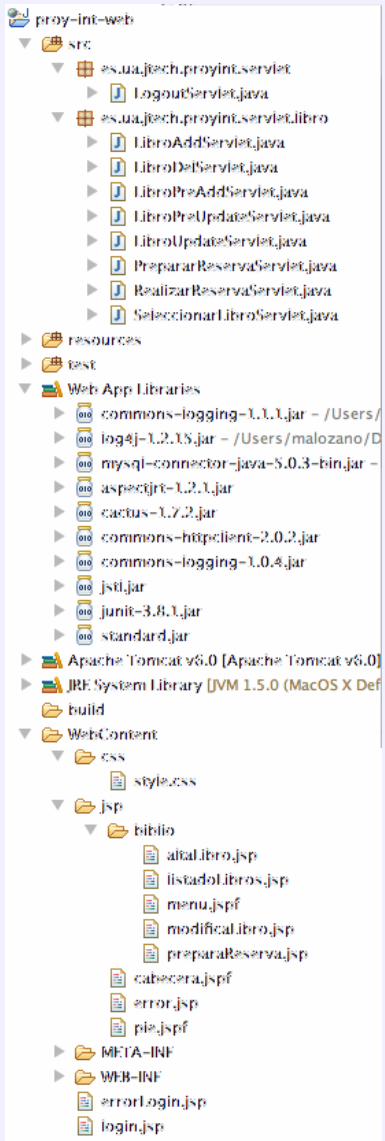
- `/jsp/biblio/modificaLibro.jsp`: Contiene un formulario para editar los datos del libro. Recibirá en el ámbito de la petición un atributo `libro` con los datos actuales del libro a editar. Mostrará estos datos como valores por defecto en los campos del formulario (isbn, titulo, autor y número de páginas). En este caso es importante que el campo ISBN sea de sólo lectura, ya que este es el identificador del libro y no debe ser modificado. Respecto a la fecha de alta, no es necesario permitir que se edite este campo.

Modificación de libros

Estos dos JSPs deberán tener todos los elementos comunes de la estructura de las páginas de nuestro sitio web (cabecera, menú y pie).

Por último, nos queda por implementar la funcionalidad de *logout*. Para ello crearemos un servlet `LogoutServlet` en el paquete `es.ua.jtech.proyint.servlet`. Lo único que deberá realizar este servlet es invalidar la sesión actual y llevarnos de nuevo a la página principal para que nos vuelva a pedir los datos de *login*.

4.8. Resumen



Proyecto a entregar

En esta sesión se deberá entregar el proyecto de integración actualizando el proyecto web con los siguientes puntos:

- Añadir las librerías `jstl.jar` y `standard.jar` al directorio `WEB-INF/lib` del proyecto web.
- Añadir seguridad declarativa para impedir el acceso al directorio `jsp` desde el cliente.
- Añadir los fragmentos de JSP con los elementos comunes de todas las páginas: cabecera (`jsp/cabecera.jspf`), menú (`jsp/biblio/menu.jspf`), y pie (`jsp/pie.jspf`).
- Modificar los servlets `SeleccionarLibroServlet`, `PrepararReservaServlet`, y `RealizarReservaServlet`.
- Añadir los servlets `LibroDelServlet`, `LibroPreAddServlet`, `LibroAddServlet`, `LibroPreUpdateServlet`, `LibroUpdateServlet`, y `LogoutServlet`.
- Añadir los JSPs `listadoLibros.jsp`, `preparaReserva.jsp`, `altaLibro.jsp`, `modificaLibro.jsp`, y `error.jsp`.
- Añadir la hoja de estilo `/css/style.css`.
- Eliminar el mapeo `/` del servlet `SeleccionarLibroServlet`, y en su lugar poner este servlet como *welcome file*.
- Renombrar la página `error.jsp` implementada en la sesión anterior por `errorLogin.jsp`.

5. Aplicación web con Struts

5.1. Introducción

En estas dos sesiones de integración vamos a convertir la aplicación web a MVC con Struts. Los objetivos básicos son:

- Convertir los servlets en acciones de Struts
- Validar datos de entrada: uso de actionforms y tags de formularios de Struts
- Internacionalizar la aplicación
- Añadir algunos casos de uso nuevos

5.2. Configuración de Struts en el proyecto

Antes de empezar a refactorizar el código, hay que importar las librerías de Struts y crear los ficheros de configuración. Seguid estos pasos:

1. **Copiad a la carpeta `WEB-INF/lib` los .jar** necesarios:
   - Jars para Struts 1.3 comprimidos en .zip. Son los incluidos en la distribución estándar de Struts más la librería struts-el, por si queréis usar el lenguaje de expresiones en las etiquetas de Struts. Commons-logging no se incluye porque ya lo tenéis en el proyecto, aunque es necesario para poder usar Struts.
   - **IMPORTANTE:** como se están añadiendo librerías al proyecto sobre la marcha, puede ser necesario actualizarlo (F5), cerrarlo y volverlo a abrir o hacer `Clean` para que Eclipse tome en cuenta los cambios.
2. **Cread el struts-config.xml**. Podéis usar por ejemplo este texto como plantilla:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration
1.2//EN"
"http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
<struts-config>
    <form-beans>
    </form-beans>

    <global-exceptions>
    </global-exceptions>

    <global-forwards>
    </global-forwards>

    <action-mappings>
    </action-mappings>

</struts-config>
```

3. **Modificad el web.xml para usar el servlet de Struts** Ahora todas las peticiones

acabadas en .do deben llegar al servlet de Struts. Por ello **necesitáis añadir un `<servlet>`. y un `<servlet-mapping>` para hacer el mapeo**

```
<servlet>
          <servlet-name>controlador</servlet-name>
<servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
          <init-param>
                    <param-name>config</param-name>
<param-value>/WEB-INF/struts-config.xml</param-value>
          </init-param>
          <load-on-startup>2</load-on-startup>
</servlet>
<servlet-mapping>
          <servlet-name>controlador</servlet-name>
          <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

4. Comprobar que Struts funciona correctamente. Provisionalmente, podéis crear una acción de prueba que simplemente se redirija a un JSP. En el struts-config se puede definir la acción

```
<action path="/test" forward="/index.jsp"/>
```

   y crear una página `index.jsp` provisional que contenga simplemente un "Hola Struts". Ahora podéis desplegar el proyecto y acceder a la URL `http://localhost:8080/proy-int-web/test.do`. Tras autentificarnos debe aparecer el index.jsp recién creado. Si no lo hace, revisad los pasos anteriores.
5. El *welcome-file* del `web.xml` ya no es necesario, ya que la página index.jsp actuará como página principal de la aplicación.

5.3. Cambios en la seguridad declarativa

Actualmente, todas las URL del proyecto están restringidas mediante autentificación declarativa. Esto es adecuado en la versión actual, pero sería problemático a partir de esta sesión, ya que va a haber operaciones, como la elección de idioma, que no pueden estar protegidas.

Vamos a cambiar la restricción a todas las URL (/*) por una restricción a un nuevo directorio, que vamos a llamar `auth`. Redirigiremos al usuario a una URL dentro de este directorio para forzar la autentificación. Por tanto, **en el `web.xml` hay que cambiar el `url-pattern` `/*` por `/auth/*`**

Ahora, vamos a forzar que se muestre la página de login colocando en index.jsp una redirección a una nueva página, que crearemos, llamada `/auth/auth.jsp`. En esta, a su vez, haremos una redirección a la "página inicial" de la aplicación, es decir, por el momento el servlet `SeleccionarLibroServlet`.

```
(fichero "/index.jsp")
```

```
<% response.sendRedirect("auth/auth.jsp"); %<
```

```
(fichero "/auth/auth.jsp")

<% response.sendRedirect("../SeleccionarLibroServlet"); %<
```

Nota:
Es importante diferenciar entre *redirect* y *forward*. Este último no nos sirve, ya que se salta la seguridad. Recordemos que los servlets hacen `forward` a los JSP en /jsp y en teoría estos no tienen acceso permitido a ningún rol.

Struts dispone de una etiqueta llamada `redirect` que nos permite hacer lo anterior pero de forma ligeramente más "elegante", ya que damos nombres lógicos a las URLs. **Una alternativa a lo anterior** sería

```
(fichero "/index.jsp")

<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic"
%>
<logic:redirect forward="auth"/>
```

```
(fichero "/auth/auth.jsp")

<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic"
%>
<logic:redirect action="/init"/>
```

Con este tag, una redirección a un forward es simplemente a otra URL, mientras que una redirección a una acción, como en el segundo caso, permite llamar a una acción de Struts. Por el momento no la hay, pero dentro de poco la introduciremos. En el `struts-config.xml` definiríamos

```
<global-forwards>
        <forward name="auth" path="/auth/auth.jsp"/>
</global-forwards>

<action-mappings>
        <action path="/init" forward="/SeleccionarLibroServlet"/>
        ...
```

Comprobad que al ejecutar el proyecto se sigue mostrando la pantalla de login y se puede ver el listado de libros entrando como bibliotecario.

5.4. Refactorización de un servlet en acción de Struts

La primera acción que vamos a implementar es la correspondiente al servlet. `SeleccionarLibroServlet`. Esta os puede servir como ejemplo para todos los servlets que no requieran validación de datos de entrada

5.4.1. Uso de constantes

Debemos evitar en la medida de lo posible el uso de literales de tipo cadena en nuestro código, sustituyéndolos por constantes. De esta forma evitaremos posibles inconsistencias y errores de tecleo. En Struts, cada vez que referenciemos un *forward* o una clave en un *.properties*, vamos a hacerlo a través de una constante y no poniendo la cadena literalmente en el código.

Para almacenar estas constantes, **definiremos un *interface* `Tokens`, dentro del nuevo paquete `es.ua.jtech.proyint.struts`**. Por el momento, en él definiremos dos constantes, para indicar respectivamente resultado correcto y erróneo de una acción:

```
package es.ua.jtech.proyint.struts;

public interface Tokens {
        // Forwards comunes
        String FOR_OK = "OK";
        String FOR_ERROR = "error";
}
```

5.4.2. Conversión del servlet en acción

Las acciones de libros las meteremos en el **nuevo paquete `es.ua.jtech.proyint.struts.acciones.libro`**. La acción correspondiente al servlet `SeleccionarLibroServlet` será la clase `LibroListTodosAccion`. Haremos que dicha clase herede de la clase `Action` de Struts y sobreescribiremos el método `execute`

Cambio de nombre

Aunque el servlet de listar libros se llamaba `SeleccionarLibroServlet` por "razones históricas", a partir de ahora vamos a cambiar el nombre de la acción para indicar que es un listado. Reservaremos el prefijo "Sel" para indicar que tratamos con un único objeto (libro, en este caso).

Del servlet básicamente nos interesa la variable *static* `logger` y casi todo el contenido del método `doGet`, salvo la llamada al `RequestDispatcher` del final. A continuación se muestra parte del código de la nueva acción (faltan los imports). Destacamos en negrita la parte que se ha tomado del servlet:

```
public class LibroListTodosAccion extends Action {

 private static Log logger =
LogFactory.getLog(LibroListTodosAccion.class);

 @Override
 public ActionForward execute(ActionMapping mapping, ActionForm
form,
```

```
                HttpServletRequest request, HttpServletResponse
response)
                throws Exception {
        try
        {
          if
(!request.isUserInRole(TipoUsuario.bibliotecario.name())) {
           throw new BibliotecaException("Solo los bibliotecarios
pueden" +
                                                 " realizar esta
operacion.");
          }
          IOperacionDAO iod =
FactoriaDAOs.getInstance().getOperacionDAO();
          List<LibroEntity> lista = iod.getLibros();
          request.setAttribute("listaLibros", lista);
          return mapping.findForward(Tokens.FOR_OK);

        } catch (BibliotecaException ex) {
          request.setAttribute("error", "Error obteniendo el
listado de libros. " + ex);
          logger.error("Error obteniendo el listado de libros. " +
ex);
          return mapping.findForward(Tokens.FOR_ERROR);
        }

 }
}
```

El código que chequea si el usuario actual es bibliotecario tampoco nos interesa porque en Struts lo vamos a controlar de manera declarativa, como veremos en el siguiente apartado.

En la acción todavía nos quedan dos literales de tipo cadena: "listaLibros" y "error". Sería recomendable cambiarlos por constantes dentro de `Tokens`. Para sistematizar el nombre de la constante

- Daremos el nombre RES_XXX (de *respuesta*) a un atributo en el que vayamos a almacenar algo
- Daremos el nombre MSJ_XXX a un mensaje de error. Recordemos que en Struts esto será una clave en un .properties, de la que nos ocuparemos más adelante.

`Tokens` quedará:

```
package es.ua.jtech.proyint.struts;

public interface Tokens {
        //Forwards
        String FOR_OK = "OK";
        String FOR_ERROR = "error";

        //Atributos
        String RES_LIBROS = "listaLibros";
        //Mensajes
        String MSJ_LIBRO_LIST_ERROR = "msj.libro.list.error";
```

```
}
```

5.4.3. Configuración en el struts-config

Hay que cambiar la acción index que teníamos para que llame a la de listar libros. Además hay que definir la acción de listar con sus *forwards*. El de error lo vamos a hacer global para que se pueda compartir con más acciones.

Mostramos en negrita los cambios introducidos

```
<global-forwards>
   <forward name="error" path="/jsp/error.jsp" />
</global-forwards>

<action-mappings>
   <action path="/init" forward="/libroListTodos.do"/>

   <action path="/libroListTodos"
type="es.ua.jtech.proyint.struts.acciones.libro.LibroListTodosAccion"
         roles="bibliotecario">
        <forward name="OK" path="/jsp/biblio/listadoLibros.jsp" />
   </action>
</action-mappings>
```

Obsérvese que la seguridad se controla declarativamente en este fichero mediante el atributo `roles` de la acción, por lo que **el código java de los antiguos servlets que chequea el rol del usuario actual ya no será necesario**.

Si se intenta acceder a una operación no autorizada se lanzará una excepción. Así, si en este momento se entra a la aplicación con un usuario no bibliotecario se producirá la excepción y se mostrará el mensaje de Tomcat en pantalla. Para evitar que el usuario final vea este mensaje, podemos indicarle a Struts que en su lugar muestre una página de error, en la sección "global-exceptions" del struts-config

```
<global-exceptions>
        <exception key="msj.noautorizado.error"
type="org.apache.struts.chain.commands.UnauthorizedActionException"
                path="/jsp/noAutorizado.jsp"/>
</global-exceptions>
```

El código anterior indica que cuando se produzca un intento de ejecutar una acción por un rol no autorizado (en Struts, una excepción `UnauthorizedActionException`) se salte a una página JSP. En dicha página podemos mostrar el error con las tags de Struts `<html:messages>` o `<html:errors>`. El texto del error estará en el fichero .properties de mensajes bajo la clave `msj.noautorizado.error`.

5.4.4. Configuración provisional como "pantalla" inicial

En `/auth/auh.jsp`, hay que poner una redirección a la acción recién creada. Si antes

usábamos el `response.sendRedirect` habrá que cambiarlo por `../init.do`. En caso de haber usado la tag `redirect` de struts, sería a `/init`.

Ahora podéis probar a entrar en la aplicación como bibliotecario para ver si sigue funcionando. También deberíais entrar como otro tipo de usuario para comprobar que se muestra el error.

5.4.5. Cambios en los JSP

Los enlaces al antiguo servlet en los JSP deben cambiarse por enlaces a la nueva acción. En concreto, en este caso debemos cambiar el enlace que aparece en `/jsp/biblio/menu.jspf`. Es recomendable usar la etiqueta `link` de Struts, que nos permite especificar el nombre lógico de la acción en lugar de la URL física. Además eliminamos la necesidad de tener en cuenta el `request.contextPath`:

```
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html"
%>
<div id="sidebar">
<h2>Libros</h2>
<a
href="${pageContext.request.contextPath}/LibroPreAddServlet">Alta</a><br/>
<html:link action="/libroListTodos">Listar</html:link>
</div>
```

5.4.6. Gestión de mensajes de error

Conviene usar el soporte de Struts para gestión de mensajes de error, ya que ofrece ventajas como la internacionalización. Por ello vamos a reemplazar el atributo de request llamado "error" que usábamos hasta el momento por código de Struts. En la acción, cambiamos la parte en la que se captura la excepción:

```
} catch (BibliotecaException ex) {
  ActionMessages am = new ActionMessages();
  am.add(ActionMessages.GLOBAL_MESSAGE,
                    new ActionMessage(Tokens.MSJ_LIBRO_LIST_ERROR,
ex));
  saveErrors(request, am);
  logger.error("Error obteniendo el listado de libros. " + ex);
  return mapping.findForward(Tokens.FOR_ERROR);
}
```

Nótese que el mensaje de error tiene un parámetro, que es la excepción producida. Nos queda definir el propio mensaje de error. Antes de nada, hay que **crear el fichero .properties en la carpeta fuente *resources* y referenciarlo en el struts-config**. Al final del archivo, antes de la etiqueta de cierre de </struts-config>, incluir la línea

```
<message-resources parameter="mensajes"/>
```

En el fichero `mensajes.properties` definiremos el único mensaje de error que tenemos por el momento:

```
msj.libro.list.error=Error en el listado de libros: {0}
```

Nos falta cambiar el JSP de error, usando las tags de struts para mostrar el mensaje. Cambiaremos `/jsp/error.jsp`

```
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html"
%>
...
<p>
   <html:errors/> <!-- sustituye al ${requestScope.error} de antes
-->
</p>
...
```

Aclaración

Al poner html:errors estamos mostrando todos los mensajes de error juntos, lo que nos impide formatear con HTML cada mensaje de error por separado. En este caso solo hay un mensaje, así que no importa. Recordemos que si usamos html:messages vamos iterando por cada mensaje y podemos meter HTML por enmedio, pero la etiqueta es más tediosa de usar.

Comprobad que el mensaje de error se muestra adecuadamente. Tal y como está la aplicación es difícil probarlo, ya que el error únicamente se podría producir por una excepción en el `getLibros` del DAO `IOperacionDAO`. Aunque sea un poco extraño, colocad un `throws new BibliotecaException` después de la llamada al DAO, de manera provisional,para verificar la gestión de errores.

Una vez hechos todos los cambios, el servlet `SeleccionarLibroServlet` ya no es necesario y podemos eliminar tanto su código como su mapeo en el `web.xml`. Recordad que teníais también un caso de prueba para el servlet, que también habrá que eliminar.

Por supuesto, **tenéis que convertir el resto de servlets en acciones de Struts, siguiendo el modelo anterior**. El caso de alta de libro es un caso "especial", ya que hay que validar datos de entrada.

5.5. Acción de login

Vamos a introducir una acción de login, que no existía hasta el momento en forma de servlet. De este modo, vamos reestructurando la aplicación para mostrar distintas pantallas según el tipo de usuario. La acción distinguirá este tipo y se redirigirá a un grupo de JSPs distintos para cada caso:

- Carpeta `/jsp/admin` para usuarios `administrador`
- Carpeta `/jsp/biblio` para usuarios `bibliotecario`

- Carpeta `/jsp/socio` para usuarios `profesor` o `socio`

El código del método execute sería como sigue (aquí tenéis el código completo de la acción). Previamente tendremos que haber creado las constantes correspondientes en la clase `Tokens`

```
@Override
public ActionForward execute(ActionMapping mapping, ActionForm
form,
                HttpServletRequest request, HttpServletResponse
response)
                throws Exception {

        ActionForward forward = null;

        String login = request.getUserPrincipal().getName();

        IUsuarioDAO udao =
FactoriaDAOs.getInstance().getUsuarioDAO();
        UsuarioEntity usuario = udao.getUsuario(login);

        logger.debug("Login con usuario " + usuario);

        if
(request.isUserInRole(TipoUsuario.administrador.toString())) {
                forward = mapping.findForward(Tokens.FOR_ADMIN);
        } else if (request
.isUserInRole(TipoUsuario.bibliotecario.toString())) {
                forward = mapping.findForward(Tokens.FOR_BIBLIO);
        } else if
(request.isUserInRole(TipoUsuario.socio.toString())
                        ||
request.isUserInRole(TipoUsuario.profesor.toString())) {
                if (usuario.getEstado() == EstadoUsuario.moroso) {
                        logger.debug("Multas de " + login + "-> " +
usuario.getMultas());
                        forward =
mapping.findForward(Tokens.FOR_MOROSO);
                } else {
                        forward =
mapping.findForward(Tokens.FOR_SOCIO);
                }

        } else {
                forward = mapping.findForward(Tokens.FOR_ERROR);
        }

        return forward;
}
```

Por supuesto, también tendremos que mapear la acción en el struts-config, y cambiar el `init` anterior por la llamada a la acción de login:

```
<action path="/init" forward="/login.do"/>
```

```
<action path="/login"
type="es.ua.jtech.proyint.struts.acciones.LoginAccion">
        <forward name="indexAdmin" path="/usuarioList.do"/>
        <forward name="indexBiblio" path="/libroListTodos.do"/>
        <forward name="indexSocio" path="/libroListSocio.do"/>
        <forward name="indexMoroso" path="/jsp/socio/moroso.jsp"/>
</action>
```

Nótese que, **de los posibles resultados de la acción, el único que tenemos implementado por el momento es el del usuario "bibliotecario"**, listar todos los libros

5.6. Alta de libros

El alta de libro es un caso especial, ya que se deben validar los datos de entrada porque estos se introducen en un formulario. La refactorización de servlet en acción de Struts será igual que la vista antes, pero además, habrá que:

- Definir un *actionform* para recibir los datos
- En el código de la acción, copiar los datos del *actionform* al *entity*.
- Configurar la validación usando el *plugin* `validator`
- Cambiar el formulario HTML de introducción de datos por etiquetas de Struts
- Verificar que no se intenta realizar dos veces la operación

5.6.1. Configuración del plugin "validator"

Aseguráos de que Struts está preparado para usar `validator`, es decir que en WEB-INF existen los ficheros validation.xml y validator-rules.xml. Para este último podéis usar el incluido en la distribución estándar de struts, mientras que el primero debe contener únicamente

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE form-validation PUBLIC
   "-//Apache Software Foundation//DTD Commons Validator
   Rules Configuration 1.0//EN"
   "http://jakarta.apache.org/commons/dtds/validator_1_0.dtd">
<form-validation>
</form-validation>
```

Además en el struts-config se debe cargar el plugin validator, esto se consigue con la etiqueta <plug-in> que debemos colocar al final del archivo de configuración.

```
        ...
    <plug-in
className="org.apache.struts.validator.ValidatorPlugIn">
        <set-property property="pathnames"
value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml"/>
    </plug-in>
</struts-config>
```

5.6.2. Uso de ActionForm

El uso de ActionForms implica una serie de cambios con respecto a las acciones que no los necesitan:

- Hay que crear el propio actionform: las clases se deben crear en el paquete `es.ua.jtech.proyint.struts.actionforms`. Por el momento, únicamente habrá una clase `LibroForm` para los datos del libro. Recordemos que las propiedades deben ser todas de tipo String, para que puedan acomodar cualquier valor que introduzca el usuario, incluso datos de tipo incorrecto.
- Hay que referenciar el actionform en el `struts-config`, con la etiqueta `form-bean` y además configurar la accion en que se va a usar (atributos `name`, `validate`, `scope` e `input`.
- Hay que definir la validación en el `validation.xml`: al dar de alta un libro, el isbn debe tener como mínimo 10 caracteres, y el número de páginas debe ser un entero positivo. Son obligatorios los campos isbn, título y autor.
- En el código de la acción, hay que copiar los datos del actionform al objeto entity. Para ello nos ayudaremos de la librería *beanutils*, cuyo uso se describe en el apartado siguiente

5.6.3. La librería beanutils

En lugar de copiar los datos manualmente campo a campo del actionform al entity podemos usar una librería de jakarta denominada *beanutils*. Esta librería contiene métodos de utilidad para el trabajo con beans. Entre ellos ofrece un método que permite copiar automáticamente los campos de un objeto en otro, si tienen el mismo nombre. Beanutils usa *reflection* para hacer la copia y puede convertir automáticamente los tipos.

```
public ActionForward execute(ActionMapping mapping, ActionForm
form,
                HttpServletRequest request, HttpServletResponse
response)
                throws Exception {

        LibroEntity libro = new LibroEntity();
        LibroForm libroForm = (LibroForm) form;
        //copiar propiedades (destino, origen)
        BeanUtils.copyProperties(libro, libroForm);
        ...
}
```

Aunque la librería beanutils no se incluye en la distribución estándar de Struts, la tenéis entre los JARs que os habéis bajado al principio de la sesión, así que podéis usarla en el proyecto sin problemas.

5.7. Actualización de libros

La actualización de libros es similar al alta. También usaremos un actionform, pero en este caso para copiar los datos desde el propio actionform al formulario HTML. Por tanto, su sitio es la acción de "PreUpdate", la que nos lleva a ver el formulario HTML:

```
<action path="/libroPreUpdate"
type="es.ua.jtech.proyint.struts.acciones.libro.LibroPreUpdateAccion"
   roles="bibliotecario" name="libroForm" scope="request"
validate="false">
   <forward name="OK" path="/jsp/biblio/modificaLibro.jsp" />
</action>
```

Obsérvese que el actionform tiene `validate=false` porque no es necesario validar los datos, ya que proceden de la base de datos. La acción de "PreUpdate" los obtiene gracias al DAO y los coloca en el actionform, copiándolos del entity libro gracias a BeanUtils. Se muestra a continuación el fragmento del `execute` de la acción que realiza la tarea descrita:

```
String isbn = request.getParameter("isbn");

try {
        if(isbn==null) {
          throw new BibliotecaException("Se debe especificar un
ISBN.");
        }

        //Obtener el libro de la BD
        ILibroDAO lDao = FactoriaDAOs.getInstance().getLibroDAO();
        LibroEntity libro = lDao.getLibro(isbn);

        if(libro==null) {
          throw new BibliotecaException("No se puede obtener el
libro indicado.");
        }
        //form es el parámetro ActionForm del método execute
        LibroForm libroForm = (LibroForm) form;
        //copiar el libro en el actionform
        BeanUtils.copyProperties(libroForm, libro);
        result = mapping.findForward(Tokens.FOR_OK);
} catch (BibliotecaException ex) {
        ...
```

Al copiar los datos en el ActionForm que recibe el execute como parámetro, conseguimos que Struts los pase a los campos del formulario de modificación de libro (siempre que usemos las tags HTML de Struts).

5.8. Nuevos casos de uso

Implementar casos de uso adicionales para listar libros disponibles, prestados y reservados. En el DAO de operación ya tenéis implementados dichos métodos, por lo que

el trabajo a hacer será:

- Crear una nueva acción por cada caso de uso: `LibroListPrestadosAccion`, `LibroListReservadosAccion` y `LibroListDisponiblesAccion`, todas en el paquete `es.ua.jtech.proyint.struts.acciones.libro`.
- Crear un nuevo JSP para cada acción, dentro de la carpeta `jsp/biblio`, con los mismos nombres que las clases de las acciones. Aunque podríamos procesarlas todas con un único JSP, tendrán menos código Java si cada uno se ocupa de un caso distinto.
- Configurar las acciones en el strut-config.xml
- Modificar el menú de bibliotecario `jsp/biblio/menu.jspf` para acomodar las nuevas opciones. Agrupad en el menú por un lado las operaciones de gestión (por el momento únicamente alta) y por otro los listados.

5.9. Internacionalización

La aplicación debe estar internacionalizada al menos para castellano e inglés, y para todas las opciones del bibliotecario. **Se deben traducir tanto las pantallas como los mensajes de error**.

Requisitos de la implementación:

- La acción para cambiar el idioma se implementará en la clase `CambiaIdiomaAccion` en el paquete `es.ua.jtech.proyint.struts.acciones`. Al menos en la página de login se incluirá un enlace a dicha acción, pasándole como parámetro el código de idioma.
- Los mensajes internacionalizados se colocarán en el correspondiente .properties de la manera más sistemática posible, para que sean sencillos de localizar. Intentad seguir lo más fielmente posible una estructura como la del siguiente ejemplo:

```
# opciones de menu
menu.usuario.List = Listar todos los usuarios
menu.libro.List = Listar todos los libros
menu.libro.Sel = Ver los datos de un libro
menu.libro.Add = Dar de alta un libro
menu.titulo = Menú principal

#index
index.biblioteca = Biblioteca JTECH
#cabecera
cabecera.cerrarSesion = Cerrar sesion
#pie

#datos de un libro
libro.ISBN = ISBN
libro.titulo = Título
libro.autor = Autor
libro.numPaginas = Nº págs
libro.fechaAlta = Fecha de Alta
```

```
#pantallas sobre libros
libro.crearLibro = Crear libro
libro.datosLibro = Datos del libro
```

6. Aplicación web con JPA

6.1. Introducción

En esta sesión de integración vamos a refactorizar la aplicación para integrar la gestión de la capa de persistencia con JPA. El aspecto de la aplicación no cambiará nada, porque se seguirán utilizando las mismas interfaces de la capa de datos. Modificaremos sólo el proyecto común donde se encuentra esta capa de datos.

El trabajo va a consistir básicamente en dos partes. En la primera instalaremos las librerías necesarias de JPA en el proyecto común y realizaremos el mapeado de las entidades a las tablas de la base de datos, añadiéndoles las oportunas anotaciones de JPA a las clases Java. Comprobaremos que todo funciona correctamente con algunos tests de JUnit.

Una vez mapeadas las entidades, deberemos refactorizar los DAO, para que usen JPA en lugar de JDBC. Lo haremos con las tres clases DAO implementadas en JPA, desarrollando las clases `LibroJPADAO`. Modificaremos los tests de Junit para que prueben estos DAO y por último (cuando hayamos comprobado que todo funciona en el proyecto común) haremos las modificaciones necesarias para que el proyecto web utilice esta nueva capa de persistencia.

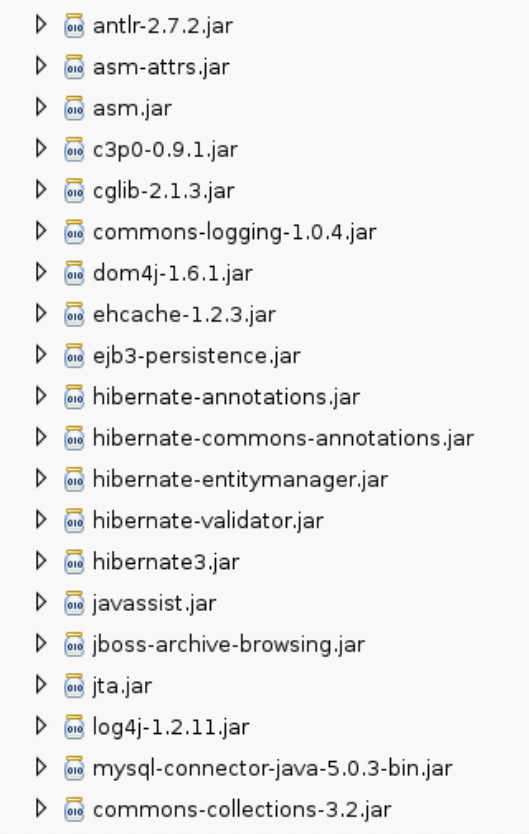
6.2. Instalación de las librerías de Hibernate

Vamos a comenzar instalando las librerías de Hibernate JPA en el proyecto común y exportándolas al proyecto web.

Descomprime el fichero `jpa-hibernate3.zip` (está en el módulo de JPA) y coloca todas las librerías que no estén ya en el directorio `lib` del proyecto común. Evita que haya librerías duplicadas con el proyecto web. En el caso en que en el proyecto web haya alguna librería que coincide con la que añadimos en el proyecto común, **deja en el proyecto común la versión más reciente** y borra la del proyecto web. Por ejemplo, eso sucede con `commons-collections`; en el proyecto web se encuentra `commons-collections-3.2` y en hibernate se encuentra `commons-collections-2.1.1`. Si dejamos ambos ficheros, podrían aparecer problemas en el proyecto web.

Modifica la configuración del build path del proyecto común para añadir las librerías anteriores al classpath del proyecto. Sustituye la `commons-collections-2.1.1` por la `commons-collections-3.2` y elimina esta última del proyecto web.

Las librerías quedarán así:



6.3. Integración JPA

Crea un directorio `META-INF` en algún directorio de fuentes del proyecto común (por ejemplo, `resources`) y crea allí el fichero `persistence.xml` con el siguiente contenido:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
version="1.0">

<persistence-unit name="Biblioteca">
   <properties>
      <property name="hibernate.archive.autodetection"
                value="class, hbm" />
      <property name="hibernate.connection.driver_class"
                value="com.mysql.jdbc.Driver" />
      <property name="hibernate.connection.url"
                value="jdbc:mysql://localhost:3306/biblioteca" />
      <property name="hibernate.connection.username"
                value="root" />
      <property name="hibernate.connection.password"
                value="especialista" />
      <property name="hibernate.c3p0.min_size"
```

```
                value="5" />
      <property name="hibernate.c3p0.max_size"
                value="20" />
      <property name="hibernate.c3p0.timeout"
                value="300" />
      <property name="hibernate.c3p0.max_statements"
                value="50" />
      <property name="hibernate.c3p0.idle_test_period"
                value="3000" />
      <property name="hibernate.dialect"
                value="org.hibernate.dialect.MySQL5Dialect" />
     <!-- <property name="hibernate.hbm2ddl.auto"
                value="validate" />-->
      <property name="hibernate.show_sql"
                    value="true"/>
          <property name="hibernate.format_sql"
                    value="false"/>
   </properties>
</persistence-unit>
</persistence>
```

Fíjate en que la unidad de persistencia se va a conectar con la BD `biblioteca`. Hemos llamado a la unidad de persistencia `Biblioteca`.

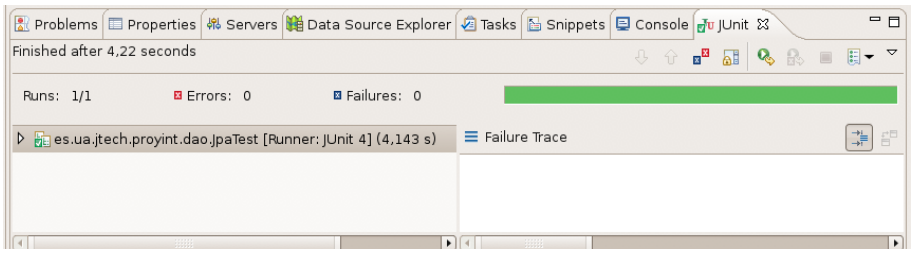
Debemos también introducir en alguna parte del código la posibilidad de obtener un `EntityManagerFactory`. Creamos un *singleton* llamado `EmfSingleton` específicamente para eso en el paquete `proyint.dao`:

```
package es.ua.jtech.proyint.dao;

import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class EmfSingleton {
   final static String persistenceUnitName = "Biblioteca";
   private static EmfSingleton me = new EmfSingleton();
   private EntityManagerFactory emf = null;

   private EmfSingleton() {}

   public static EmfSingleton getInstance() {
      return me;
   }

   public EntityManagerFactory getEntityManagerFactory() {
      if (emf == null) {
         emf =
Persistence.createEntityManagerFactory(persistenceUnitName);
      }
      if (!emf.isOpen()) {
         throw new RuntimeException(
               "Error: alguien ha cerrado el
EntityManagerFactory");
      }
      return emf;
   }
}
```

Por último, creamos un test para comprobar que la conexión JPA funciona perfectamente. Para ello crea la clase `JpaTest.java` en el directorio de fuentes `test`, en el paquete `es.ua.jtech.proyint.dao`. Crea un test dentro de la clase que pruebe a obtener el `entityManagerFactory` y a crear y cerrar un `entityManager`. Comprueba lanzando el test que todo está bien configurado.

```
package es.ua.jtech.proyint.dao;

//imports

public class JpaTest {
   @Test
   public void persistenceTest() {
      // obtenemos un emf
      // obtenemos un entity manager
      // creamos una transacción
      // hacemos un commit de la transacción
      // cerramos el entity manager
   }
}
```

El test debe aparecer en verde cuando se ejecute:



6.4. Mapeado de las entidades

Vamos ahora a mapear las entidades en las tablas de la base de datos. Deberemos mapear los atributos de las entidades, las relaciones entre entidades y la relación de herencia entre la operación y las operaciones activas e históricas. Hay que tener un cuidado especial con los siguientes aspectos:

- Los nombres de las tablas y de las columnas no se corresponden en ocasiones con los nombres de las entidades y de sus atributos. Deberemos definirlos explícitamente en las anotaciones.
- Hay que tener también un cuidado especial con las columnas que tienen un tipo enumerado como base. Los valores del tipo enumerado deben coincidir con los valores del tipo enumerado de la columna.
- Relación de herencia entre `Operacion` y `OperacionHistorico` y `OperacionActiva`. Vamos a usar la estrategia de una única tabla y hay que definir correctamente la columna y los valores discriminantes.

En concreto, las entidades que debes mapear con la base de datos son:

- `LibroEntity`
- `OperacionEntity`
- `OperacionHistoricoEntity`
- `OperacionActivaEntity`
- `UsuarioEntity`
- `MultaEntity`

Como apoyo, proporcionamos a continuación el código fuente de `LibroEntity` y de `OperacionHistoricoEntity`, incluyendo ya las anotaciones que definen el mapeo.

Fichero **`LibroEntity.java`**

```
@Entity
@Table(name = "libro")
public class LibroEntity extends EntityObject {

        private static final long serialVersionUID =
3957186972437085430L;

        @Id
        private String isbn;
        private String titulo;
        private String autor;
        private int numPaginas;

        @Temporal(TemporalType.TIMESTAMP)
        private Date fechaAlta;

        @OneToMany(mappedBy="libro")
        private List<OperacionHistoricoEntity> operaciones;

        @OneToOne(mappedBy="libro")
        private OperacionActivaEntity activa;

        ...
}
```

Fichero **`OperacionHistorico.java`**

```
@Entity
@DiscriminatorValue("historico")
public class OperacionHistoricoEntity extends OperacionEntity {

        private static final long serialVersionUID =
1750153792663968362L;

        @ManyToOne
        @JoinColumn(name="isbn")
        private LibroEntity libro;

        @Temporal(TemporalType.TIMESTAMP)
        private Date fechaFinReal;

        ...
}
```

También podéis encontrar a continuación el código fuente del mapeo de la relación de herencia en la tabla única de operaciones.

```
@Entity
@Table(name = "operacion")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "estadoOperacion")
@ForceDiscriminator
public abstract class OperacionEntity extends EntityObject {

   ...
   @Transient
   private EstadoOperacion estado;
   ...
}
```

La anotación `@ForceDiscriminator` es una anotación propia de Hibernate que resuelve un bug en la recuperación de la lista de operaciones históricas relacionadas con un libro. Sin esa anotación Hibernate no construye bien la SELECT y guarda en la lista operaciones tanto históricas como activas.

La columna discriminante `estadoOperacion` no puede ser parte de la entidad. Sin embargo, en la entidad tenemos definidos ese atributo. Para no mapearlo con la base de datos, lo declaramos como *transient* con la anotación `@Transient`.

El código para mapear un atributo enumerado (por ejemplo, el estado de una `MultaEntity`) puede incluir un mapeo explícito de la definición de la columna, que relacione los valores enumerados de la base de datos con los valores Java del tipo enumerado. En nuestro caso no es estrictamente necesario, porque las cadenas que definen los valores en la base de datos coinciden con los valores del tipo enumerado.

```
@Enumerated(EnumType.STRING)
@Column(name =
"estadoMulta",columnDefinition="enum('activa','historico')")
private EstadoMulta estado = EstadoMulta.activa;
```

6.4.1. Tests de JUnit

Para comprobar que el mapeo se realiza correctamente, implementad en el fichero `JpaTest.java` un test JUnit por cada una de las entidades. El test debe buscar en la base de datos un elemento que sabemos que existe y comprobar sus atributos, incluyendo las relaciones con otras entidades.

Ejemplo de un test:

```
   @Test
   public void multaTest() {
      EntityManagerFactory emf = EmfSingleton.getInstance()
            .getEntityManagerFactory();
      EntityManager em = emf.createEntityManager();
      em.getTransaction().begin();
      MultaEntity multa = em.find(MultaEntity.class, 1);
```

```
      assertNotNull(multa);
      assertEquals(multa.getEstado(), EstadoMulta.activa);
      assertEquals(multa.getUsuario().getLogin(), "antonio");
      assertEquals(compararFechas(multa.getFechaInicio(),2008, 9,
16, 12, 35, 5), 0);
assertEquals(compararFechas(multa.getFechaFin(),2010,4,21,12,35,5),0);
      em.getTransaction().commit();
      em.close();
   }
   ...
   private int compararFechas(Date fecha, int anyo, int mes, int
dia, int horas, int minutos, int segundos) {
      GregorianCalendar fechaCalendar = new GregorianCalendar();
      fechaCalendar.setTime(fecha);
      // El formato de una fecha en el GregorianCalendar es
      // año, mes (empezando en 0), día, hora, minuto, segundo
      GregorianCalendar fecha2 = new
GregorianCalendar(anyo,mes-1,dia,horas,minutos,segundos);
      return fecha2.compareTo(fechaCalendar);
   }
```

6.5. Implementación de los DAO con JPA

Una vez implementadas las entidades y comprobado su correcto funcionamiento con los tests, pasamos a implementar los DAO.

Define las clases que implementan las interfaces `IUsuarioDAO`, `ILibroDAO` y `IOperacionDAO` utilizando JPA. Llámalas `UsuarioJPADAO`, `LibroJPADAO` y `OperacionJPADAO` e implementa todos sus métodos.

Recuerda que, para no cambiar el funcionamiento de lo que hemos implementado hasta ahora, cada operación del DAO debe ser atómica y debe abrir y cerrar el `EntityManager`. Dejamos para otra ocasión la refactorización del código para utilizar EAOs.

Comienza con la clase `LibroJPADAO` se implementa como sigue:

```
package es.ua.jtech.proyint.dao.libro;

//imports

public class LibroJPADAO implements ILibroDAO {

   EntityManagerFactory emf;

   public LibroJPADAO() {
      emf = EmfSingleton.getInstance().getEntityManagerFactory();
   }

   public void addLibro(LibroEntity libro) throws DAOException {
      EntityManager em = emf.createEntityManager();
      try {
         em.getTransaction().begin();
         em.persist(libro);
         em.getTransaction().commit();
      } catch (PersistenceException e) {
```

```
         if (em.getTransaction().isActive())
            em.getTransaction().rollback();
         throw new DAOException("Error de JPA", e);
      } finally {
         em.close();
      }
   }

   public int delLibro(String isbn) throws DAOException {
      int borrado = 0;
      EntityManager em = emf.createEntityManager();
      em.getTransaction().begin();
      LibroEntity libro = em.find(LibroEntity.class, isbn);
      if (libro != null) {
         borrado = 1;
         em.remove(libro);
      }
      em.getTransaction().commit();
      em.close();
      return borrado;
   }

   public LibroEntity getLibro(String isbn) throws DAOException {
      EntityManager em = emf.createEntityManager();
      em.getTransaction().begin();
      LibroEntity libro = em.find(LibroEntity.class, isbn);
      em.getTransaction().commit();
      em.close();
      return libro;
   }

   @SuppressWarnings("unchecked")
   public List<LibroEntity> getLibros() throws DAOException {
      EntityManager em = emf.createEntityManager();
      List<LibroEntity> lista = em.createQuery("SELECT l FROM
LibroEntity l")
            .getResultList();
      em.close();
      return lista;
   }

   public int updateLibro(LibroEntity libro) throws DAOException {
      int actualizado = 0;
      EntityManager em = emf.createEntityManager();
      em.getTransaction().begin();
      LibroEntity managedLibro = em.getReference(LibroEntity.class,
libro
            .getIsbn());
      if (managedLibro != null) {
         em.merge(libro);
         actualizado = 1;
      }
      em.getTransaction().commit();
      em.close();
      return actualizado;
   }
}
```

Hay que tener un cuidado especial con las consultas JPA en el `OperacionJPADAO`. Un ejemplo de consulta es el siguiente:

```
public List<LibroEntity> getLibrosDisponibles() throws DAOException
{
      List<LibroEntity> result = null;

      EntityManager em = emf.createEntityManager();
      result = (List<LibroEntity>) em.createQuery(
            "SELECT l FROM LibroEntity l WHERE l.activa is EMPTY")
            .getResultList();
      em.close();

      return result;
   }
```

Si alguna de las consultas se te resiste (tanto en esta integración como en el proyecto web), envía una pregunta al foro y daremos alguna pista.

6.5.1. Tests de JUnit

Define algunos tests que prueben algunas de las funciones de los nuevos DAO (alguna consulta que no modifique la base de datos). No hace falta que compruebes todas las funciones; basta con algunas. Incluye los tests en el directorio de tests, en el paquete `es.ua.jtech.proyint.jpa`, junto con los tests definidos anteriormente.

6.6. Modificación de la FactoriaDAOs

Una vez creados los nuevos DAO, debemos modificar la clase `FactoriaDAOs` para que devuelva implementaciones de JPA en lugar de implementaciones de JDBC.

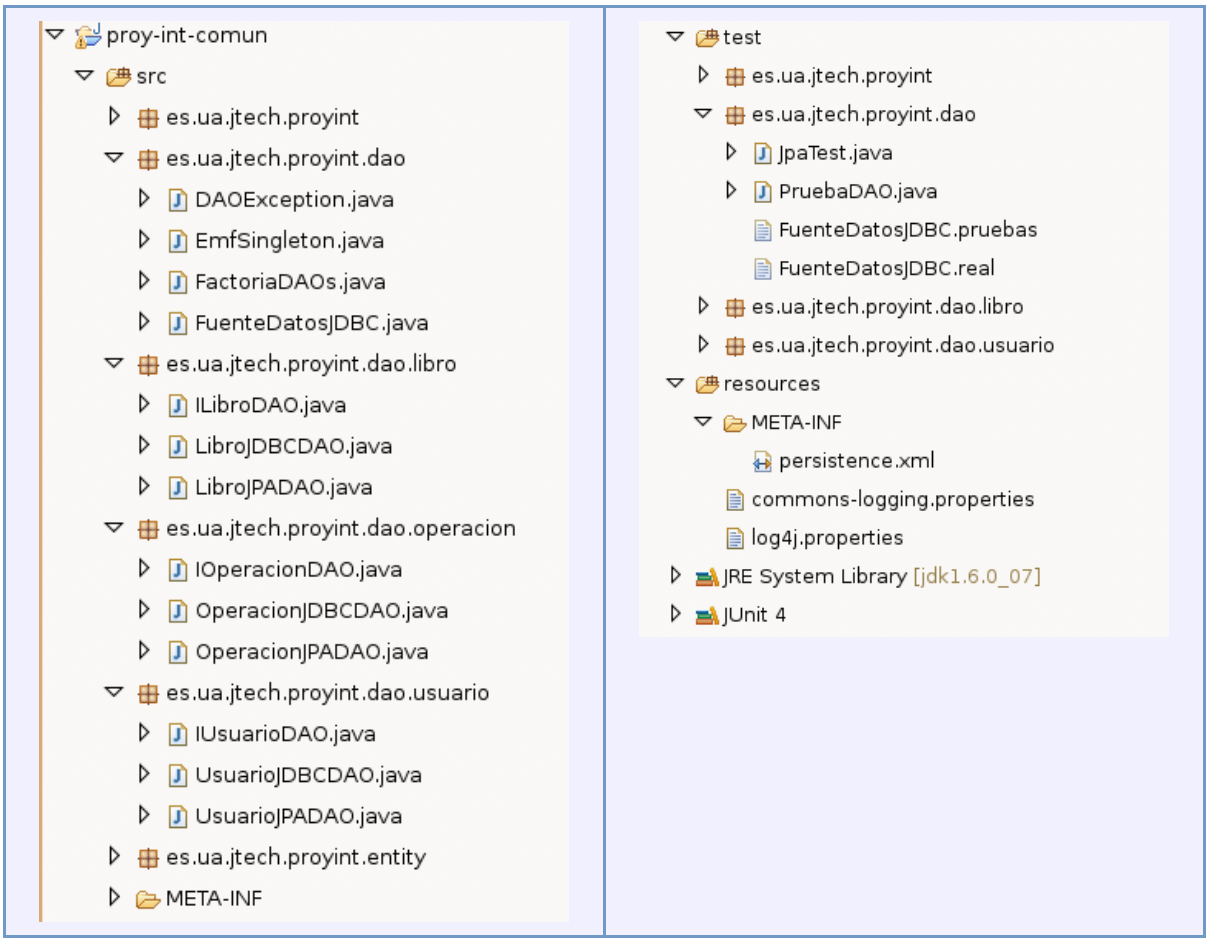
6.7. Proyecto Web

Exporta las librerías para que se desplieguen en la aplicación web.

No habría que tocar nada de código del proyecto web. Seguirá funcionando utilizando la abstracción proporcionada por las interfaces DAO. Pero ahora las entidades que se devuelven han sido obtenidas utilizando JPA en lugar de JDBC.

Comprueba que la aplicación sigue funcionando correctamente. En concreto, analiza el código de `RealizarReservaAccion` para estudiar si puede encontrarse algún problema al haber cambiado la implementación de los DAO. Comprueba que funciona correctamente.

6.8. Entrega



Se deberán realizar las siguientes modificaciones en los proyectos común y web:

- Instalar las librerías de JPA/Hibernate en el proyecto común y el fichero de configuración de JPA `persistence.xml`.
- Incluir la creación del `EntityManagerFactory` en la clase `FactoriaDAOs`.
- Mapear con las tablas de la base de datos las entidades `LibroEntity`, `OperacionEntity`, `OperacionActivaEntity`, `OperacionHistoricoEntity`, `UsuarioEntity` y `MultaEntity` definiendo las relaciones entre ellas.
- Definir y probar los tests de JUnit para las entidades.
- Implementar los DAO `LibroJPADAO`, `OperacionJPADAO` y `UsuarioJPADAO` utilizando JPA y las entidades definidas anteriormente. Los JPADAO deben implementar la misma interfaz que los JDBCDAO; así no hay que modificar el código de las clases que acceden a ellos (tests de JUnit y capa web).
- Modificar los tests JUnit de los DAO Libro y Operacion para usen los DAO JPA.
- Modificar la factoría de DAOs para que devuelva las implementaciones de JPA.
- Actualizar las librerías en el proyecto web y comprobar que funciona correctamente.

7. Proyecto Web

7.1. Introducción

El objetivo principal de las sesiones del proyecto web es refactorizar la aplicación de la biblioteca para dotarla de una calidad semejante a la de un proyecto empresarial real. Esto se traduce en un subconjunto de actividades:

- Crear un esquema de navegación apropiado
- Crear un interfaz de usuario amigable
- Aplicar una nomenclatura adecuada a todo el proyecto
- Uso extensivo de Struts, mediante sus taglibs, gestión de errores, validaciones, etc...
- Uso de JPA para el acceso a la base de datos
- Internacionalización de toda la aplicación en castellano e inglés
- Mejorar la calidad del código implementado

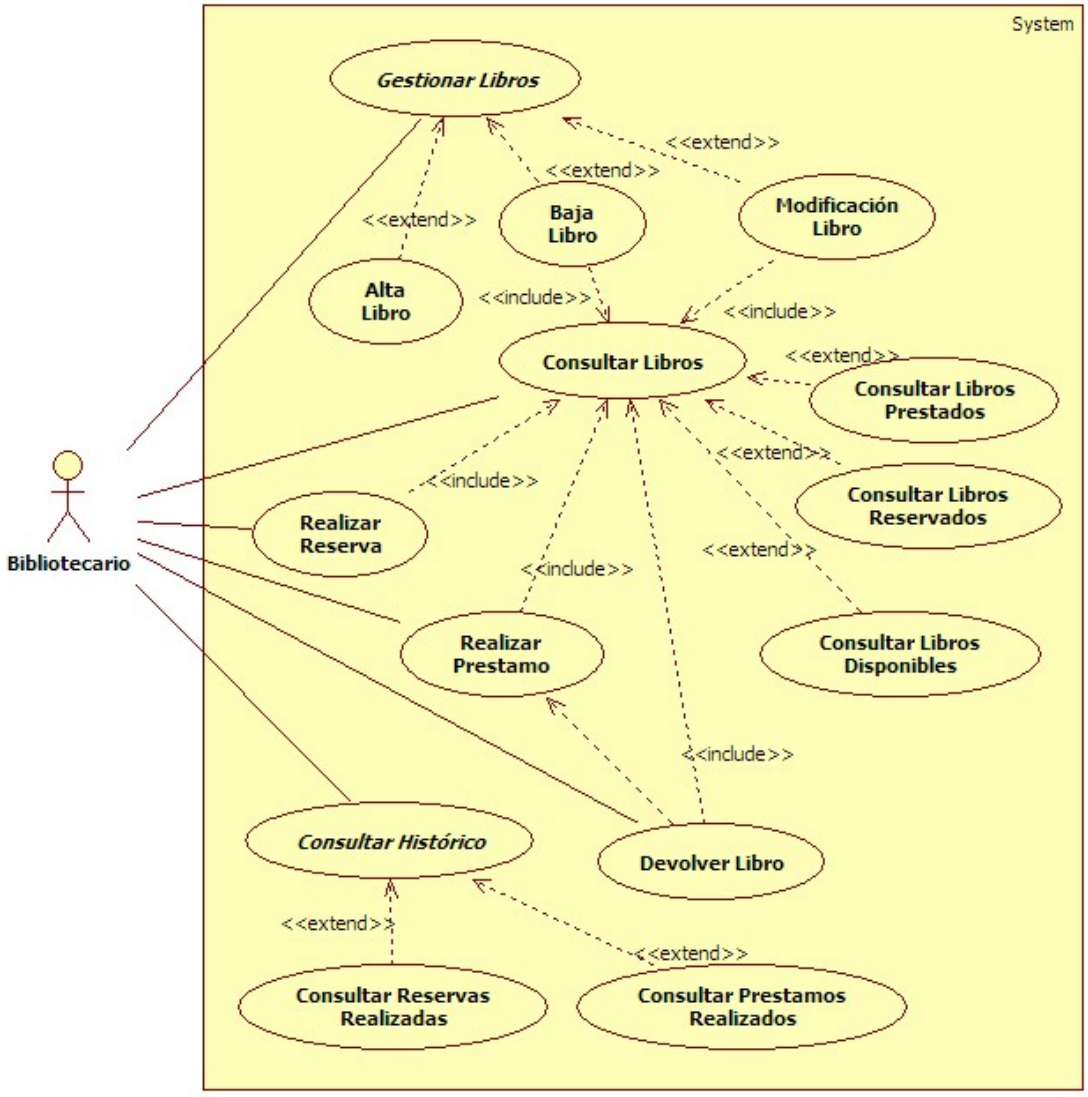
Aclaración

A lo largo de la sesión nos vamos a dedicar a retocar/reescribir código previamente implementado, pero ¿porqué no lo hemos hecho bien desde el principio? El planteamiento del supuesto inicial, conjunto al desarrollo incremental de cada sesión, fomenta que cada sesión adopte las necesidades del proyecto a las tecnologías empleadas. Ahora es el momento de rehacer gran parte de la aplicación y formar una base estable para futuros desarrollos.

7.2. Casos de Uso

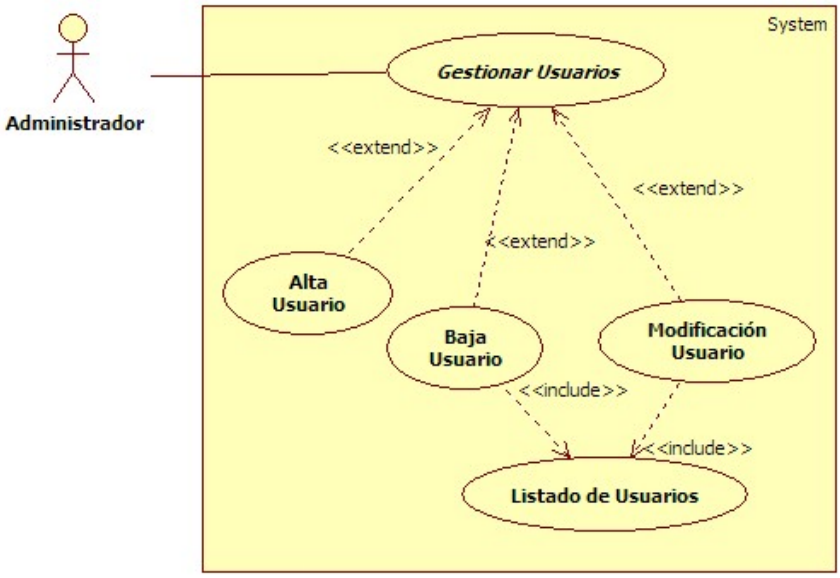
Tal como mostramos en la primera sesión, la aplicación consta de diversos casos de uso dependiendo del rol que se autentique contra el sistema. A lo largo del proyecto web nos vamos a centrar principalmente tanto en el rol bibliotecario, como en el de administrador.

Los casos de uso que lleva asociado un usuario con rol bibliotecario son:



Casos de uso rol Bibliotecario

En cuanto al administrador, los casos de uso son:



Casos de uso rol Administrador

Cada una de las siguientes secciones, analizará y resolverá algunas partes de la aplicación, partiendo de la funcionalidad, interfaz de usuario y calidad del código. El resto de funcionalidades, quedarán como ejercicios.

7.3. Refactorizando el Login

Vamos a empezar por la puerta de entrada de la aplicación. Nuestro departamento de diseño gráfico nos ha entregado la siguiente hoja de estilo que será común para toda la web (la colocaremos en `WebContent/css/style.css`), una hoja de estilo para el login (la colocaremos en `WebContent/css/login.css`), y un conjunto de imágenes (que colocaremos en la carpeta `WebContent/imagenes`). Además, nos anexan los siguientes pantallazos con la apariencia de la aplicación:

- Login

Pantalla de Login

Podemos observar 3 bloques importantes: las cajas de login/password, la zona para los mensajes de error al entrar al sistema, y los enlaces para cambiar el idioma de la aplicación.

- Web del Bibliotecario

Web del Bibliotecario

7.3.1. Página de login

Esta página es la puerta de entrada a la aplicación. La página ya maquetada en html y con los tags de struts, la colocaremos en `WebContent/login.jsp`. Recordar de añadir las traducciones en el `mensajes.properties` adecuado.

```
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html"%>
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean"%>
<%@ page language="java" contentType="text/html;
charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html:html>
<head>
<html:base />
<title><bean:message key="login.titulo"/></title>
<meta http-equiv="Content-Type" content="text/html;
```

```
charset=ISO-8859-1">
<link rel="stylesheet" type="text/css"
    href="<html:rewrite page="/css/login.css" />" title="login
style"
    media="screen,projection" />
</head>

<body>

<br /><br /><br /><br />

<form action="j_security_check" method="post">
<div id="wrap">
    <div id="hueco-arriba">
    <h3><bean:message key="login.bienvenido"/></h3>
    <p><bean:message key="login.instrucciones"/></p></div>

    <div id="izq">
    <bean:message key="login.login"/><p /><br />
    <bean:message key="login.password"/></div>

    <div id="der"><input type="text" name="j_username">
        <p /><input type="password" name="j_password">
        <input type="submit" value="<bean:message
key="login.entrar"/>"/></div>

    <div id="hueco-abajo">
    <p><html:link page="/cambiaIdioma.do?idioma=en"><bean:message
key="login.ingles"/></html:link>
    -
    <html:link page="/cambiaIdioma.do?idioma=es"><bean:message
key="login.espanyol"/></html:link></p>
    <p> </p></div>
</div>
</form>

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

Destacar el uso intensivo de las capas html para simplificar el código. Para añadir el mensaje de error, utilizaremos una nueva capa con `id="box"`:

```
<div class="box">Aqui mostraremos los mensajes de error<br />
    <strong><bean:message key="errorLogin.mensaje"/></strong></div>
```

7.3.1.1. Problemas de la Seguridad Declarativa

Muchas aplicaciones web colocan el formulario de login en cada página pública que es accesible, normalmente en la parte superior de la página, ya sea a izquierda o derecha. Pero con la seguridad declarativa, el contenedor solo puede referenciar al login desde la página fijada con `login-config`.

Del mismo modo, no podemos controlar mensajes de error al realizar el login. Por esto, la solución planteada para mostrar los mensajes de error no es la más elegante, ya que el uso de la seguridad declarativa reduce el control que tenemos sobre el comportamiento de la

aplicación.

Sin Seguridad Declarativa

Si no utilizásemos la seguridad declarativa, al hacer login, introduciríamos dentro de la sesión algún atributo (lo más ligero objeto, por ejemplo, el login del usuario y el rol del usuario), el cual comprobaríamos que existe para cada acción que requiera seguridad. Para evitar la redundancia en múltiples acciones, sobreescribiríamos parte del controlador, en concreto el método `processRoles` de la clase `RequestProcessor`. Más información en www.devarticles.com/c/a/Java/Securing-Struts-Applications/

Optativo I

Si duplicamos las páginas de `login.jsp` y `errorLogin.jsp`, al realizar una modificación en una de ellas, tendremos que volver a modificar la otra. Realiza fragmentos de página intermedios y únelos mediante `includes`.

7.4. Web de Bibliotecario

Tal como hemos mostrado antes, las opciones del bibliotecario serán la gestión de los libros (alta, baja, modificación), así como los listados de libros y operaciones (libros disponibles, reservados, prestados, etc...).

Una vez nos hemos autenticado, y nuestro controlador nos lleva a la página adecuada al rol autenticado (el cual hemos de mostrar al usuario), hemos de mostrar un menú con las posibles opciones a realizar, así como ofrecer la posibilidad de invalidar la sesión.

Menú del Bibliotecario

El esquema de las páginas html de los tres roles es similar, siendo el menú de opciones lo único que cambian, y a partir de las opciones mostradas, las diferentes acciones que puede realizar un determinado rol.

El equipo de maquetación nos entrega el código html de la web del bibliotecario (para facilitar el trabajo, un programador ha "mejorado" la navegación, y tenemos la página separada en fragmentos de página para separar y reutilizar tanto el pie como la cabecera). Este esquema ya lo hemos utilizado desde el proyecto de jsp y con Struts.

Plantillas

A la hora de implementar una aplicación de este tipo, el uso de un gestor de plantillas, del tipo de *Tiles* (tiles.apache.org), *SiteMesh* (www.opensymphony.com/sitemesh) o *Velocity* (velocity.apache.org), facilita mucho el desarrollo, minimizando la cantidad de código a implementar y la redundancia de HTML.

7.4.1. /jsp/cabecera.jspf

La cabecera va a mostrar el logo de la aplicación, así como qué usuario ha entrado al sistema, y cual es el rol de dicho usuario. Dentro de la cabecera también se ofrece la posibilidad de cerrar la sesión, mediante la llamada a un action, el cual deberá invalidar el objeto *Session* de la *request*.

El siguiente código lo colocaremos en /jsp/cabecera.jspf

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean"
%>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html"
%>
<div id="header">
<table width="100%">
<tr>
    <td><h1><bean:message key="cabecera.titulo"/></h1></td>
    <td> </td>
    <td align="right"><html:img page="/imagenes/banner-java.gif"
/></td>
</tr>
    <tr>
        <td colspan="3" align="right">
            <html:link action="/logout"><bean:message
key="cabecera.cerrarSesion" /> &raquo;</html:link>
        </td>
    </tr>
    <tr>
        <td colspan="3" align="right">
        <bean:message key="cabecera.usuario" />:
            <strong><%= request.getUserPrincipal().getName()
%></strong>
        -
        <bean:message key="cabecera.rol" />:
            <strong>
              <%=
session.getAttribute(es.ua.jtech.proyint.struts.Tokens.SES_ROL) %>
            </strong>
        </td>
    </tr>
</table>
```

```
</div>
```

Para poder obtener el rol, en el action del login, previamente deberemos haber guardado dicho rol dentro de la sesión.

7.4.2. /jsp/pie.jspf

En el pie, únicamente vamos a poner una firma. El siguiente código lo colocaremos en /jsp/pie.jspf

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean"
%>
<div id="footer">
    <p><bean:message key="pie.mensaje"/> -
       <a href="http://www.jtech.ua.es">www.jtech.ua.es</a></p>
</div>
```

7.4.3. /jsp/biblio/menu.jspf

El menú va a depender de cada rol existente, y por tanto, vamos a repetir este archivo con diferentes opciones tanto para los usuarios bibliotecario, como los administradores, socios y registrados. El siguiente código lo colocaremos en /jsp/biblio/menu.jspf

```
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html"
%>
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean"
%>
<div id="sidebar">
<h2><bean:message key="mbiblio.libros"/></h2>
<form>
<fieldset>
<legend><bean:message key="mbiblio.operaciones"/></legend>
    <ul>
    <li>
        <html:link action="/libroPreAdd"><bean:message
key="mbiblio.alta"/></html:link>
    </li>
    </ul>
</fieldset>
</form>
<br/>
<form>
<fieldset>
<legend><bean:message key="mbiblio.listados"/></legend>
    <ul>
    <li>
        <html:link action="/libroListTodos"><bean:message
key="mbiblio.todos"/></html:link>
        <html:link action="/libroListPrestados"><bean:message
key="mbiblio.prestados"/></html:link>
        <html:link action="/libroListReservados"><bean:message
key="mbiblio.reservados"/></html:link>
        <html:link action="/libroListDisponibles"><bean:message
key="mbiblio.disponibles"/></html:link>
    </li>
```

```
    </ul>
</fieldset>
</form>
</div>
```

7.5. Alta de Libros

La opción de alta de libros está accesible desde el menú del rol bibliotecario. Tal como habeis hecho en la sesiones anteriores, tendremos un Action para la precarga donde haremos un `saveTokens` para evitar el submit doble. Tras ello, mostraremos un formulario para dar de alta un libro, similar al siguiente pantallazo:

Alta de Libro

7.5.1. /jsp/biblio/altaLibro.jsp

El equipo de maquetación nos envía el siguiente jsp con los tags de Struts, el cual colocaremos en /jsp/biblio/altaLibro.jsp. Fijaros que ya está activada la validación en *Javascript* del formulario mediante el *Validator* de *Struts*:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html"%>
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html:html>
<head>
<html:base />
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title><bean:message key="altaLibro.titulo" /></title>
<link rel="stylesheet" type="text/css" href="<html:rewrite
page="/css/style.css" />"
    title="800px style" media="screen,projection" />
```

```
</head>
<body>

<div id="wrap"><%@include file="/jsp/cabecera.jspf"%>
<%@include file="/jsp/biblio/menu.jspf"%>

<div id="content">

<h2><bean:message key="altaLibro.titulo" /></h2>

<html:form action="/libroAdd">
<bean:message key="altaLibro.introISBN" />
<fieldset><legend><bean:message key="altaLibro.datosLibro"
/></legend>
    <table>
        <tr>
            <td><bean:message key="libro.ISBN" />:</td>
            <td><html:text property="isbn" /> <html:errors
property="isbn" /></td>
        </tr>
    </table>
</fieldset>
<bean:message key="altaLibro.introDatos" />
<fieldset><legend><bean:message key="altaLibro.datosPersonales"
/></legend>
    <table>
        <tr>
            <td><bean:message key="libro.titulo" />:</td>
            <td><html:text property="titulo" /> <html:errors
                property="titulo" /></td>
        </tr>
        <tr>
            <td><bean:message key="libro.autor" />:</td>
            <td><html:text property="autor" /> <html:errors
property="autor" /></td>
        </tr>
        <tr>
            <td><bean:message key="libro.paginas" />:</td>
            <td><html:text property="numPaginas" /> <html:errors
                property="numPaginas" /></td>
        </tr>
        <tr>
            <td></td>
            <td><html:submit>
                <bean:message key="mbiblio.alta" />
            </html:submit></td>
        </tr>
    </table>
</fieldset>
</html:form>
<br />
</div>

<%@include file="/jsp/pie.jspf"%></div>

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

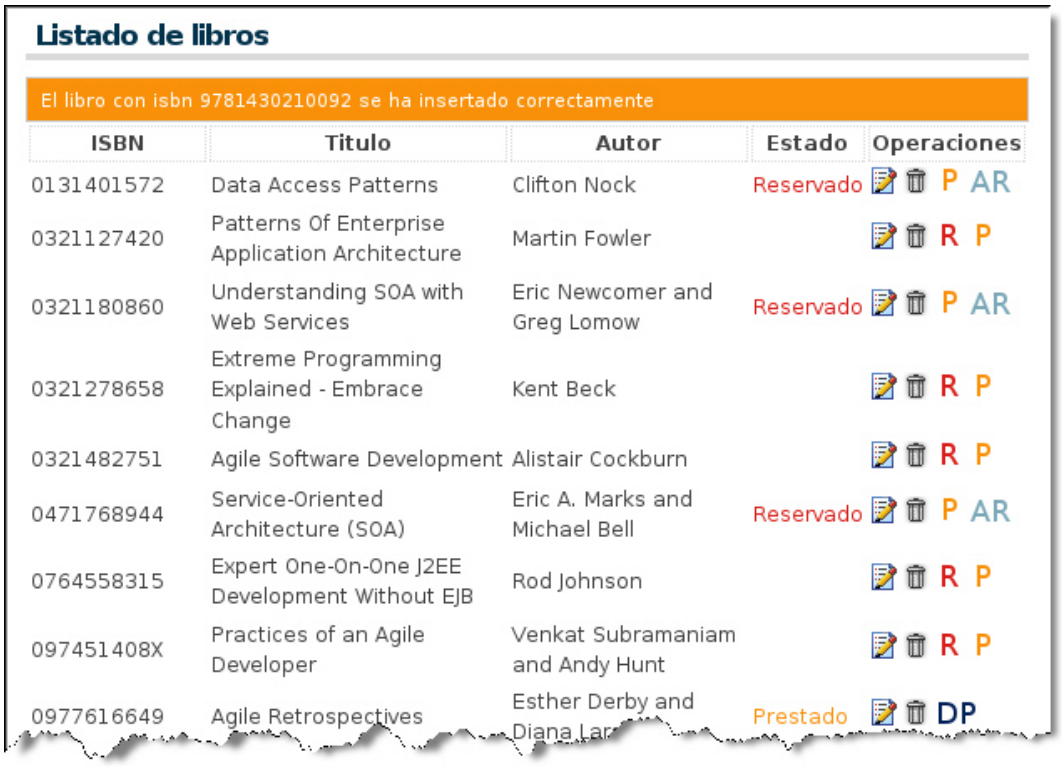
Tras insertar un libro, vamos a ir al listado de todos los libros, mostrando un mensaje de que el libro se ha insertado correctamente.

Mensaje tras Alta de Libro

7.6. Listado de Libros

El listado de libros es el lugar que va a desencadenar más acciones, pudiéndose realizar cualquier operación sobre un determinado libro (editar, borrar, reservar, prestar, devolver, ...).

La apariencia de los listados debe ser semejante a la siguiente, teniendo en cuenta que dependiendo del estado del libro, se deberán mostrar las opciones pertinentes (**P**restar, **R**eservar, **A**nular **R**eserva, o **D**evolver **P**réstamo):



Listado de Todos los Libros

La dificultad de los listados va a residir, por un lado en realizar la consulta SQL/JPQL adecuada, y por otro, dibujar los datos teniendo en cuenta el estado de los libros.

Curiosidad

Aunque la codificación de JDBC esta cayendo en desuso, las empresas que siguen empleándolo, utilizan algún tipo de framework para evitar repetir el código de apertura y cierre de recursos, por ejemplo *Commons DbUtils* (jakarta.apache.org/commons/dbutils) o una parte de iBatis (ibatis.apache.org). Además, suelen definir los nombres de los campos con constantes (igual que la interfaz `Tokens`), por si surge la necesidad de renombrar un campo en la base de datos, minimizar los cambios en el código Java.

7.6.1. /jsp/biblio/listadoLibros.jsp

La página que va a pintar el listado de libros es la misma independientemente del tipo de listado que sea. Para poder hacer esto, tendremos diferente zonas con etiquetas condicionales que comprobaran el tipo de operación del libro, y mostrará las imagenes/acciones adecuadas.

El contenido de la capa del marco de información es el siguiente:

```
<div id="content">
```

```
<h2><bean:message key="listLibros.titulo"/></h2>

<logic:messagesPresent message="true">
    <div class="box">
    <html:messages message="true" id="msg"><c:out value="${msg}"
/><br/></html:messages>
    </div>
</logic:messagesPresent>

<logic:messagesPresent>
    <div class="box"><html:errors /></div>
</logic:messagesPresent>

<c:choose>
<c:when test="${empty requestScope.listaLibros}">
    <p><bean:message key="listLibros.noencontrados"/></p>
</c:when>
<c:otherwise>
    <table>
    <tr>
    <th><strong><bean:message key="libro.ISBN"/></strong></th>
    <th><strong><bean:message key="libro.titulo"/></strong></th>
    <th><strong><bean:message key="libro.autor"/></strong></th>
    <th><strong><bean:message key="libro.estado"/></strong></th>
    <th><strong><bean:message
key="libro.operaciones"/></strong></th>
    </tr>

    <c:forEach items="${requestScope.listaLibros}" var="libro">
        <tr>
        <td>${libro.isbn}</td>
        <td>${libro.titulo}</td>
        <td>${libro.autor}</td>

        <c:choose>
        <c:when test="${libro.activa==null}">
            <td> </td>
        </c:when>
        <c:when test="${libro.activa.tipo eq 'reserva'}">
            <td><span style="color:#e11a1a"><bean:message
key="libro.reservado"/></span></td>
        </c:when>
        <c:when test="${libro.activa.tipo eq 'prestamo'}">
            <td><span style="color:#ff8e09"><bean:message
key="libro.prestado"/></span></td>
        </c:when>
        </c:choose>

        <td>
            <html:link action="/libroPreUpdate" paramId="isbn"
paramName="libro"
             paramProperty="isbn">
                <html:img page="/imagenes/editar.gif"
titleKey="libro.editar" />
            </html:link>
            <html:link action="/libroDel" paramId="isbn"
paramName="libro"
             paramProperty="isbn">
```

```
                <html:img page="/imagenes/borrar.gif"
titleKey="libro.borrar" />
            </html:link>

            // operaciones según estado
        </td>
        </tr>

    </c:forEach>

    </table>
</c:otherwise>
</c:choose>

</div>
```

Título de los Listados

Si todos los listados reutilizan este jsp, ¿qué podemos hacer para que dependiendo de si listamos las reservas, los préstamos o todos los libros, el titulo del listado cambie acorde al tipo de listado? ¿Realmente es mejor tener un único jsp, o tener una vista para tipo de listado?

7.6.2. Apariencia de los listados

Dependiendo del tipo de listado, vamos a mostrar cierta cantidad de información, y el usuario dispondrá de diferentes opciones. A continuación se muestra un pantallazo de cada uno de los listados:

Listado de todos los libros

Todos los libros

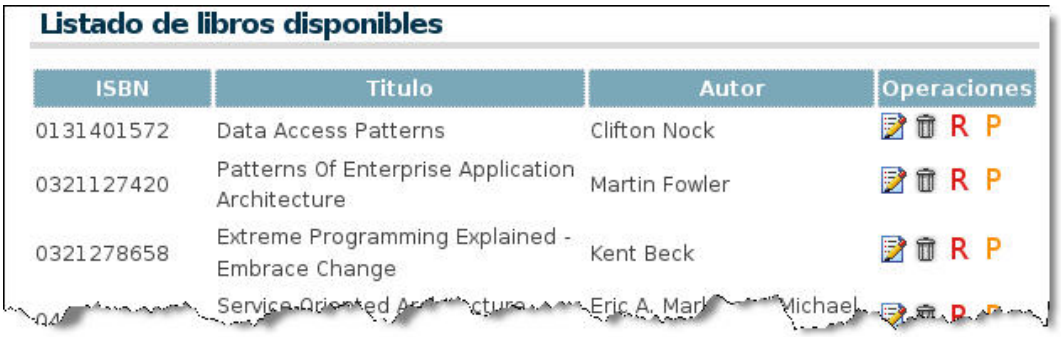
Listado de libros prestados

Libros prestados

Listado de libros reservados

Libros reservados

Listado de libros disponibles



Libros disponibles

Respecto a la leyenda de colores utilizada para cada listado son:

Listado de Libros	Color	Código HTML
Todos	sin color	
Prestados	naranja	#ff8e09
Reservados	rojo	#e11a1a
Disponibles	azul	#79a9b9

Por ejemplo, para el listado de libros disponibles, comenzaremos la tabla con:

```
<tr style="background-color:#79a9b9; color:white"></tr>
```

7.7. Edición y Borrado de Libros

Para poder editar o borrar un libro, previamente debemos haberlo seleccionado desde cualquier de los listados de libros.

La edición de un libro se basa en el mismo formulario que el de alta, con la diferencia que el campo ISBN no va a ser editable. En cuanto al borrado, al pulsar sobre la papelera, debería pedirle confirmación al usuario, mediante una validación en Javascript (**optativo**)

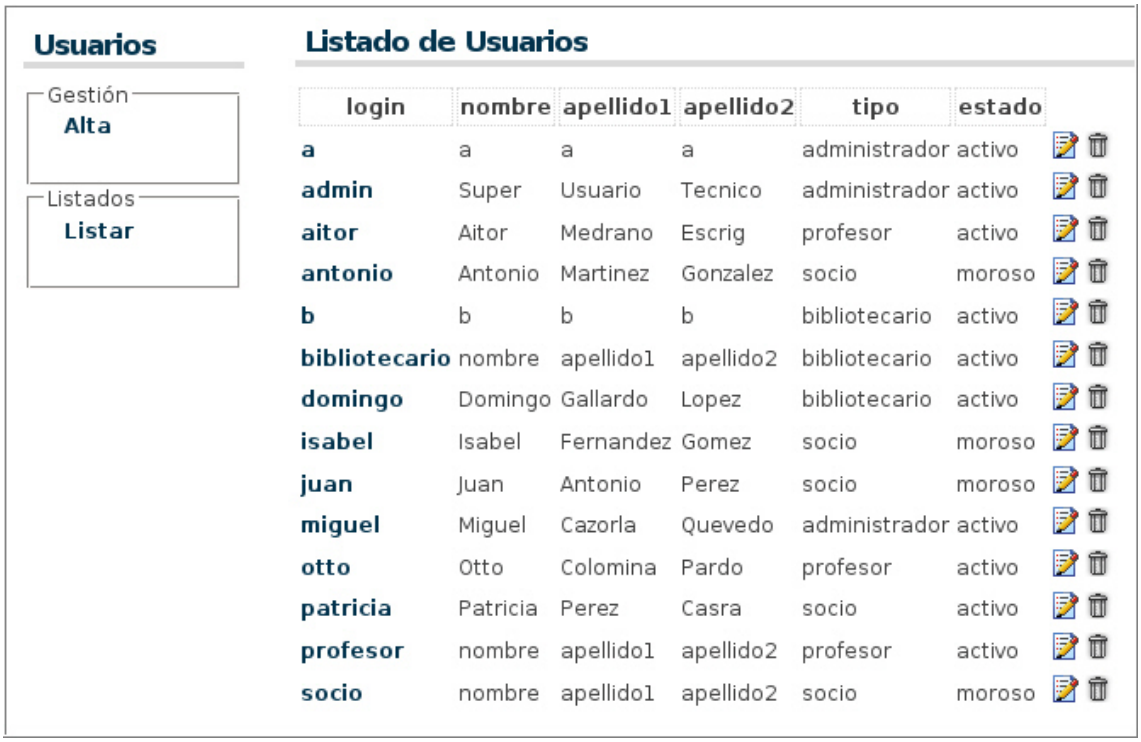
En ambas operaciones, una vez realizada la acción, se devolverá el control a listado de todos los libros, informando al usuario que la edición o el borrado se ha realizado con éxito.

7.8. Web del Administrador

Se plantea como parte de los ejercicios obligatorios el desarrollo del web del Administrador, siguiendo las mismas pautas que para la web del Bibliotecario. El administrador del sistema va a ser el encargado de gestionar los usuarios, de modo que al entrar a su web, le debe aparecer un listado de todos los usuarios, dándole la posibilidad de crear un nuevo usuario, o de modificar/borrar uno de los usuarios existentes.

7.8.1. Listado de Usuarios

A continuación se muestra un pantallazo de la apariencia del listado, donde podemos observar que para cada usuario, podemos editar sus datos o bien eliminarlo (mediante los iconos correspondientes):



Web del Administrador

Paginación y/o Buscador

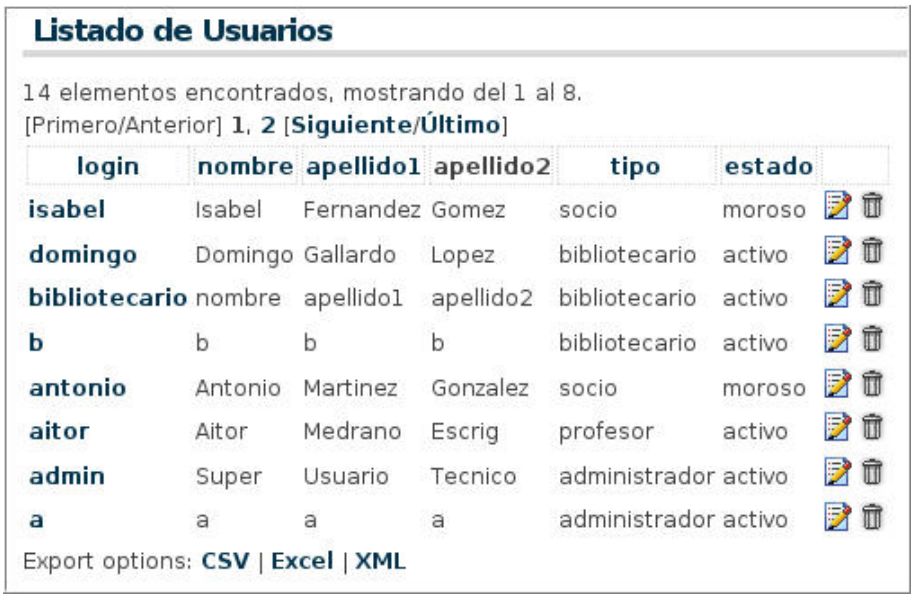
Problema: ¿Qué pasa cuando el listado contenga 10000 usuarios?
Solución I: Paginación (por ejemplo, 20 usuarios por pantalla) y Ordenar (permitir clickar sobre

los nombres de las columnas y ordenar el listado por dicho campo).
Solución II: Crear un buscador que permita obtener usuarios que cumplan ciertos requisitos (expresión regular con el login, estado del usuario, etc...)

Este tipo de listados, siempre van acompañados de un buscador para filtrar los resultados y la posibilidad de ordenación de los resultados mediante la acción de clickar sobre las columnas.

Optativo II

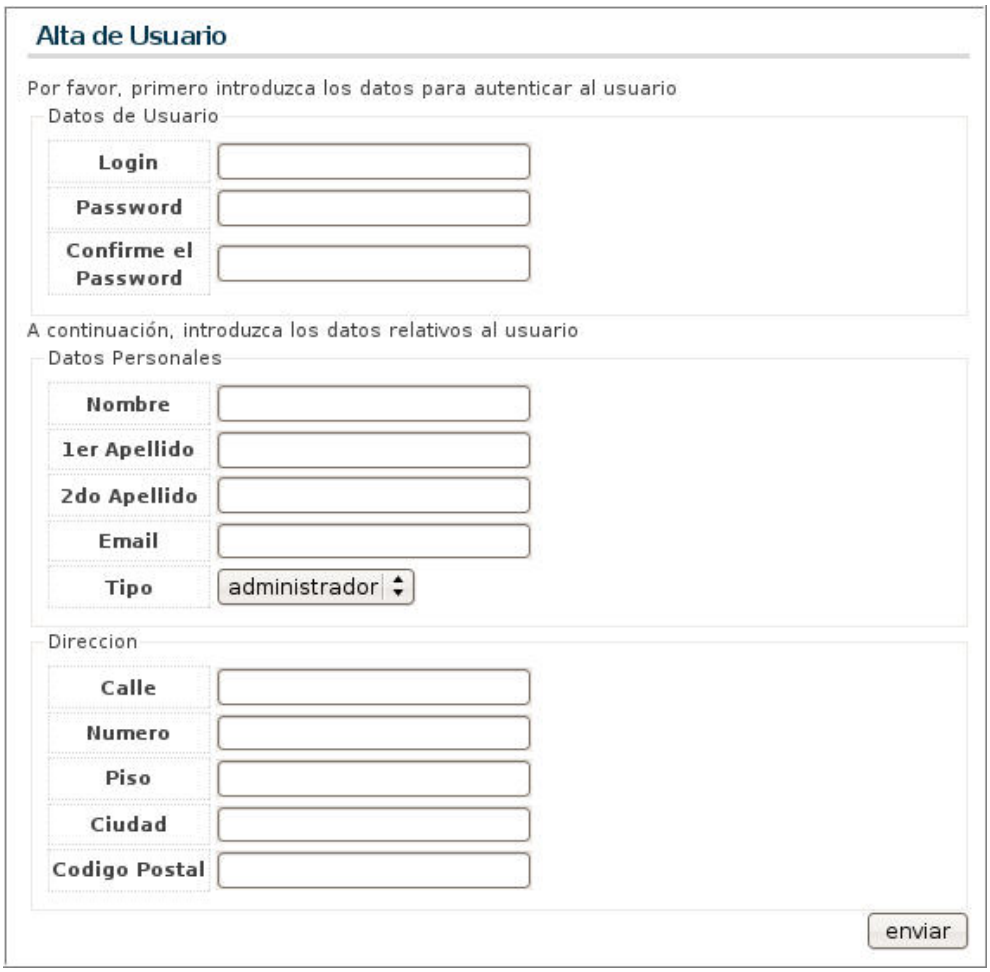
El uso de la librería displaytags (displaytag.sf.net) permite la paginación y la ordenación de los listados. Se plantea como ejercicio optativo su inclusión dentro del listado de usuarios.



Web del Administrador con displaytags

7.8.2. Alta/Modificación de Usuario

Cuando se crea un usuario, se debe comprobar que la contraseña introducida es correcta, mediante una introducción redundante de la misma:



Alta de Usuario

También hemos ocultado la creación del estado del usuario, ya que un usuario recién creado es un usuario activo. En cambio, en el formulario de modificación de un usuario, si que se debe ofrecer la posibilidad de modificar su estado:

Modificación de Usuario

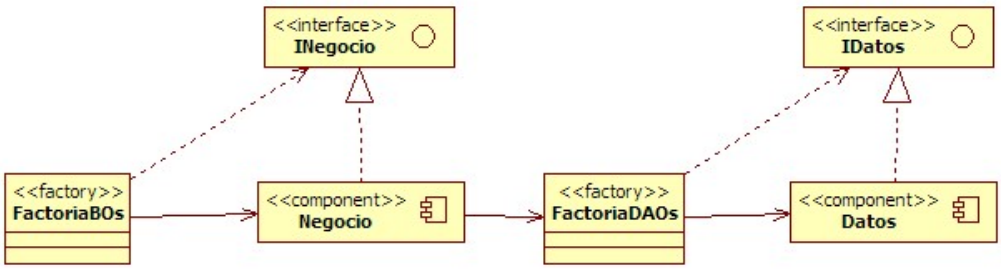
¿Negocio en el Action?

Si el Action es el encargado de llamar al DAO, a la hora de realizar una reserva, entonces deberá obtener la fecha actual (fecha de inicio) y dependiendo del estado del usuario, a partir del número de días que tiene derecho, calcular la fecha de fin. Previamente, deberá obtener el número de reservas o prestamos activos, para comprobar si ya tiene completo el cupo de libros. ¿Y todo esto lo colocamos en lo que llamamos *controlador*?

7.9. Capa de Negocio

Ya hemos visto cómo estamos poniendo lógica de negocio en nuestros Actions. Por ejemplo, al hacer una inserción de un libro, el *Action* le pone la fecha de alta al libro asignandole la fecha actual, o cuando hacemos una reserva/préstamo, el *Action* se está encargando obtener la fecha actual, y dependiendo del rol, poner la fecha de finalización. Todo esto es **lógica de negocio** que debe residir en una capa aparte.

Para ello, vamos a crear un POJO (*Plain Old Java Object*) que simule el patrón *Business Object* por cada subsistema. Este BO, cuando la aplicación pase a ser distribuida, se convertirá en el delegador del negocio (patrón *Business Delegate*). Para ello, del mismo modo que tenemos una factoría que hace de puerta de entrada a la capa de los datos, vamos a añadir otra factoría que nos sirve para entrar al negocio.



Capa de Negocio

Como ejemplo, vamos a refactorizar el *Action* de `LibroAddAccion` de modo que llame a la capa de negocio y le ceda el control.

Proyecto y Paquete

La capa de negocio es independiente de la capa de presentación, y por tanto no tenemos porque crear estos objetos en el proyecto web. Tiene mucho más sentido crearlos en el **proyecto común**. Además, colocaremos todas las clases dentro del paquete **`es.ua.jtech.proyint.bo`**

7.9.1. ILibroBO.java

Así pues, primero crearemos el interfaz con las operaciones de negocio

```
package es.ua.jtech.proyint.bo.libro;

// imports

/**
 * Operacion de Negocio de los Libros
 *
 * @author $Author: amedrano $
 * @version $Revision: 1.42 $
 */
public interface ILibroBO {

    /**
     * Selecciona un libro de la BD
     *
     * @param isbn Isbn del libro a seleccionar
     * @return Libro seleccionado
     * @throws LibroException
     */
    LibroEntity recuperaLibro(String isbn) throws LibroException;
```

```
    /**
     * Añade un libro en la BD
     *
     * @param libro Libro a añadir.
     * @throws LibroException
     */
    void anyadeLibro(LibroEntity libro) throws LibroException;

    /**
     * El libro pasado por parámetro es borrado de la BD
     *
     * @param isbn libro a eliminar
     * @return Número de registros afectados por el borrado
     * @throws LibroException
     */
    void eliminaLibro(String isbn) throws LibroException;

    /**
     * Devuelve una lista con todos los libros en la BD
     *
     * @return Lista con todos los libros
     * @throws LibroException
     */
    List<LibroEntity> listaLibros() throws LibroException;

    /**
     * Modifica los datos de un libro
     *
     * @param lib Libro a modificar
     * @throws LibroException
     */
    void actualizaLibro(LibroEntity lib) throws LibroException;
}
```

De este código podemos observar cómo los métodos lanzan la excepción `LibroException`. Hemos creado una excepción de aplicación para abstraer las capas posteriores del negocio. ¡La presentación no debe saber que hay detrás del lado oscuro!

7.9.1.1. LibroException.java

Así pues, por cada subsistema crearemos una o más excepciones de aplicación.

```
package es.ua.jtech.proyint.bo.libro;

import es.ua.jtech.proyint.BibliotecaException;

public class LibroException extends BibliotecaException {

    private static final long serialVersionUID =
6158132279964132104L;

    public LibroException() {
        super();
    }

    public LibroException(String message) {
```

```
        super(message);
    }

    public LibroException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

7.9.2. LibroBO.java

A partir de la interfaz, crearemos la implementación, en la cual colocaremos las reglas de negocio y realizará las llamadas a la capa de datos.

```
package es.ua.jtech.proyint.bo.libro;

// imports ...

/**
 * Operacion de Negocio de los Libros
 *
 * @author $Author: amedrano $
 * @version $Revision: 1.42 $
 */
public class LibroBO implements ILibroBO {

    public void actualizaLibro(LibroEntity libro) throws
LibroException {
        if (libro == null) {
            throw new IllegalArgumentException("Se esperaba un
libro");
        }

        ILibroDAO dao = FactoriaDAOs.getInstance().getLibroDAO();

        try {
            int numAct = dao.updateLibro(libro);
            if (numAct == 0) {
                throw new LibroException("No ha actualizado ningun
libro");
            }
        } catch (DAOException daoe) {
            throw new LibroException("Error actualizando libro",
daoe);
        }
    }

    public void anyadeLibro(LibroEntity libro) throws
LibroException {
        if (libro == null) {
            throw new IllegalArgumentException("Se esperaba un
libro");
        }

        ILibroDAO dao = FactoriaDAOs.getInstance().getLibroDAO();

        try {
            libro.setFechaAlta(new Date());
```

```
            dao.addLibro(libro);
        } catch (DAOException daoe) {
            throw new LibroException("Error insertando libro",
daoe);
        }

    }

    // resto de métodos

    public List<LibroEntity> listaLibros() throws LibroException {
        List<LibroEntity> result = null;
        ILibroDAO dao = FactoriaDAOs.getInstance().getLibroDAO();

        try {
            result = dao.getLibros();
        } catch (DAOException daoe) {
            throw new LibroException("Error listando libros",
daoe);
        }

        return result;
    }

}
```

La capa de negocio debe funcionar como puerta de entrada a la inteligencia de la aplicación, y por lo tanto, no debe dar por sentado que los datos de entrada son correctos. Al menos se debe comprobar que los parámetros llegan con valores rellenados, y dependiendo de la robustez deseada, duplicar algunas de las validaciones realizadas en los `ActionForm`.

Factoría de BOs

Del mismo modo que con los DAOs, hemos de crear una factoría de BOs encargada de desacoplar la creación de las implementaciones de sus interfaces.

7.9.3. LibroAddAccion.java

Como ejemplo, refactorizaremos este Action llevando la lógica de negocio a su correspondiente capa. Dentro del método `execute` teníamos:

```
// ...
// Recogemos los datos del formulario
LibroEntity libro = new LibroEntity();
BeanUtils.copyProperties(libro, (LibroForm) form);
libro.setFechaAlta(new Date());

// Anyadimos el libro
ILibroDAO dao = FactoriaDAOs.getInstance().getLibroDAO();
dao.addLibro(libro);
// ...
```

Al refactorizar y llevar la asignación de la fecha de alta y la invocación al DAO, nos

queda lo siguiente:

```
// ...
LibroEntity libro = new LibroEntity();
BeanUtils.copyProperties(libro, (LibroForm) form);

// Anyadimos el libro
ILibroBO bo = FactoriaBOs.getInstance().getLibroBO();
bo.anyadeLibro(libro);
// ...
```

Como ejercicio, se plantea refactorizar toda la aplicación para incluir la capa de negocio. Todas las comprobaciones que no sean de entradas de un Action se deben realizar en negocio.

7.10. Reserva de un Libro

Para comprobar el uso de la capa de negocio, vamos a repasar el flujo de llamadas. Para realizar una reserva desde el rol bibliotecario, primero hemos de elegir que libro se quiere reservar, para posteriormente, seleccionar que usuario va a realizar la reserva.

Alternativa

También podríamos hacerlo al revés, primero elegir el usuario y luego visualizar las opciones que puede realizar ese usuario

Así pues, en el siguiente ejemplo seleccionamos el libro de "*Agile Software Development*"

Selección del Libro a Reserva

Para poder mostrar las imágenes para **P**restar, **R**eservar, **A**nular **R**eserva, o **D**evolver **P**réstamo, tenemos que comparar para cada libro que estado tiene, y a partir de su estado, que acciones se pueden realizar.

El siguiente jsp (es la parte que falta de `listadoLibro.jsp`) muestra una forma de hacerlo:

```
// tras el td con los iconos de editar y borrar libro

<c:choose>
<c:when test="${libro.activa==null}">
```

```
    <html:link action="/opPrepararReserva" paramId="isbn"
        paramName="libro" paramProperty="isbn">
        <html:img page="/imagenes/r.gif" titleKey="libro.reservar"
/>
    </html:link>
    <html:link action="/opPrepararPrestamo" paramId="isbn"
        paramName="libro" paramProperty="isbn">
        <html:img page="/imagenes/p.gif" titleKey="libro.prestar"
/>
    </html:link>
</c:when>
<c:when test="${libro.activa.tipo eq 'reserva'}">
    <html:link action="/opReserva2Prestamo" paramId="idOperacion"
        paramName="libro" paramProperty="activa.idOperacion">
        <html:img page="/imagenes/p.gif" titleKey="libro.prestar"
/>
    </html:link>
    <html:link action="/opAnularReserva" paramId="idOperacion"
        paramName="libro" paramProperty="activa.idOperacion">
        <html:img page="/imagenes/ar.gif"
titleKey="libro.anularReserva" />
    </html:link>
</c:when>
<c:when test="${libro.activa.tipo eq 'prestamo'}">
    <html:link action="/opDevolverPrestamo" paramId="idOperacion"
        paramName="libro" paramProperty="activa.idOperacion">
        <html:img page="/imagenes/dp.gif"
titleKey="libro.devolverPrestamo" />
    </html:link>
</c:when>
</c:choose>
```

7.10.1. OpPreReservarAccion.java

Al pulsar sobre la R, le pasamos el control a `OpPreReservarAccion`, el cual se encarga de recuperar los datos del libro a reservar (para mostrárselos al usuario) y de los usuarios que pueden realizar reservas (socios y profesores no morosos).

```
ActionMessages errors = new ActionMessages();
ActionForward forward = mapping.getInputForward();

FactoriaBOs fd = FactoriaBOs.getInstance();
IOperacionBO io = fd.getOperacionBO();
ILibroBO il = fd.getLibroBO();

ReservaForm rf = (ReservaForm) form;

try {
    LibroEntity libro = il.recuperaLibro(rf.getIsbn());
    request.setAttribute(Tokens.RES_LIBRO, libro);
} catch (LibroException le) {
    logger.error("Error recuperando la info del libro a reservar",
le);
    errors.add(ActionMessages.GLOBAL_MESSAGE, new ActionMessage(
            Tokens.MSJ_LIBRO_GET_ERROR));
}
```

```
try {
    List<UsuarioEntity> usuarios =
io.listadoPosiblesPrestatarios();
    request.setAttribute(Tokens.RES_USUARIOS, usuarios);
    forward = mapping.findForward(Tokens.FOR_OK);
} catch (OperacionException ue) {
    logger.error("Error en el listado de los socios", ue);
    errors.add(ActionMessages.GLOBAL_MESSAGE, new ActionMessage(
            Tokens.MSJ_OP_LISTADO_PRESTATARIOS_ERROR));
}

if (errors.size() > 0) {
    this.saveErrors(request, errors);
}

return forward;
```

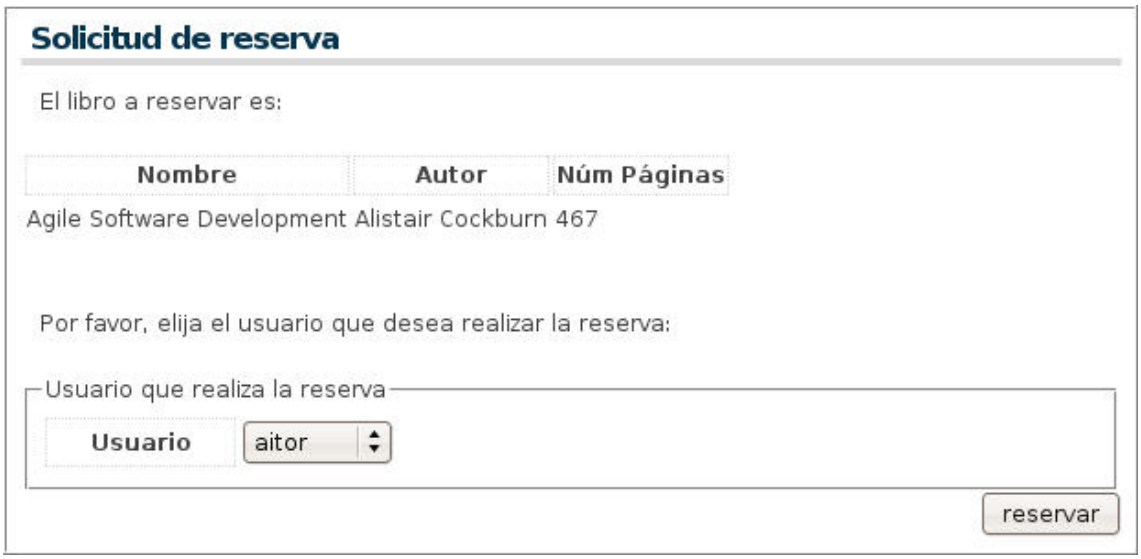
7.10.2. preparaReserva.jsp

El `Action` anterior nos redirigirá a la vista que se encarga de pintar los datos, a la espera de seleccionar el usuario y realizar la reserva:

```
<div id="content">
<h2><bean:message key="prepReserva.titulo"/></h2>

<logic:messagesPresent><div class="box"><html:errors
/></div></logic:messagesPresent>

<p><bean:message key="prepReserva.intro"/>:</p>
<table>
    <tr>
        <th><bean:message key="libro.titulo"/></th>
        <th><bean:message key="libro.autor"/></th>
        <th><bean:message key="libro.paginas"/></th>
    </tr>
    <tr>
        <td><bean:write name="libro" property="titulo" /></td>
        <td><bean:write name="libro" property="autor" /></td>
        <td><bean:write name="libro" property="numPaginas" /></td>
    </tr>
</table>

<p> </p>

<p><bean:message key="prepReserva.eligeUsuario"/>:</p>
<html:form action="/opRealizarReserva">
<html:hidden property="isbn" />
<fieldset>
<legend><bean:message key="prepReserva.usuarioReserva"/></legend>
<table>
    <tr><th width="100"><bean:message key="usuario"/></th>
    <td><html:select property="login">
        <html:options collection="usuarios" property="login" />
        </html:select>
    </td></tr>
</table>
</fieldset>
```

```
<div align="right">
    <html:submit><bean:message key="libro.reservar"/></html:submit>
</div>
</html:form>
<br />
</div>
```

El resultado es la siguiente vista:



Selección del Usuario

Al pulsar sobre reservar es cuando realmente se realiza la reserva.

7.10.3. OpRealizarReservarAccion.java

El formulario anterior le cede el control a este `Action`, el cual a su vez, cederá el control al negocio para realizar la reserva. Destacar como negocio esta lanzando 2 excepciones: la primera por si el usuario tiene el cupo de reserva llenas, y la segunda por si se produce cualquier otro error. Primero siempre hemos de poner las excepciones mas restrictivas, y de ahí, a las menos restrictivas y que abarquen un mayor abanico de errores.

Consejo

Es muy útil definir excepciones de aplicación para informar de las acciones anómalas de la aplicación. De ahi el nombre de excepción.

```
ActionMessages ams = new ActionMessages();
ActionForward forward = mapping.getInputForward();

FactoriaBOs fd = FactoriaBOs.getInstance();
IOperacionBO io = fd.getOperacionBO();

ReservaForm rf = (ReservaForm) form;
```

```
try {
    io.realizaReserva(rf.getLogin(), rf.getIsbn());

    logger.info("Reserva de " + rf.getLogin() + " para el libro "
            + rf.getIsbn() + " realizada");

    ams.add(ActionMessages.GLOBAL_MESSAGE, new ActionMessage(
            Tokens.MSJ_OP_RESERVA_REALIZADA, rf.getIsbn()));

    forward = mapping.findForward(Tokens.FOR_OK);
} catch (OperacionCupoCompletoException occe) {
    logger.error("Error en la reserva por cupo completo", occe);
    ams.add(ActionMessages.GLOBAL_MESSAGE, new ActionMessage(
            Tokens.MSJ_OP_CUPO_RESERVA_COMPLETO_ERROR,
rf.getLogin()));
} catch (OperacionException oe) {
    logger.error("Error en la reserva", oe);
    ams.add(ActionMessages.GLOBAL_MESSAGE, new ActionMessage(
            Tokens.MSJ_OP_RESERVA_ERROR));
}

if (ams.size() > 0) {
    this.saveErrors(request, ams);
}

return forward;
```

Todo el código que teníamos como método privado del `Action` pasa a ser el método de negocio dentro la implementación del interfaz `IOperacionBO`.

7.10.4. ReservaForm.java

Las 2 acciones comentadas anteriormente comparte un `ActionForm` para poder utilizar información como salida del primero y entrada del segundo.

```
package es.ua.jtech.proyint.struts.actionforms;

public class ReservaForm extends ValidatorForm {

    private static final long serialVersionUID =
4725787336428426243L;

    private String idOperacion;
    private String isbn;
    private String login;
    private List<UsuarioEntity> usuarios;

    // getters y setters
}
```

7.10.5. struts-config.xml

Las 2 acciones involucradas en el proceso de reserva se encadenan mediante la página de

preparaReserva.jsp, quedando su definición de mapeo de Actions del siguiente modo:

```
<action path="/opPreReservar"
type="es.ua.jtech.proyint.struts.acciones.operacion.OpPreReservarAccion"
    name="reservaForm" scope="request" validate="false"
    input="/libroList.do" roles="bibliotecario">
    <forward name="OK" path="/jsp/biblio/preparaReserva.jsp" />
</action>

<action path="/opReservar"
type="es.ua.jtech.proyint.struts.acciones.operacion.OpReservarAccion"
    name="reservaForm" scope="request" validate="false"
    input="/opPreReservar.do" roles="bibliotecario">
    <forward name="OK" path="/libroList.do" />
</action>
```

7.10.6. IOperacionBO

Las operaciones que tenemos que implementar en el proyecto web son las siguientes:

```
package es.ua.jtech.proyint.bo.operacion;

// imports

/**
 * Operaciones de Negocio
 *
 * @author $Author: amedrano $
 * @version $Revision: 1.42 $
 */
public interface IOperacionBO {

    String realizaReserva(String idUsuario, String idLibro)
            throws OperacionException,
OperacionUsuarioInvalidoException,
            OperacionLibroNoDisponibleException,
OperacionCupoCompletoException;

    String realizaPrestamo(String login, String isbn)
            throws OperacionException,
OperacionUsuarioInvalidoException,
            OperacionLibroNoDisponibleException,
OperacionCupoCompletoException;

    String pasaReserva2Prestamo(String idOperacion) throws
OperacionException,
            OperacionCaducadaException;

    void anulaReserva(String idOperacion) throws
OperacionException;

    void devuelvePrestamo(String idOperacion) throws
OperacionException,
            OperacionMultadaException;

    List<LibroEntity> listadoLibros() throws OperacionException;
```

```
    List<OperacionActivaEntity> listadoReservas() throws
OperacionException;

    List<OperacionActivaEntity> listadoPrestamos() throws
OperacionException;

    List<LibroEntity> listadoDisponibles() throws
OperacionException;

    List<UsuarioEntity> listadoPosiblesPrestatarios() throws
OperacionException;
}
```

Interfaz de Negocio

El interfaz de negocio no tiene porque tener una relación 1:1 con la de datos. En este ejemplo se demuestra como el negocio toma decisiones que luego la capa de datos aprovecha.

7.10.7. OperacionBO.realizaReserva(login, isbn)

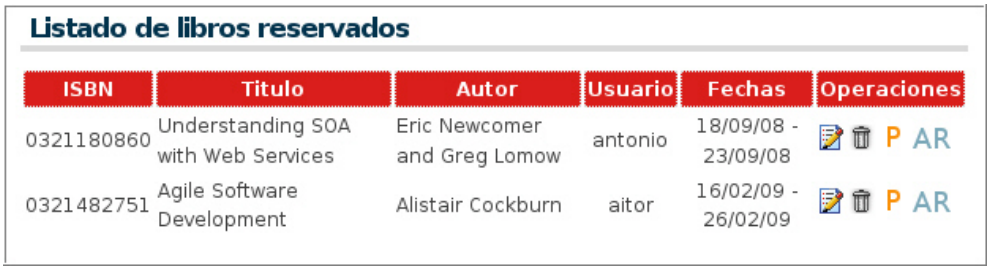
Respecto a la implementación de esta interfaz, sólo disponemos ya implementado del método de realizaReserva, que sería similar al siguiente:

```
public String realizaReserva(String login, String isbn)
        throws OperacionException, OperacionCupoCompletoException,
        OperacionUsuarioInvalidoException,
OperacionLibroNoDisponibleException {

    if (login == null || isbn == null) {
        throw new IllegalArgumentException("Se esperaba un login y
un isbn");
    }

    String result = null;

    UsuarioEntity usuario = null;
    try {
        usuario =
FactoriaDAOs.getInstance().getUsuarioDAO().getUsuario(
                login);
    } catch (DAOException daoe) {
        throw new OperacionException(
                "Error recuperando datos del usuario que realiza la
reserva",
                daoe);
    }

    // Comprobamos que el usuario no sea moroso, y que sea socio o
profesor
    if ((usuario.getTipo() != TipoUsuario.profesor &&
usuario.getTipo() != TipoUsuario.socio)
            || usuario.getEstado() == EstadoUsuario.moroso) {
        throw new OperacionUsuarioInvalidoException(
                "Este usuario no puede hacer una reserva");
    }
```

```
    // Comprobamos que el usuario no tiene lleno el cupo de
reservas
    IOperacionDAO oDao =
FactoriaDAOs.getInstance().getOperacionDAO();
    int numOp = 0;
    try {
        numOp = oDao.countOperacionesActivas(login);
    } catch (DAOException daoe) {
        throw new OperacionException("Error recuperando reservas
activas",
                daoe);
    }

    try {
        BibliotecaBR.getInstance().compruebaCupoOperaciones(
                usuario.getTipo(), numOp);
    } catch (BibliotecaException e) {
        throw new OperacionCupoCompletoException(
        "El cupo de reservas posibles esta lleno");
    }

    // Comprobamos que el libro esta disponible
    try {
        if (!oDao.isLibroDisponible(isbn)) {
            throw new OperacionLibroNoDisponibleException(
                    "Libro no disponible para crear una reserva");
        }
        ILibroDAO lDao = FactoriaDAOs.getInstance().getLibroDAO();
        if (lDao.getLibro(isbn) == null) {
            throw new OperacionLibroNoDisponibleException("El libro
solicitado no existe");
        }
    } catch (DAOException daoe) {
        throw new OperacionException(
                "Error comprobando disponibilidad del libro",
daoe);
    }

    // Calculamos el numero de dias que tiene validez la reserva
    Calendar cal = GregorianCalendar.getInstance();
    Date ahora = cal.getTime();

    int dias = 0;
    try {
        dias = BibliotecaBR.getInstance().calculaNumDiasReserva(
                usuario.getTipo());
    } catch (BibliotecaException e) {
        throw new OperacionException(
                "Error al calcular la fecha de fin de la reserva",
e);
    }

    cal.add(Calendar.DATE, dias);
    Date ffin = cal.getTime();

    // Damos de alta la reserva
    OperacionActivaEntity op = new OperacionActivaEntity(login,
isbn,
```

```
            TipoOperacion.reserva, ahora, ffin);

    try {
        result = oDao.addOperacion(op);
    } catch (DAOException daoe) {
        throw new OperacionException("Error realizando reserva",
daoe);
    }

    return result;

}
```

El resultado final es una nueva reserva:



Reserva Realizada

7.11. Resto de Operaciones

En cuanto al resto de operaciones, vamos a detallar en una tabla las tareas que conlleva cada operación:

Anular Reserva
Eliminar tanto la operación, como la operación activa

Realizar Préstamo
Igual que la reserva, pero cambia el cálculo de la fecha de devolución.

Devolver Préstamo
Pasa la operación de activa a histórico (recordar que hay que rellenar el campo `fechaFinReal`). Comprueba si la devolución conlleva una multa. En ese caso, crea una nueva operación de tipo multa con fecha de finalización dependiendo del rol del usuario, y se el cambiar el estado del usuario a moroso.

Pasar una Reserva a Préstamo
Comprobamos que la reserva no ha caducado y que realmente es una reserva. A continuación, pasamos la reserva de activa a histórico. Tras calcular la fecha de devolución, creamos e insertamos una nueva operación de tipo préstamo.

Queda pendiente...

Un parte importante es la anulación automática de las reservas. Para ello necesitamos la ejecución periódica de proceso que compruebe que reservas han caducado, y consecuentemente,

anule dichas reservas. Veremos como podemos hacer esto en el módulo de EJB.

7.12. Guia de Estilo

Toda la aplicación debe hacer un uso consistente de los taglibs de *Struts*, tanto a nivel de formularios con sus correspondientes `ActionForms` como referencias a enlaces, tratamiento de errores, etc...

Todas las clases Java deben contener una cabecera *Javadoc* indicando el propósito de la clase, y los atributos javadoc de `author` y `revision` con los valores del CVS. Por ejemplo, en la interfaz `Tokens.java` tenemos:

```
/**
 * Interfaz con las constantes utilizadas dentro de los actions
 *
 * @author $Author: amedrano $
 * @version $Revision: 1.42 $
 */
```

Siempre que sea posible, se minimizará el acoplamiento entre las diferentes capas y objetos, pasando como parámetros el menor número de información posible.

Además, el proyecto deberá incluir únicamente el código fuente de la funcionalidad desarrollada. Todo los desarrollos previos (*Servlets*, *AccionPrueba*, etc...) debe ser eliminados del proyecto.

No debe haber ningún mensaje de error o advertencia en la pestaña *Problems* de Eclipse (excepto el de aviso del uso de librerías en el proyecto común). Para ello, se debe revisar que todo el código es correcto, así como objetos declarados y no utilizados, imports de sobra, etc...

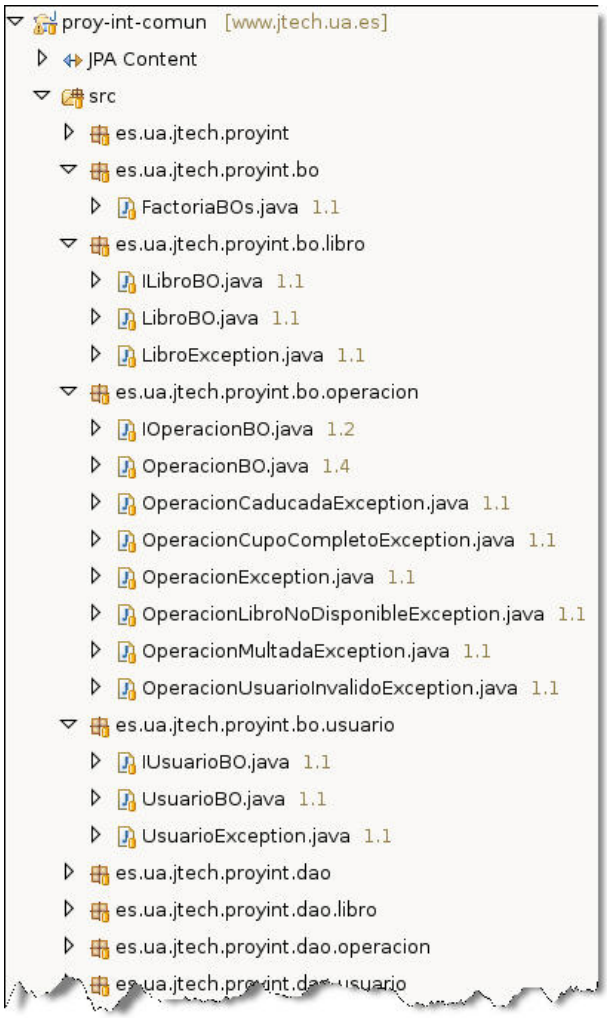
7.13. Entrega

La entrega se realizará por parejas. Si alguien quiere implementar la aplicación de forma individual, no se valorará el trabajo extra. El objetivo del proyecto es mejorar la calidad, trabajar en equipo (modelado ágil y programación en parejas), y aprender. Recordar que 2 cerebros son mejor que 1 :)

Al finalizar el proyecto web, deben quedar implementadas las webs del bibliotecario y del administrador, en ambas con el ciclo completo de Action -> Capa de Negocio -> DAO con JPA.

7.13.1. Proyecto Común

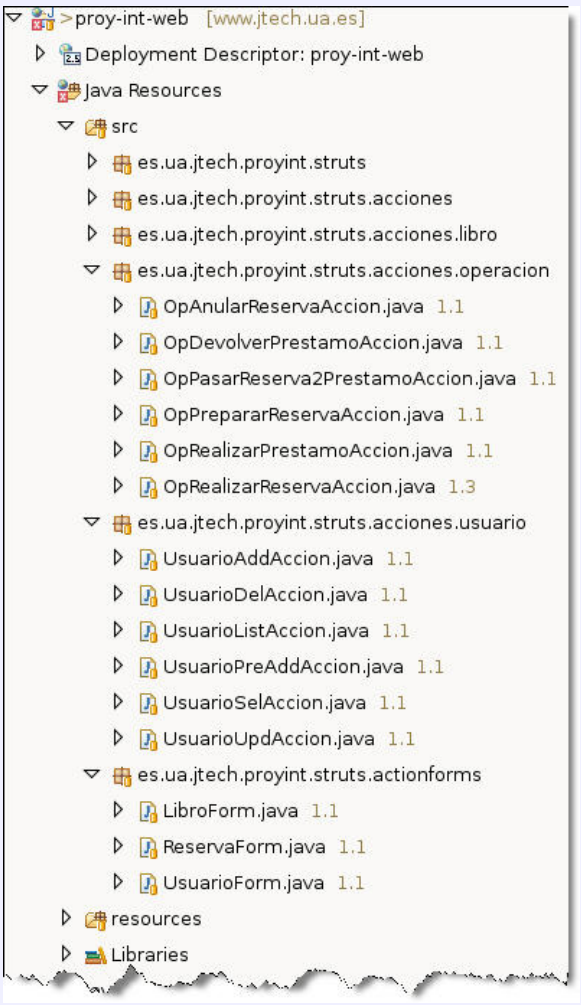
A continuación se muestra un imagen del contenido a entregar dentro del proyecto común.



Contenido Proyecto Común

7.13.2. Proyecto Web

Para este proyecto, mostramos 2 imágenes, una con el contenido del código Java y la otra con el contenido web.



Contenido Java Proyecto Web

Contenido Web Proyecto Web

Se plantea como optativos aquellos ejercicios planteados en rojo a lo largo del presente documento.

8. Integración con el servidor de aplicaciones

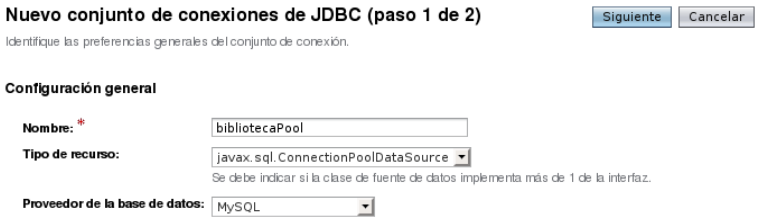
En esta sesión de integración integraremos el trabajo realizado en el servidor de aplicaciones GlassFish.

8.1. Configuración del dominio GlassFish

Vamos a utilizar el dominio por defecto *domain1* que se crea al instalar GlassFish. Lo tenemos instalado con los ejercicios del servidor de aplicaciones.

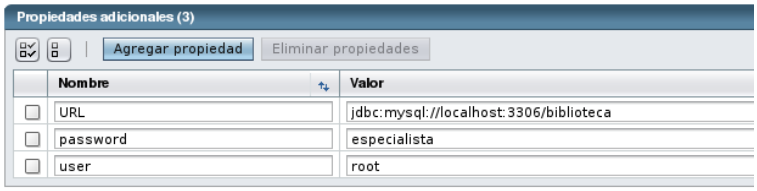
En primer lugar debemos crear la fuente de datos JNDI que proporcionará a la aplicación el acceso a la base de datos. Debemos crearla de la misma forma que vimos en la sesión del servidor de aplicaciones. Recordad que esto implica dos pasos:

- Crear el *connection pool* o conjunto de conexiones a la BD. En *Recursos* > *JDBC* > *Conjuntos de conexiones* crearemos un nuevo `bibliotecaPool`. En el **Paso 1** especificamos las propiedades básicas.



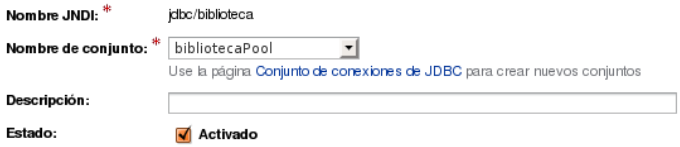
Conjunto de conexiones

Al final del **paso 2**, eliminamos todas las "propiedades adicionales" y añadimos las necesarias para conectar con la BD



Propiedades adicionales del conjunto de conexiones

- Ahora podemos crear el Datasource accesible mediante JNDI. Se hará desde la opción *Recursos* > *JDBC* > *Recursos JDBC*. Usaremos como nombre JNDI *jdcb/biblioteca*. Por supuesto, referenciamos el conjunto de conexiones que hemos creado antes.



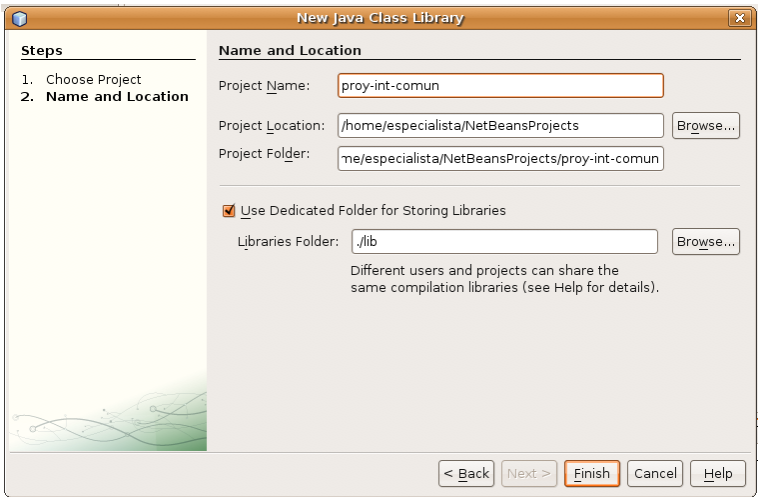
Datasource JNDI

8.2. Creación del EAR en NetBeans

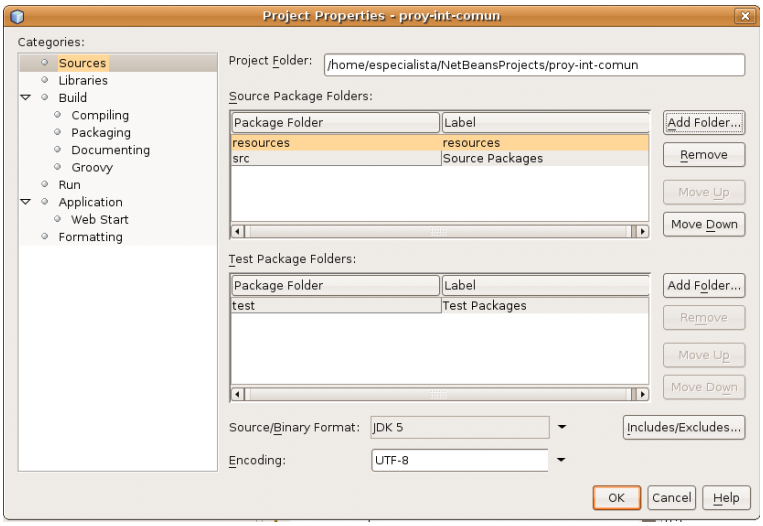
En este segundo paso vamos a configurar los proyectos en el IDE NetBeans, creando un EAR que desplegaremos en el servidor de aplicaciones. En el EAR incluiremos todas las librerías usadas por el proyecto común, el propio proyecto común y el proyecto web. En el WAR del proyecto web sólo deberemos incorporar las librerías específicas. Vamos a verlo paso a paso.

8.2.1. Configuración del proyecto común

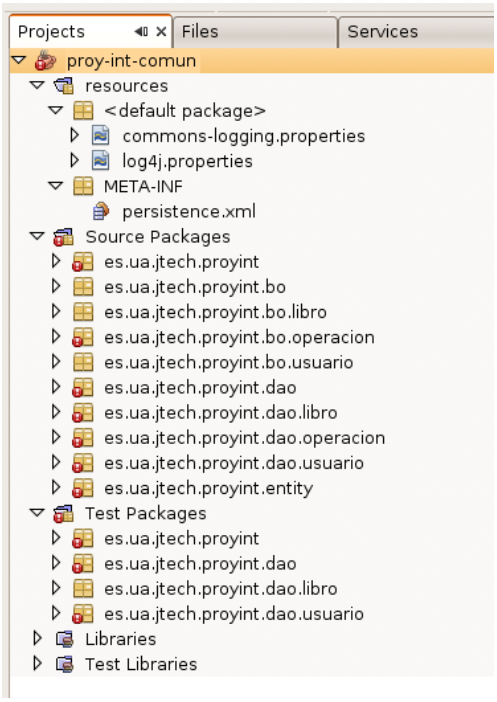
Creamos en NetBeans un proyecto nuevo de tipo *Java Class Library* llamado *proy-int-comun*. Activamos la creación del directorio *./lib* en el que guardamos las librerías:



Creamos también una carpeta llamada *resources* en la que guardaremos los ficheros de configuración. Como deben estar accesibles en el classpath, configuramos esta carpeta como una carpeta de fuentes, entrando en la opción *Properties* del proyecto y seleccionando la opción *Sources*:



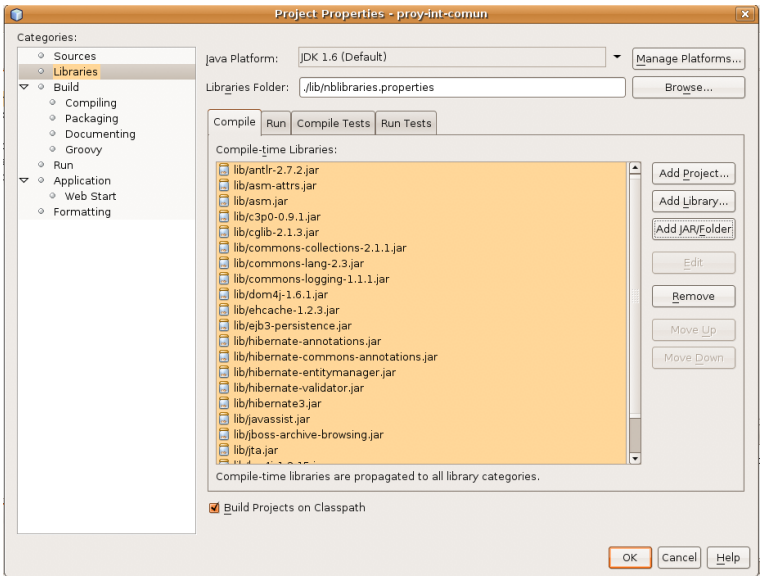
Copiamos en la carpeta *src* los fuentes del proyecto de integración, en la carpeta *resources* los ficheros y directorios incluidos en la carpeta original de Eclipse y en la carpeta *test* los tests:



Los errores se deben a que todavía no hemos incorporado las librerías. Para hacerlo, seleccionamos la vista *Files* y copiamos en la carpeta `lib` todas las librerías usadas por el proyecto común, incluidas todas las de Hibernate.

Una vez hecho esto, entramos en las propiedades del proyecto, seleccionamos la opción

Libraries y el botón *Add JAR/Folder* para incluir las librerías en el classpath de compilación del proyecto:

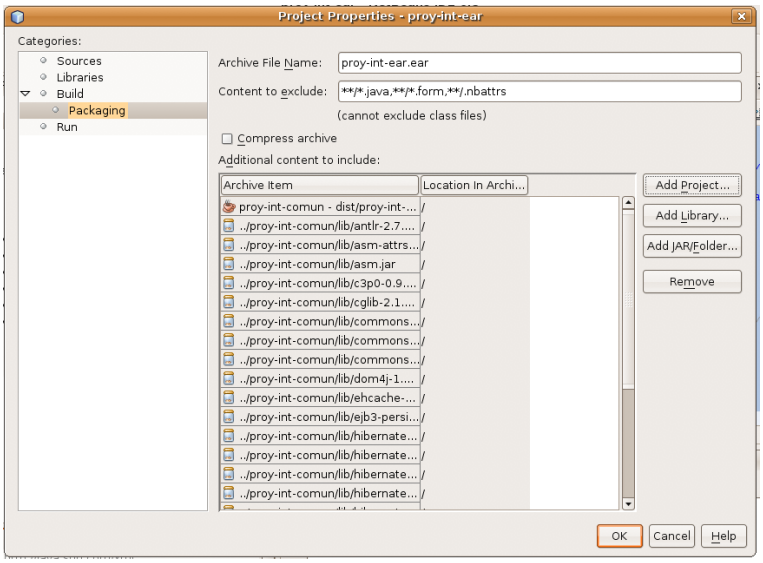


Los errores de compilación deben haber desaparecido. Lanzamos los tests para comprobar que funciona correctamente la librería JPA.

Una vez lanzados los tests y comprobado que todo funciona correctamente, modificamos el fichero *persistence.xml* para que se conecte con la fuente de datos definida en el dominio. Debe quedar como sigue:

```
<persistence-unit name="Biblioteca">
<non-jta-data-source>jdbc/biblioteca</non-jta-data-source>
   <class>es.ua.jtech.proyint.entity.UsuarioEntity</class>
<class>es.ua.jtech.proyint.entity.OperacionHistoricoEntity</class>
   <class>es.ua.jtech.proyint.entity.OperacionEntity</class>
   <class>es.ua.jtech.proyint.entity.OperacionActivaEntity</class>
   <class>es.ua.jtech.proyint.entity.MultaEntity</class>
   <class>es.ua.jtech.proyint.entity.LibroEntity</class>
   <properties>
      <property name="hibernate.archive.autodetection"
                value="class, hbm" />
     <!-- Comentamos la conexión directa
          <property name="hibernate.connection.driver_class"
                value="com.mysql.jdbc.Driver" />
      <property name="hibernate.connection.url"
                value="jdbc:mysql://localhost:3306/biblioteca" />
      <property name="hibernate.connection.username"
                value="root" />
      <property name="hibernate.connection.password"
                value="especialista" />
      -->
      <property name="hibernate.c3p0.min_size"
                value="5" />
```

Por último creamos un proyecto EAR llamado *proy-int-ear* sin ningún módulo asociado. Se los añadiremos a continuación, entrando en las propiedades, y añandiendo en la opción *Packaging* el proyecto *proy-int-comun* y todas sus librerías por separado. De esta forma los empaquetará en el fichero EAR:



Puedes comprobar con el navegador de archivos que se ha creado el fichero *proy-int-ear.ear* y que contiene todos los JAR con las librerías y el JAR con el proyecto común.

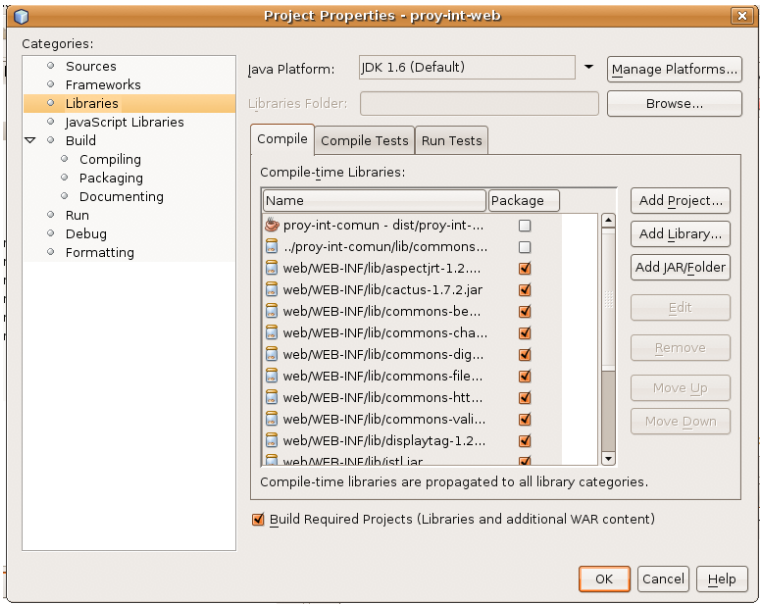
8.2.2. Configuración de la aplicación web

Procedemos a crear el proyecto web. Creamos un proyecto nuevo, esta vez un proyecto *web*, con nombre *proy-int-web*. **Lo añadimos también al EAR, utilizando la opción correspondiente en el asistente de creación**.

Ahora tenemos que copiar los ficheros del proyecto web en el nuevo proyecto en Netbeans.

- Copiamos los fuentes en *Source Packages*.
- Creamos un directorio de fuentes *resources* y copiamos en él los ficheros de recursos.
- Copiamos en la carpeta *Web Pages* todos los ficheros y carpetas del directorio *WebContent* que teníamos en Eclipse.
- Copiamos en la carpeta *WEB-INF/lib* todas las librerías exclusivas del proyecto web. No copiamos las que ya están en el EAR.

Modificamos las propiedades del proyecto web para incluir las librerías en el classpath. Incluimos todas las que hemos copiado en el directorio *WEB-INF/lib* (y las marcamos para que se empaqueten en el WAR) y las necesarias del proyecto común (sin marcar que se incluyan en el WAR, porque ya se empaquetan en el EAR):



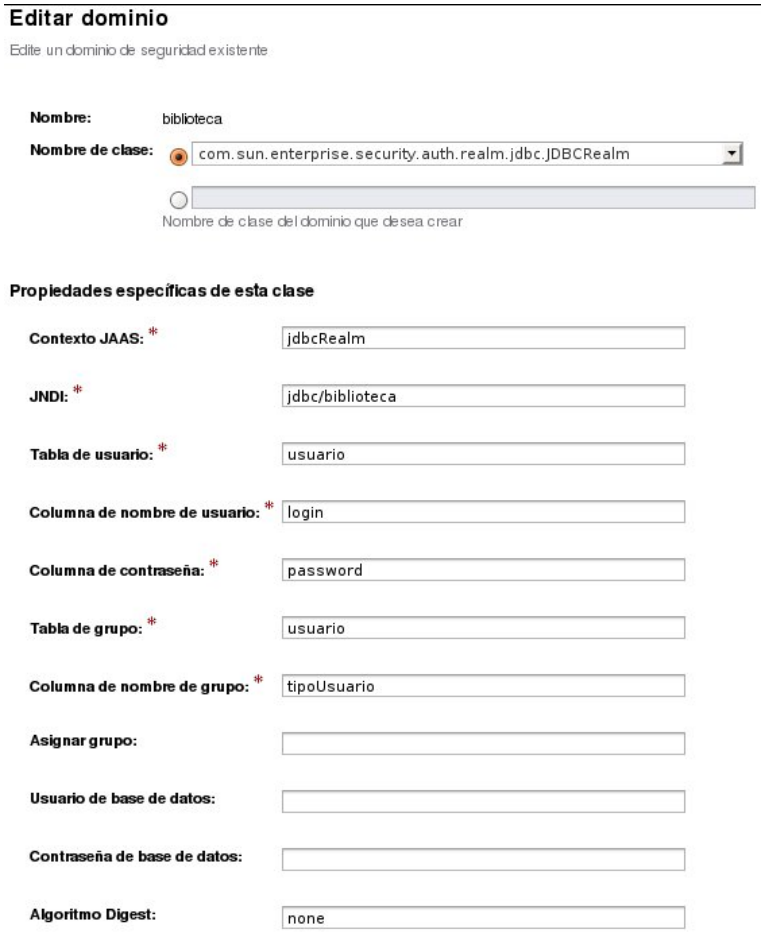
En este momento no tiene que haber errores de compilación y el proyecto está listo para desplegarse en el servidor de aplicaciones. Podemos probar a ejecutarlo pulsando el botón derecho sobre el EAR y seleccionando la opción *Run*. El navegador mostrará la página principal, pero no será posible autentificarse porque todavía falta configurar la seguridad en el servidor de aplicaciones.

8.3. Configuración del la seguridad

En el proyecto desarrollado para Tomcat, los usuarios y contraseñas, junto con los roles, se cogían de un *realm* definido por una fuente de datos. Aquí vamos a hacer lo mismo, pero la definición de ese realm es distinta. En Tomcat se definía en el fichero *context.xml* en el directorio META-INF. Ahora tenemos que hacerlo con Glassfish. Para ello, nos tenemos que ir a Configuración->Seguridad->Dominios. Vamos a crear un nuevo dominio que estará definido por JDBC. Pinchamos en nuevo y seleccionamos un realm JDBC. Las opciones que se pueden configurar son:

- Nombre: el nombre que usaremos para invocar a este realm.
- Contexto JAAS: siempre jdbcRealm.
- JNDI: nombre de la fuente de datos. Como ya la tenemos definida antes, la usamos.
- Tabla de usuario: nombre de la tabla que contiene la información de usuario. En nuestro caso, `usuario`
- Columna de nombre de usuario: nombre de la columna que contiene el login. En nuestro caso, `login`.
- Columna de contraseña: nombre de la columna de contraseña.En nuestro caso, `password`
- Tabla de grupo: nombre de la tabla que contiene los roles. En nuestro caso,es la

misma tabla con los usuarios, `usuario`
- Columna de grupo: la columna que contiene los roles. Para nosotros, `tipoUsuario`
- Algoritmo Digest: indicamos si usamos algún algoritmo para codificar la contraseña. Como nosotros no tenemos la contraseña encriptada, usamos `none`



Dominio de seguridad

Una vez introducida la información, aceptamos los datos y ya tenemos creado este tipo de autentificación.

Nos falta configurar el proyecto web para que use este realm. Abrimos el **fichero web.xml** dentro de *Web Pages->WEB-INF*. Tenemos que eliminar todas los mapeos de servlets, pues nos los usamos. Desde el servlet-name LibroDelServlet asociado a es.ua.jtech.proyint.servlet.libro.LibroDelServlet hasta el comentario de *Seguridad*. No afecta, pero Glassfish da una excepción al no encontrar las clases asociadas a los servlets (que de hecho no están).

Una cosa que no hicimos en Tomcat es indicar al formulario (seguimos en el fichero web.xml) qué realm tiene que usar para autentificar, puesto que Glassfish permite crear

varios. Para ellos, añadimos un dato en el formulario (realm-name) para indicar el nombre del realm a usar. Tiene que ser el que hayamos indicado en la creación del realm.

```
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>RealmBiblioteca</realm-name>
  <form-login-config>
        <form-login-page>/login.jsp</form-login-page>
        <form-error-page>/loginError.jsp</form-error-page>
  </form-login-config>
</login-config>
```

Como veremos, Netbeans ha creado el fichero **descriptor adicional propio de glassfish, *sun-web.xml***. Tenemos que añadir la información que mostramos abajo para mapear los roles. En el fichero web.xml hemos definido las restricciones de seguridad y hemos asociado roles a esas restricciones. También hemos indicado que el formulario use el realm definido en Glassfish para buscar tanto los usuarios y contraseñas como los roles asociados. El mapeo que se indica más abajo asocia un rol definido en web.xml con el grupo (o rol) al que pertenece el usuario (pueden ser varios). En este caso coinciden los roles y los grupos, pero no tiene que ser necesariamente así.

```
(cabecera del sun-web.xml)
...
<security-role-mapping>
  <role-name>administrador</role-name>
  <group-name>administrador</group-name>
</security-role-mapping>

<security-role-mapping>
  <role-name>socio</role-name>
  <group-name>socio</group-name>
</security-role-mapping>

<security-role-mapping>
  <role-name>profesor</role-name>
  <group-name>profesor</group-name>
</security-role-mapping>

<security-role-mapping>
  <role-name>bibliotecario</role-name>
  <group-name>bibliotecario</group-name>
</security-role-mapping>
...
```

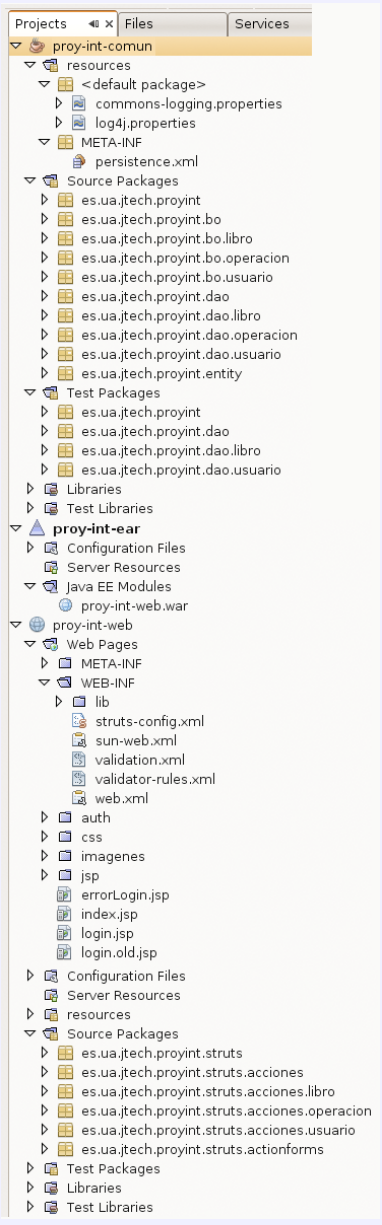
Hay un último problema. En Glassfish, **el método `isUserinRole`**, que nos dice si el usuario actual tiene determinado rol, **requiere la declaración de los posibles roles con una anotación especial** en la clase que los referencie. Como hacemos uso de este método en la clase `LoginAccion` tendremos que modificarla para declarar los roles .

```
@DeclareRoles({"administrador","bibliotecario","socio","profesor"})
public class LoginAccion extends Action {
```

```
        @Override
        public ActionForward execute(ActionMapping mapping,
ActionForm form,
                        HttpServletRequest request,
HttpServletResponse res)
                        throws Exception {
                        ...
                        ...
        }
}
```

Ya podemos ejecutar y/o desplegar el proyecto. Lo hacemos y arrancará el servidor, desplegará la aplicación y abrirá la aplicación en un navegador.

8.4. Entrega



Se deberán realizar las siguientes modificaciones en los proyectos común y web:

- Configurar la fuente de datos en el dominio de GlassFish.
- Crear los dos proyectos y el EAR en NetBeans.
- Copiar los fuentes y carpetas auxiliares a los nuevos proyectos.
- Modificar el fichero persistence.xml en el proyecto común.
- Configurar las librerías en ambos proyectos.
- Crear el realm en Glassfish.
- Adaptar el proyecto web para que use el realm anterior.

Table 1: Modificaciones a realizar

9. Enterprise JavaBeans

9.1. Introducción

El objetivo de esta sesión de integración es incorporar una capa de EJB que asuma la lógica de negocio que implementa actualmente el proyecto común en las clases BOs y DAOs. El objetivo final será sustituir en la capa web las llamadas a los BOs por llamadas a EJBs.

El proyecto web terminó con los DAOs implementando operaciones atómicas y siendo llamados por los BOs. El problema que se presenta es que en los métodos implementados por los BOs se realizan distintas llamadas a varios métodos de los DAOs sin asegurar la transaccionalidad. Vamos hacer que la capa EJB se encarge de esta gestión. De está forma, además de asegurar la transaccionalidad, abrimos la posibilidad de acceso remoto y seguro a las funcionalidades de la capa de negocio. Para ello tenemos que realizar los siguientes pasos:

- Añadir en el proyecto común una nueva implementación de los DAO JPA que utilice un contexto de persistencia gestionado por el contenedor y el mecanismo de JTA de gestión de transacciones.
- Crear tres EJB: `OperacionBOBean`, `LibroBOBean` y `UsuarioBOBean` que implementen las interfaces definidas en el paquete común y que utilicen las nuevas clases DAO.
- Sustituimos en la capa web las llamadas a los BO del paquete común por llamadas a los nuevos EJB.

9.2. Configuración de JPA

Comenzamos por modificar la configuración de JPA. La versión anterior en la que utilizábamos JPA (el proyecto de integración web) se basaba en el uso de JPA gestionado por la aplicación. Allí se utilizaba una clase singleton (la clase `FactoriaDAOs`) para inicializar la factoría de entity managers. El código era el siguiente:

```
private FactoriaDAOs() {
   //Obtenemos el emf
   this.emf = Persistence.createEntityManagerFactory("Biblioteca");
}
```

Ahora, sin embargo, vamos a utilizar JPA en una aplicación enterprise, y los entity managers y sus contextos de persistencia van a estar gestionado por el contenedor EJB. Para poder utilizar JPA desde el contenedor EJB tenemos que añadir en el fichero `persistence.xml` una nueva unidad de persistencia. La llamamos `BibliotecaEE`, copiando todas las propiedades de la unidad de persistencia ya existente y modificando las características de la fuente de datos:

```
<persistence-unit name="BibliotecaEE" transaction-type="JTA">
```

```
<provider>org.hibernate.ejb.HibernatePersistence</provider>
<jta-data-source>jdbc/biblioteca</jta-data-source>
<class>es.ua.jtech.proyint.entity.UsuarioEntity</class>
...
```

Con el elemento `provider` le indicamos a GlassFish que no utilice la implementación de JPA que trae de fábrica (TopLink de Oracle), sino las librerías proporcionadas de Hibernate.

Esta nueva unidad de persistencia será la que utilicemos en los EJB.

9.3. Nuevas clases DAO

Las clases DAO existentes tratan los entity managers como conexiones JDBC. En el código de cada una de las operaciones de los DAO se obtiene el entity manager, se abre una transacción, se realizan las operaciones de JPA y se cierra la transacción y el entity manager:

```
public String addOperacion(OperacionEntity operacion)
                                    throws DAOException {
   String result;
   EntityManager em = emf.createEntityManager();
   try {
      em.getTransaction().begin();
      em.persist(operacion);

      result = operacion.getIdOperacion();

      em.getTransaction().commit();
   } catch (PersistenceException e) {
      if (em.getTransaction().isActive())
         em.getTransaction().rollback();
      throw new
         DAOException("Error de JPA al anyadir una operacion", e);
   } finally {
      em.close();
   }

   return result;
}
```

Ahora tenemos que cambiar de enfoque, ya que, al estar JPA gestionado por el contenedor, no vamos a poder crear y cerrar entity managers. Esto va ser responsabilidad del contenedor. Vamos a obtener el entity manager en los EJB y pasarlos como parámetros en la creación de los DAO.

Al no cerrar el entity manager, tenemos la ventaja adicional de que el contexto de persistencia va a poder ser compartido entre más de un DAO o más de una operación del DAO. Podremos además utilizar el DAO en cualquier tipo de contexto de persistencia: de ámbito de transacción o extendido.

1. En concreto, hay que añadir unas nuevas clases DAO en las que no se gestiona

explícitamente ni el entity manager ni las transacciones, sino que se utilizará como entity manager la variable `em` inicializada en la creación del DAO. Llamamos a estas nuevas clases `OperacionJPAEEDAO`, `LibroJPAEEDAO` y `UsuarioJPAEEDAO`. Copiamos algo de código y debes completar el resto:

```
public class OperacionJPAEEDAO implements IOperacionDAO {
   EntityManager em;

   public OperacionJPAEEDAO(EntityManager em) {
      this.em = em;
   }

   public String addOperacion(OperacionEntity operacion)
                                          throws DAOException {
      String result;
      try {
         em.persist(operacion);
         result = operacion.getIdOperacion();
      } catch (PersistenceException e) {
         throw new
            DAOException("Error de JPA al anyadir una operacion",
e);
      }
      return result;
   }

   public int countOperacionesActivas(String login)
                                          throws DAOException {
      int result;

      String sql =
 "SELECT COUNT(o) FROM OperacionActivaEntity o WHERE
o.usuario.login=:login";

      Query q = em.createQuery(sql);
      q.setParameter("login", login);

      result = ((Long) q.getSingleResult()).intValue();
      return result;
   }
   ...
   public int updateOperacion(OperacionEntity operacion)
                                          throws DAOException {
      int result = 0;

      try {
         OperacionEntity usu =
em.getReference(OperacionEntity.class,
               Integer.parseInt(operacion.getIdOperacion()));
         if (usu != null) {
            em.merge(operacion);
            result = 1;
         }
      } catch (PersistenceException e) {
         throw new
            DAOException("Error de JPA al modificar una operacion",
```

```
e);
      }
      return result;
   }
   ...
}]
```

Es interesante hacer notar que los métodos devuelven entidades gestionadas. También es interesante repasar el planteamiento del método `updateOperacion`. Recibe una entidad `operacion` que puede o no estar gestionado por el contexto de persistencia actual. La forma de descubrirlo es con la llamada a `getReference`. Si no está gestionado, se convierte en gestionado llamando al método `merge` del entity manager. Y se realizará la actualización cuando decida el contenedor (en nuestro caso, como se va a gestionar desde un bean, cuando se cierre la transacción al terminar el método de negocio del bean).

Hay que completar el resto de métodos y de clases siguiendo este patrón. Básicamente se trata de copiar los métodos y clases ya definidos y eliminar de ellos la obtención las operaciones con el entity manager.

2. Además hay que añadir en la factoría de DAOs los nuevos métodos que devuelven los nuevos DAOs definidos anteriormente. Para ello, creamos tres nuevos métodos de obtención de los DAOs pasando como parámetro un entity manager. Dejamos los métodos ya existentes para mantener la compatibilidad con las clases ya definidas:

```
public class FactoriaDAOs {
   private static FactoriaDAOs me = new FactoriaDAOs();

   private FactoriaDAOs() { }

   public static FactoriaDAOs getInstance() {
      return me;
   }

   public ILibroDAO getLibroDAO() {
          return new LibroJPADAO();
   }

    public ILibroDAO getLibroDAO(EntityManager em) {
           return new LibroJPAEEDAO(em);
    }

   public IOperacionDAO getOperacionDAO() {
      return new OperacionJPADAO();
   }

   // Hay que añadir los otros métodos que devuelven los
   // nuevos DAOs OperacionJPAEEDAO y UsuarioJPAEEDAO
}
```

9.4. Creación de la capa EJB y de la factoría de BOs en la capa EJB

Tal y como hemos comentado en la introducción, el objetivo principal de esta sesión del

proyecto de integración es la inclusión de una capa de EJB que implemente la misma lógica de negocio de la clases BO del proyecto de integración. Vamos ahora a crear esta capa, sus EJB y la clase factoría que los develve.

En la siguiente descripción comentamos los pasos a seguir para el EJB `OperacionBOBean`. Deberemos hacer lo mismo después con los beans `LibroBOBean` y `UsuarioBOBean`.

1. Crea el proyecto EJB `proy-int-ejb` añadiéndolo al EAR. Configura sus propiedades para que referencie al proyecto común.

2. Crea el bean de sesión `OperacionBOBean` en el paquete `es.ua.jtech.proyint.bean`. Haz que tenga ambas interfaces (local y remota).

3. En la interfaz local extiende la interfaz `IOperacionBO` definida en el proyecto común. De esta forma obligamos al bean a implementar todos los métodos de la interfaz. En la interfaz remota define únicamente el método:

```
String realizaReservaRemota(String idUsuario, String idLibro)
                    throws OperacionException,
OperacionCupoCompletoException;
```

4. En la implementación del bean obtenemos el entity manager por inyección de dependencias, utilizando la unidad de persistencia definida anteriormente. También definimos el tipo de gestión de transacción definido por el contenedor.

```
@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
public class OperacionBOBeanBean implements OperacionBOBeanRemote,
OperacionBOBeanLocal {

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

    @Resource
    private SessionContext context;

    // Métodos
    ...
```

5. En la implementación del bean completamos automáticamente todos los métodos de la interfaz. Verás que aparecen todos los métodos definidos en `IOperacionBO` con una implementación vacía. Debemos implementar todos los métodos de la interfaz, copiando el código ya existente en los objetos de negocio del paquete común, pero modificando la obtención de los DAO para conseguir las nuevas implementaciones que acabamos de programar. Comenzamos, por ejemplo, por el método `anulaReserva`:

```
    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void anulaReserva(String idOperacion) throws
OperacionException {
      if (idOperacion == null) {
```

```
         throw new IllegalArgumentException(
               "Se esperaba identificador de la operacion");
      }
      IOperacionDAO dao =
FactoriaDAOs.getInstance().getOperacionDAO(em);
      try {
         dao.delOperacion(idOperacion);
      } catch (DAOException daoe) {
         context.setRollbackOnly();
         throw new OperacionException("Error anulando reserva",
daoe);
      }
    }
    ...
```

El código es exactamente el mismo que el ya definido en la implementación del objeto de negocio, añadiendo el atributo de transacción a REQUIRED y la llamada a `setRollbackOnly()` del contexto de sesión para marcar la transacción actual como inválida.

6. Ahora debemos definir una forma de que la capa web obtenga el bean recién creado. La capa web obtiene los objetos de negocio mediante una factoría. Vamos a definir una factoría en la capa EJB que devuelva enterprise beans. Creamos la clase `FactoriaBOBeans` en el mismo paquete que el bean y en el proyecto EJB. El código es el siguiente:

```
public class FactoriaBOBeans {

   private static FactoriaBOBeans me = new FactoriaBOBeans();

   private FactoriaBOBeans() {
   }

   public static FactoriaBOBeans getInstance() {
      return me;
   }

   public IOperacionBO getOperacionBOBean() {
        try {
            InitialContext ic = new InitialContext();
            return (OperacionBOBeanLocal)
                    ic.lookup("java:comp/env/OperacionBOBean");
        } catch (NamingException ex) {
            ex.printStackTrace();
            throw new RuntimeException();
        }
   }
}
```

Esta factoría deberá devolver un acceso local al bean `OperacionBO`. Para ello busca en el contexto JNDI local el nombre `java:comp/env/OperacionBOBean`. El nombre lógico del bean en esa expresión es `OperacionBOBean`. Para darle un nombre lógico a un bean podemos usar anotaciones en el componente en el que vamos a ejecutar la llamada a JNDI. En este caso es un componente web, ya que la llamada al método anterior

`getOperacion()` la haremos desde la capa web. El problema que se nos plantea es que la capa web está implementada en Struts, y no hay forma de utilizar anotaciones. Debemos entonces utilizar el fichero de configuración `web.xml`.

7. Abrimos el fichero `web.xml` del proyecto web (en *Configuration Files*) y añadimos una referencia al bean utilizando el editor de Netbeans. Para ello introducimos en el apartado *References* > *EJB References*, los siguientes campos:

- *EJB Reference Name*: OperacionBOBean
- *EJB Type*: Session
- *Interface Type*: Local
- *Local Interface*: es.ua.jtech.proyint.bean.OperacionBOLocal

Aviso:

Ahora hay dos clases factorías que devuelven implementaciones de objetos de negocio: `FactoriaBOs` en el paquete común y `FactoriaBOBeans` en el paquete EJB. La diferencia entre ambas está en el método `getOperacionBO()`: la `FactoriaBOs` devuelve un `OperacionBO` y la `FactoriaBOBeans` devuelve un EJB local `OperacionLocal`. Debemos modificar el proyecto web para que use la factoría que devuelve EJBs.

9.5. Modificación de la capa web

1. Por último, modificamos la capa web para que se llame a la nueva factoría. Cambia la clase `OpAnularReservaAccion` en el paquete `es.ua.jtech.proyint.struts.acciones.operacion` para que se obtenga el objeto de negocio utilizando la nueva factoría. Así lo que se obtiene realmente es el EJB local:

```
public class OpAnularReservaAccion extends Action {

   private static Log logger =
LogFactory.getLog(OpAnularReservaAccion.class
         .getName());

   @Override
   public ActionForward execute(ActionMapping mapping, ActionForm
form,
         HttpServletRequest request, HttpServletResponse response)
         throws Exception {

      ActionMessages ams = new ActionMessages();
      ActionForward forward = mapping.getInputForward();

      FactoriaBOBeans fd = FactoriaBOBeans.getInstance();
      IOperacionBO io = fd.getOperacionBOBean();

      ReservaForm rf = (ReservaForm) form;

     ...
```

En este momento, ya tenemos todo listo para probar el método de negocio en el EJB.

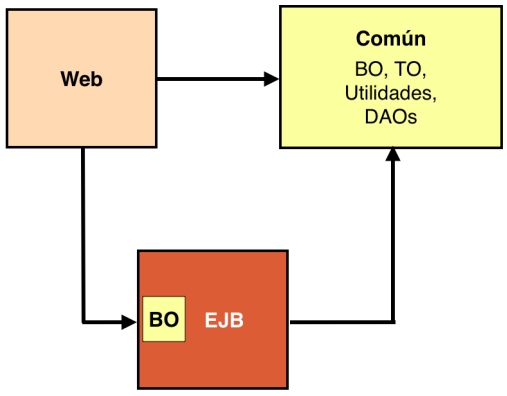
Lanza el EAR y prueba a crear una reserva y a anularla. Todo debe seguir funcionando correctamente. Modifica el método de negocio para que escriba un mensaje en la salida estándar ("El EJB está anulando una reserva") y comprueba que se ejecuta realmente.

Nota:

Resumen de la arquitectura de la capa de negocio EJB:
-La capa de negocio EJB está implementada por la clase `FactoriaBOBeans` y el EJB `OperacionBOBean`.
-Las operaciones de negocio en la clase `OperacionBOBean` deben llamadas a las operación de negocio con el mismo nombre definidas en el paquete común, envolviéndolas con todos los servicios de EJB. Por ejemplo, se ejecutan en un entorno transaccional. Si la operación devuelve una excepción provocada por algún fallo en algún DAO, el bean realiza un *rollback* de la transacción.

9.6. Un comentario sobre la arquitectura de la aplicación

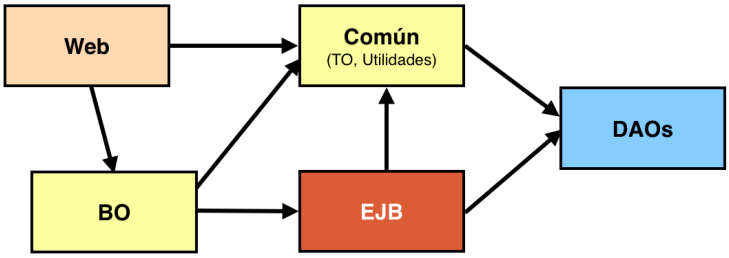
La arquitectura de la aplicación queda como aparece en la siguiente figura.



Vemos que la capa EJB contiene una implementación de objetos de negocio y la capa común contiene otra. El proyecto EJB también accede al proyecto común para usar los TO y los DAO.

Un defecto de esta arquitectura es que obliga a modificar la capa web (acciones de struts) según queramos utilizar un objeto de negocio EJB o un objeto de negocio procedente de la capa común. Esto no es correcto. Con una arquitectura bien diseñada los módulos ya implementados no debería tener que modificarse para incluir nuevas implementaciones (como la capa EJB).

Un estructura más correcta sería el que muestra la figura siguiente.



Aquí se ha separado el proyecto común en tres subproyectos. Un proyecto con los BO, otro con lo transfer object y algunas clases de utilidad y otro con los DAO. El proyecto EJB accede al nuevo proyecto común y a los DAO, pero no a los BO. La capa web se comunica directamente con los BO, que se pueden implementar como objetos del proyecto común o EJBs. La elección de una u otra implementación no afectará a la capa web.

En el proyecto de integración enterprise modificaremos todo el proyecto para conseguir esta estructura.

9.7. Cliente remoto

1. Implementa en el bean la operación remota `realizaReservaRemota(String idUsuario, String idLibro)` con el mismo código que la operación local.

2. Crea el proyecto Java "proy-int-cliente", configura su build path para que pueda usar de forma remota el EJB `OperacionBean` y escribe un cliente remoto que que realice una reserva:

```
   ...
   System.out.print("Introduce el id del libro: ");
   BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));
   String idLibro = in.readLine();
   System.out.print("Introduce el id del usuario: ");
   in = new BufferedReader(new InputStreamReader(System.in));
   String  idUsuario = in.readLine();
   System.out.println("Voy a realizar la reserva");
   operacion.realizaReservaRemta(idUsuario, idLibro);
   System.out.println("Préstamo realizado");
   ...
```

Comprueba en la interfaz web que el libro aparece en estado reservado.

9.8. Completamos todos los EJB y modificamos todas las llamadas de la capa web

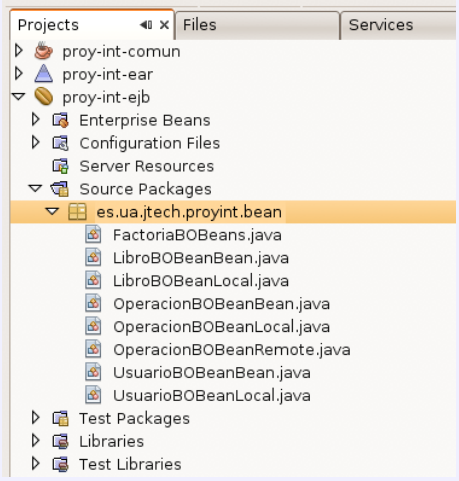
1. Implementamos todos los métodos restantes de `OperacionBOBean` y el resto de EJBs

`LibroBOBean` y `UsuarioBOBean`).

2. Modifica la factoría de BO del paquete EJB para que devuelva también los nuevos EJB.

3. Modifica todas las funciones de la capa web para que llamen a esta nueva factoría.

9.9. Entrega



Se deberán realizar las siguientes modificaciones:

- Crear las clases `OperacionJPAEEDAO`, `UsuarioJPAEEDAO` y `LibroJPAEEDAO`, y añadir en la factoría del DAO una método que devuelva instancias de estas clases.
- Crear el paquete EJB `proy-int-ejb` con los EJB `OperacionBOBean` (acceso local y remoto), `UsuarioBOBean` (acceso local) y `LibroBOBean` (acceso local). Implementa en todos las mismas funciones de negocio definidas en la capa común, pero ahora obteniendo el entity manager por inyección de dependencia en el EJB y pasándoselo a los nuevos DAO.
- Crea en el bean `OperacionBOBean` el método remoto `String realizaReservaRemota(String idUsuario, String idLibro)` y un programa Java de escritorio que acceda este método y realice la reserva de forma remota.

Deberás entregar un fichero tgz con todos los proctos (incluyendo el EAR, el proyecto común, el web y el EJB) después de haber hecho un *clean* para eliminar los ficheros JAR y los Class compilados y reducir su tamaño.

Table 1: Modificaciones a realizar

10. Integración con Mensajes: JMS

10.1. Introducción

El objetivo de esta sesión es integrar las tecnologías JMS/MDB dentro de nuestro proyecto de integración. Para ello, vamos a permitir que la aplicación envíe mensajes a un cola a través de un EJB, y que consuma mensajes mediante un MDB.

10.2. Multas Externas

Crearemos un MDB (`es.ua.jtech.proyint.mdb.MultasExternasMDB` dentro del proyecto `proy-int-ejb`)que escuche peticiones de multa de usuarios de sistemas externos. Así pues, la aplicación escuchará de una cola (`BibliotecaMultasUsuariosQueue`) los envíos de sistemas externos respecto a los usuarios que tienen multas, mediante un mensaje de texto que contiene el login y el número de días de multa, ambos separados mediante el carácter '#'.

Una vez recibido el mensaje, el MDB debe guardar la multa en la base de datos, y cambiar el estado del usuario de `activo` a `moroso`. En el caso de que el usuario ya fuese moroso, crearemos una nueva multa activa.

Multas solapadas

Se deja como optativo el tratamiento de multas solapadas. Si un usuario ya es moroso, y su multa activa acaba dentro de 5 días, y llega una nueva multa de 10 días, se debería crear una nueva multa de 15 días.

Todas las operaciones deben formar parte de una transacción distribuida, de modo que si salta alguna excepción en alguna operación, se deshagan todas los operaciones.

10.3. Deshaciendo las Reservas

En este ejercicio, vamos a crear un *Timer* para deshaga las reservas que han caducado. Una vez se invoque al método marcado con `@TimeOut`, a parte de borrar las reservas caducadas, vamos a enviar un mensaje a la cola (`BibliotecaReservasCaducadasQueue`) con el login y el isbn de la reserva que acaba de caducar.

Si al mismo tiempo caduca más de una reserva, se enviarán tantos mensajes como reservas caducadas.

Para invocar el *timer*, lo haremos desde el constructor de `FactoriaBOBeans`:

```
private FactoriaBOBeans() {
    // Creamos el InitialContext
```

```
    try {
        ic = new InitialContext();
    } catch (NamingException ex) {
        ex.printStackTrace();
        throw new RuntimeException();
    }

    // Llamamos al metodo para inicializar el timer
    try {
        OperacionBOBeanLocal bo = (OperacionBOBeanLocal)
ic.lookup("java:comp/env/OperacionBOBean");
        bo.initTimer();
    } catch (NamingException ex) {
        ex.printStackTrace();
        throw new RuntimeException();
    }
}
```

Para que el método `initTimer` sea accesible desde fuera del EJB pero que no puedan acceder a él los clientes no EJB, es necesario crearlo en el interfaz `OperacionBOBeanLocal`:

```
@Local
public interface OperacionBOBeanLocal extends IOperacionBO {
    public void initTimer();
}
```

Vamos a crear el método que caduca las reservas dentro del EJB OperacionBOBean. El esqueleto del método es el siguiente:

```
public void initTimer() {
        context.getTimerService().createTimer(1000, 1000*10,
"TimerCaducaReserva");
}

@Timeout
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void caducaReservas(Timer timer) {
        String nombre = (String) timer.getInfo();

        if ("TimerCaducaReserva".equals(nombre)) {
                // 1º comprobamos si hay reservas caducadas
                // 2º por cada reserva caducada,
                //    llamamos al método de anulaReserva y enviamos
mensaje a la cola
        }
}
```

Recordad que deberéis crear y cerrar la conexión JMS mediante los *callbacks* EJB, tal como vimos en la tercera sesión de teoría.

10.4. Probando...

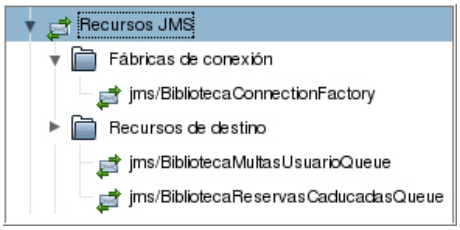
Para comprobar ambos ejercicios, vamos a crear dos aplicaciones cliente:

- `proy-int-multas-client`: cliente JMS que envía un mensaje de texto a

`BibliotecaMultasUsuariosQueue` con el login y el número de días de la multa separados por '#'.

- `proy-int-reservas-client`: cliente JMS que escucha en modo asíncrono un mensaje de texto de `BibliotecaReservasCaducadasQueue` con el login y el isbn del libro cuya reserva ha caducado, ambos separados por '#'.

Para que ambos ejercicios funcionen, tenéis que crear los recursos JMS necesarios:



Recursos JMS

10.5. A Entregar

Debéis entregar tanto el proyecto ejb como los dos proyectos cliente.

11. Spring

11.1. Introducción

En esta sesión de integración modificaremos el proyecto para usar Spring. Como modificar toda la aplicación sería impracticable en una sola sesión, cambiaremos únicamente la parte de administración de usuarios.

Vamos a desarrollar implementaciones en Spring de las tres capas:

- Una nueva implementación de `IUsuarioDAO` como un `@Repository`usando un *template* de Spring.
- Una nueva implementación de `IUsuarioBO` como un `@Service`.
- Una implementación de la capa web con Spring MVC en lugar de Struts.

Las capas de negocio y acceso a datos vamos a incluirlas en el proyecto común. La capa web la pondremos en un proyecto web aparte (`proy-int-web-spring`). Por el momento vamos a mantener los EJBs que ya existen y seguir desplegando la aplicación en Glassfish, ya que en la sesión de Servicios Web necesitamos los EJB para exportarlos como servicios.

11.2. Proyecto común: creación de DAO y BO con Spring

Debemos configurar primero el proyecto común para poder usar Spring:

1. Añadir la librería "Spring Framework 2.5" (NO Spring MVC, que recordemos que es para la capa web)
2. En la carpeta "resources" (no dentro de META-INF, sino directamente en esta carpeta), crear un fichero de configuración de beans de Spring llamado "beans.xml" (`New > Other > Other > Spring XML Configuration File`). En el último paso del asistente, marcar los espacios de nombres "context" y "jee".

11.2.1. Capa de acceso a datos

Debéis crear una nueva clase que implemente el interfaz `IUsuarioDAO`. Aunque se podría usar JPAA, dando lugar a una clase muy similar a `UsuarioJPAEEDAO`, vamos a emplear un *template* de Spring con código JDBC simplificado. Así tendréis más ejemplos de alternativas de implementación.

1. Junto con las otras implementaciones de DAOs para usuario, crear una clase llamada `UsuarioJDBCSpringDAO` que implemente el interfaz `IUsuarioDAO`. La clase será un `@Repository`
2. Recordad que para que Spring busque las anotaciones en el código fuente hacía falta poner en el "beans.xml" la etiqueta `context:component-scan`. Consultad las transparencias de la primera sesión si no recordáis su uso.

3. El DAO necesitará acceso al DataSource "jdbc/biblioteca" creado en glassfish, por lo que habrá que definirlo en el fichero "beans.xml" (consultad las últimas transparencias de la sesión 1. **CUIDADO:** en Glassfish es necesario poner el atributo "resource-ref" a false, porque los nombres JNDI son distintos a los de Tomcat.)
4. Aquí tenéis una plantilla con gran parte del código de la clase. Faltan por implementar los métodos "addUsuario" y "getUsuarios" (no iba a estar todo hecho :)). También falta por implementar "addMulta", pero en realidad ese no lo necesitamos para la aplicación de administración. Nótese que el DAO obtiene el DataSource gracias al `@Autowired`

11.2.2. Capa de negocio

En el paquete correspondiente, crear una nueva clase `UsuarioBOSpring` que implemente el interfaz `IUsuarioBO`:

1. Dicha clase será un `@Service`
2. Para el código podéis tomar como modelo la clase `UsuarioBO`. Será prácticamente igual, con la diferencia de que **la nueva implementación no debe usar factorías sino inyección de dependencias para instanciar el DAO que necesita.**

11.3. Configuración del proyecto web de Spring

Para no mezclar las librerías de Struts y Spring vamos a implementar la capa web de administración en un proyecto aparte (`proy-int-web-spring-gf`). Sin embargo queremos seguir teniendo un único punto de entrada a la aplicación, de modo que cuando se haga login como administrador saltemos a la nueva aplicación Spring MVC y en otro caso sigamos en la antigua con Struts. Esto lo podemos conseguir con una característica del servidor denominada *Single Sign On*.

11.3.1. Single Sign On

Con el *Single Sign On*, (en adelante SSO), tras autentificarnos en una aplicación estaremos automáticamente autentificados en todas las aplicaciones desplegadas en el mismo servidor. Esta característica está deshabilitada por defecto en GlassFish, por lo que el primer paso será activarla . Para ello:

1. En la consola de administración ir a "Configuración > Servicio HTTP > Servidores virtuales".
2. Veremos dos servidores: el de administracion (asadmin) y el que se usa para desplegar aplicaciones (server). Clicamos sobre este último para editar sus propiedades.
3. Debemos marcar la casilla "SSO" como "Activado" y pulsar sobre el botón "Guardar".

Nuestro siguiente objetivo es saltar a la aplicación web de Spring cuando se haga login como administrador. Si estudiamos el flujo de navegación codificado en el

"**struts-config.xml**" veremos que lo primero que se hace al entrar como administrador (en el forward llamado "indexAdmin" de la acción de login) es saltar a la URL "/listarUsuarios.do". Debemos cambiar esto:

1. Cambiamos el path del forward "indexAdmin" por "/jsp/admin/adminSpring.jsp"
2. Creamos la página "/jsp/admin/adminSpring.jsp" con el siguiente contenido:

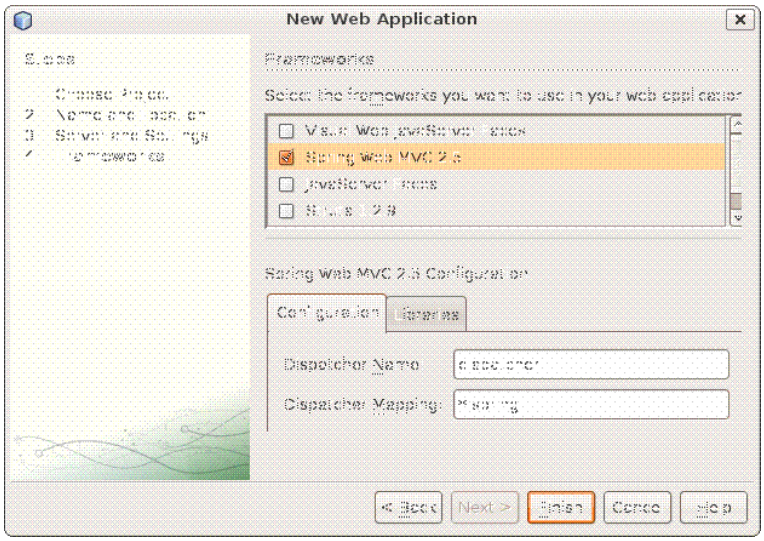
```
<%
response.sendRedirect("/proy-int-web-spring-gf/usuarioList.spring");
%>
```

Evidentemente si lo probamos ahora dará error de "no encontrado", pero al menos podemos hacerlo para asegurarnos de que salta a la URL adecuada.

Al hacer logout debemos saltar de nuevo a la aplicación principal, la de Struts. Por tanto el enlace al que apuntaba el "Cerrar sesión" cambiará. Esto ya se ha hecho en las plantillas de JSP que se os proporcionan más adelante.

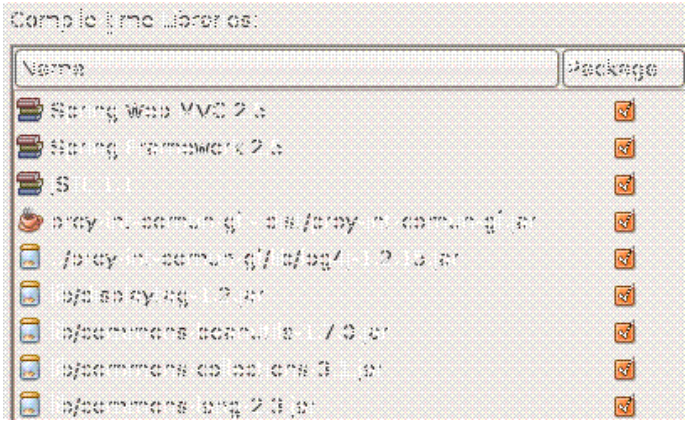
11.3.2. Creación del proyecto

Crearemos un proyecto de tipo "Web Application" llamado `proy-int-web-spring-gf`, para desplegar en Glassfish. En el paso 4 ("frameworks"), marcaremos "Spring Web MVC 2.5". En esta misma pantalla, en la configuración de "dispatcher mapping", mapearemos las URLs de tipo *.spring con el framework.



asistente de creación de proyectos web

Debemos incluir en el proyecto las librerías y JARs que se muestran en la figura:



librerías para el proyecto web de Spring

Como vemos, algunos de los JARs son dependencias del proyecto común. Los últimos 4 podéis bajároslos en esta plantilla , copiarlo en la carpeta lib del proyecto e incluirlo entre los JARs requeridos (algunos estaban ya en el proyecto web de Struts pero así evitamos dependencias directas).

11.3.3. Configuración de los descriptores de despliegue

El asistente de NetBeans ya habrá realizado automáticamente parte de la **configuración del web.xml**, pero hay cosas a añadir:

1. En la etiqueta "context-param" al comienzo del web.xml, aparece referenciado el archivo "applicationContext.xml". Este es un fichero que NetBeans crea automáticamente para que definamos en él los beans de las capas de negocio y datos. No obstante, ya hemos visto que estos beans estaban definidos en el proyecto común dentro de "beans.xml". Por tanto, **sustituiremos la referencia a `WEB-INF/applicationContext.xml` por `classpath:beans.xml`**. El prefijo "classpath:" hace que Spring busque este fichero en cualquier lugar del CLASSPATH. Podemos borrar el fichero "applicationContext.xml", no es necesario.
2. Es necesario proteger también la aplicación con seguridad declarativa, por lo que introduciremos en el web.xml:

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>todo</web-resource-name>
        <url-pattern>/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>administrador</role-name>
    </auth-constraint>
</security-constraint>
<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>RealmBiblioteca</realm-name>
</login-config>
```

La anterior configuración protege todas las URL dejando acceso solo a administradores. Usamos autentificación FORM aunque el formulario de login no está realmente en esta aplicación, pero es que hay que usar algún tipo de autentificación, y BASIC daría problemas al hacer logout.

Por otro lado, en el **descriptor sun-web.xml** debemos asociar el rol administrador con el grupo del mismo nombre, como vimos en la sesión correspondiente del módulo de servidores de aplicaciones.

```
    <security-role-mapping>
        <role-name>administrador</role-name>
        <group-name>administrador</group-name>
    </security-role-mapping>
```

Finalmente, en el **WEB-INF/dispatcher-servlet.xml** quitaremos toda la configuración que ha puesto NetBeans y pondremos esta:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
       http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
        http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">


    <bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver"
          p:prefix="/jsp/admin/"
          p:suffix=".jsp" />

    <bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
            <property name="basename" value="mensajes"/>
    </bean>

    <context:component-scan
base-package="es.ua.jtech.proyint.spring.web"/>
</beans>
```

11.4. La capa web

Vamos a implementar un interfaz alternativo al que ya tenemos con Struts, aunque lo

haremos con el mismo aspecto. Así tendréis dos implementaciones distintas y podréis juzgar mejor los puntos débiles y fuertes de cada framework.

Cuestiones generales:

- Donde en Struts teníamos acciones, aquí tendremos Controllers
- Las taglibs de Struts se sustituyen por las de Spring. Algunas no tienen equivalente y se sustituyen por tags de JSTL.
- En Spring no existen los ActionMessages ni los ActionErrors (al menos, no como en Struts), por lo que la generación de mensajes desde el controller la haremos "a mano", sin un API propio del Spring.

11.4.1. Configuración de displaytags y mensajes al usuario

Copiar el fichero **displaytag.properties** del proyecto de Struts al paquete por defecto de este proyecto. En la primera línea cambiad el "I18nStrutsAdapter" por "I18nSpringAdapter". Así los mensajes de displaytag aparecerán en el locale actual.

Copiad el **mensajes.properties** del proyecto de Struts al nuevo proyecto web. Lo más indicado es hacerlo en una carpeta fuente "resources" igual que está en el proyecto Struts, pero también podéis copiarlo al paquete por defecto.

Cuidado:

Esto no quiere decir que la aplicación esté internacionalizada. Como el idioma se escoge en la parte de Struts faltaría "pasarle" a Spring cuál es el locale elegido. Esto se podría hacer como un parámetro HTTP o bien en una cookie. No obstante no vamos a preocuparnos del tema dado el tiempo disponible en la sesión.

11.4.2. Implementación del caso de uso "listarUsuarios"

Lo primero es **implementar un interfaz `Tokens`** para guardar constantes equivalente al que teníamos con Struts. Creadlo en el paquete "es.ua.jtech.proyint.spring.web". Puede serviros este código completo

En segundo lugar, **implementar el controller** como la clase `UsuarioListController` en el paquete "es.ua.jtech.proyint.spring.web.controllers". El listado se ofrece a continuación, para que dispongáis de un ejemplo completo.

```
package es.ua.jtech.proyint.spring.web.controllers;

import es.ua.jtech.proyint.bo.usuario.IUsuarioBO;
import es.ua.jtech.proyint.bo.usuario.UsuarioException;
import es.ua.jtech.proyint.entity.UsuarioEntity;
import es.ua.jtech.proyint.spring.web.Tokens;
import java.util.ArrayList;
import java.util.List;
```

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("/usuarioList.spring")
public class UsuarioListController {

    private static Log logger =
LogFactory.getLog(UsuarioListController.class.getName());
    @Autowired
    IUsuarioBO iu;

    @RequestMapping(method = {RequestMethod.GET,
RequestMethod.POST})
    public String execute(ModelMap model) {
        List<String> mensajes = new ArrayList<String>();

        List<UsuarioEntity> lista = null;
        try {
            lista = iu.listaUsuarios();
            if (lista != null) {
                model.addAttribute(Tokens.RES_USUARIOS, lista);
                logger.info("Listado servido correctamente");
            } else {
                logger.warn("No se encontraron resultados");
                mensajes.add(Tokens.MSJ_USUARIO_LIST_ERROR);
            }
        } catch (UsuarioException ex) {
            mensajes.add(Tokens.MSJ_USUARIO_LIST_ERROR);
            logger.error("Error recuperando listado de usuarios",
ex);
        }

        if (mensajes.size() > 0) {
            model.addAttribute(Tokens.RES_MENSAJES, mensajes);
        }
        return Tokens.VISTA_LISTA_USUARIOS;
    }
}
```

puntos a destacar:

- **No hay llamadas a factorías**. Si el controller necesita un IUsuarioBO lo obtiene de Spring.
- En lugar de ActionMessages o ActionErrors se usa una lista de Strings. Luego se mostrará en el JSP con una etiqueta especial de spring.

Aclaración:

Puede que os preguntéis por qué el método "execute" se asocia no solo con GET sino también con POST (o puede que no os lo preguntéis, ciertamente). En cualquier caso, el POST se da solo

cuando, tras haber añadido o modificado un usuario (que se habrá hecho con POST) se hace un forward al listado de usuarios, que entonces también se estará haciendo con POST.

Junto con la implementación del controller, la parte más costosa de la implementación es cambiar los JSP para eliminar las tags de Struts y sustituirlas por Spring y JSTL. Para facilitaros la tarea tenéis estas plantillas donde se incluyen ya las páginas de listado y actualización de usuarios, además de menú, cabecera y pie. También tenéis el CSS y alguna imagen adicional.

11.4.3. Implementar el resto de casos de uso (modificación, alta y baja)

De los casos de uso que quedan, el más complejo es el de modificar usuario. Siguiendo el paralelismo con Struts se pueden hacer dos controller: uno para buscar el usuario a partir del login y mostrarlo en el formulario (UsuarioSelController) y otro para guardar los datos editados por el usuario (UsuarioUpdController). A modo de "pista" se os da el código del método "principal" del UsuarioSelController (falta la definición de la clase en la que meteríais este método):

```
   public String preparaForm(ModelMap modelo,
@RequestParam("login") String login) {
        List<String> mensajes;

        String vista = Tokens.VISTA_USUARIO_SEL_ERROR;

        mensajes = new ArrayList<String>();
        try {
            UsuarioEntity usuario = bo.recuperaUsuario(login);
            if (usuario != null) {
                logger.info("[" + login + "] - Datos servidos
correctamente");
                UsuarioSpringBean usb = new UsuarioSpringBean();
                BeanUtils.copyProperties(usb, usuario);
                usb.setPassword2(usuario.getPassword());
                modelo.addAttribute("usuario", usb);
                vista = Tokens.VISTA_USUARIO_SEL_OK;
            } else {
                logger.error("Error obteniendo usuario " + login);
                mensajes.add(Tokens.MSJ_USUARIO_GET_NOT_FOUND);
            }
        } catch (Exception ex) {
            mensajes.add(Tokens.MSJ_USUARIO_GET_ERROR);
            logger.error("Error recuperando usuario " + login, ex);
        }

        modelo.addAttribute(Tokens.RES_MENSAJES, mensajes);
        return vista;
    }
```

Nótese que **el código anterior referencia una clase que debéis crear vosotros, `UsuarioSpringBean`** en el paquete "es.ua.jtech.proyint.spring.web". Este es el objeto desde/hacia el que van los datos del formulario,lo que en la sesión de MVC llamábamos

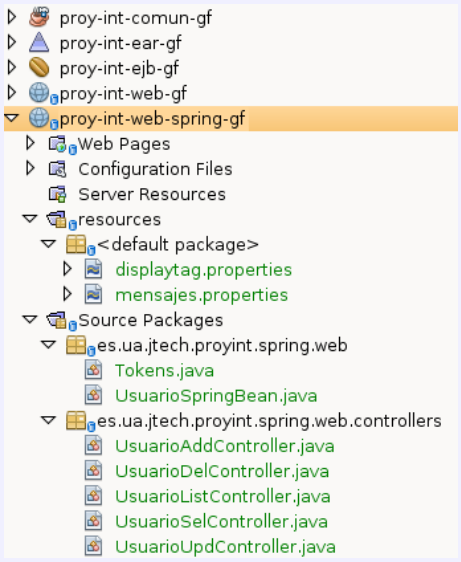
el "command". En principio podríamos usar UsuarioEntity para esto, pero tenemos el problema de que esta clase carece del "password2" que el formulario tiene. Fijaos en que lo más sencillo es hacer que `UsuarioSpringBean` herede de UsuarioEntity, y con la propiedad adicional de "password2". Por desgracia esto nos obliga a seguir usando BeanUtils para copiar datos entre UsuarioEntity y UsuarioSpringBean.

Recordad que **la página "modificaUsuario.jsp" ya está hecha** en las plantillas.

Para terminar este caso de uso faltaría definir el controller `UsuarioUpdController`. Nótese que este controller no retorna directamente a una vista sino a otro Controller (UsuarioListController), porque tras modificar el usuario queremos ver de nuevo la lista. Esto en Spring se consigue devolviendo una URL con el prefijo "forward:". En este caso sería "forward:usuarioList.spring". Así el viewresolver no toma esto como un nombre lógico de vista sino como una URL. Fijaos en que en Tokens ya está definido esto en la constante `VISTA_USUARIO_UPD_OK`

Finalmente quedarían el borrado de usuario y el alta, que va a ser muy similar a la modificación. Para el alta, además del controller tendréis que crear `altaUsuario.jsp`.

11.5. Entrega



entrega

No se muestran las páginas jsp. Recordad que salvo altaUsuario.jsp el resto ya está en las plantillas disponibles.

Deberás entregar un fichero tgz con todos los proyectos (incluyendo los que ya tenías de antes salvo el cliente java. Recordad que la aplicación Spring no implementa el login y por tanto el resto de proyectos siguen siendo necesarios. Empaquetadlos después de haber hecho un *clean* para eliminar los ficheros JAR y los Class compilados y reducir su tamaño.

Table 1: Entrega

12. Servicios Web

12.1. Introducción

En esta sesión de integración vamos a ofrecer como servicio web algunas de las operaciones implementadas mediante EJBs en sesiones anteriores. También utilizaremos servicios web externos para obtener información extendida sobre los libros de nuestra biblioteca. Por último, crearemos un cliente Java para uno de nuestros servicios.

Nota

Para realizar el trabajo de esta sesión partiremos del resultado de la sesión anterior (Spring).

12.2. Creación de servicios web

Lo primero que haremos será crear un servicio web para obtener información de los libros. Crearemos un nuevo módulo web de nombre `proy-int-ws-gf` que contendrá los servicios web que ofrece nuestra aplicación. Dentro de este módulo crearemos un servicio web `LibroWS` en el paquete `es.ua.jtech.proyint.ws.libro` a partir del EJB `LibroBOBean`. Concretamente, las operaciones que deberá ofrecer este servicio son:

- `LibroTO recuperaLibro(String isbn)`
- `List<LibroTO> listaLibros()`

Nota

Para que estas operaciones se añadan automáticamente al generar el servicio web a partir del EJB, deberán figurar en la interfaz local del EJB (`LibroBOBeanLocal`). Dado que, debido a la forma de crear los EJBs en sesiones anteriores, es posible que dicha interfaz esté vacía, tendremos dos opciones alternativas: (1) añadir los métodos que queramos exportar como servicios a la interfaz local del EJB, o (2) crear el servicio web sin operaciones y añadirlas a mano posteriormente.

Hemos de destacar que en el servicio web estamos devolviendo los datos como objetos *Transfer Object* en lugar de entidades JPA. Hemos optado por esta solución debido a que es conveniente tener control sobre las estructuras de datos que se serializan durante la llamada a los servicios, para evitar enviar información no necesaria o con un formato que no resulte adecuado para la conversión a XML. Con las entidades JPA encontramos problemas con los campos que relacionan unas entidades con otras. Por ejemplo, si devolvemos un `LibroEntity`, el servicio web generado intentará devolver también las operaciones asociadas, lo cual causará un problema al intentar recuperar esta información de forma *lazy*.

Incluso sería conveniente generar previamente el WSDL, y a partir de él los tipos de datos necesarios y el esqueleto del servicio web. Sin embargo, debido a la mayor complejidad de esta forma de trabajar, optaremos por la solución intermedia de crear una serie de objetos *Transfer Object* que encapsulen los datos con los que trabaje nuestro servicio. Concretamente necesitaremos una clase `LibroTO`, en la que volcaremos los datos de nuestros objetos `LibroEntity`. Esta clase se ubicará en el paquete `es.ua.jtech.proyint.to` del proyecto `proy-int-comum-gf`, y tendrá la siguiente

información:

```
public class LibroTO implements Serializable {

  private String isbn;
  private String titulo;
  private String autor;
  private int numPaginas;
  private Date fechaAlta;

  public LibroTO() {
  }

  public LibroTO(String isbn, String titulo, String autor,
                 int numPaginas, Date fechaAlta) {
    this.isbn = isbn;
    this.titulo = titulo;
    this.autor = autor;
    this.numPaginas = numPaginas;
    this.fechaAlta = fechaAlta;
  }

  public LibroTO(LibroEntity libro) {
    this.isbn = libro.getIsbn();
    this.titulo = libro.getTitulo();
    this.autor = libro.getAutor();
    this.numPaginas = libro.getNumPaginas();
    this.fechaAlta = libro.getFechaAlta();
  }

  // Getters y setters
  ...
}
```

Podemos ver que esta clase tiene un constructor a partir de un `LibroEntity`, que nos permitirá volcar de forma sencilla los datos de la entidad JPA a nuestro *Transfer Object*.

En segundo lugar, de la misma forma que antes, crearemos un servicio `OperacionWS` en el paquete `es.ua.jtech.proyint.ws.operacion`, esta vez a partir del EJB `OperacionBOBean`. En este caso deberá exponer la siguiente operación:

- `String realizaReserva(String idUsuario, String idLibro)`

Hay que tener en cuenta que si ofreciésemos esta operación directamente en nuestro servicio web podríamos estar creando un agujero en la seguridad de la aplicación, ya que no estamos autentificando al usuario que realiza la reserva. La forma más sencilla de añadir este tipo de seguridad es proporcionar dos parámetros extra a la operación del servicio, con el *login* y el *password* del usuario que quiere realizar la operación, y dentro del código del servicio implementar seguridad programada para comprobar si dicho usuario tiene los permisos necesarios.

Por lo tanto, la operación en el servicio debería quedar como se muestra a continuación:

- `String realizaReserva(String idUsuario, String idLibro, String login, String password)`

También debemos comentar que estamos implementando una seguridad muy básica, con la que no ofrecemos confidencialidad al transmitir la información, por lo que si alguien intercepta el mensaje SOAP de petición podrá ver tanto nuestro *login* como nuestro *password*. Si necesitásemos un nivel mayor de seguridad, podríamos cifrar los datos bien a nivel de transporte mediante SSL o a nivel de mensaje con WS-Security, aunque para nuestra aplicación optaremos por mantener simplemente la autentificación básica.

12.3. Acceso a servicios web externos

Vamos a añadir a la aplicación la funcionalidad de obtener los detalles de un libro (portada, precio, editorial, fecha de publicación, etc). Para hacer esto recurriremos a los servicios web de Amazon. Seguiremos los siguientes pasos:

1. Añadimos al paquete `es.ua.jtech.proyint.to` del proyecto común (`proy-int-comun-gf`) un nuevo *Transfer Object* que extienda `LibroTO` para añadir atributos adicionales a los libros:

```
public class LibroAmazonTO extends LibroTO {

  float precio;
  String imagen;
  int imagenAncho;
  int imagenAlto;
  String editorial;
  String fechaPublicacion;
  String urlDetalles;

  // Getters y setters
  ...

}
```

2. Creamos dentro del módulo EJB un cliente para acceder al servicio web de Amazon, cuyo documento WSDL se puede encontrar en la siguiente dirección:

```
webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl
```

3. Añadimos al EJB `LibroBOBean` un nuevo método `recuperaLibroAmazon`, que tomará como parámetro el ISBN del libro que buscamos, y nos devolverá un objeto `LibroAmazonTO` con los datos de dicho libro, accediendo para ello al servicio web de Amazon. Podemos utilizar el siguiente código para obtener los datos de un libro a través de los servicios web de Amazon:

Ayuda

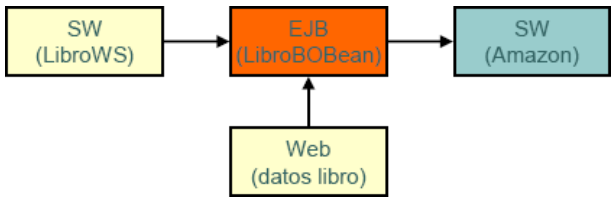
Primero podemos crear el método en el EJB e insertar dentro de él la llamada a la operación `itemLookup` de Amazon. Posteriormente modificamos el código generado automáticamente para dejarlo como se muestra a continuación.

```
public LibroAmazonTO recuperaLibroAmazon(String isbn)
                                              throws LibroException {
```

```
try {
  String marketplaceDomain = "";
  String awsAccessKeyId = "15JCR9XWMPTGV91JC1R2";
  String subscriptionId = "";
  String associateTag = "";
  String validate = "";
  String xmlEscaping = "";
  ItemLookupRequest shared = new ItemLookupRequest();
  List<ItemLookupRequest> request = null;
  Holder<OperationRequest> operationRequest =
                          new Holder<OperationRequest>();
  Holder<List<Items>> items =
                          new Holder<List<Items>>();

  shared.setSearchIndex("Books");
  shared.setIdType("ISBN");
  shared.getItemId().add(isbn);
  shared.getResponseGroup().add("Offers");
  shared.getResponseGroup().add("Medium");

  AWSECommerceServicePortType port =
                           service.getAWSECommerceServicePort();
  port.itemLookup(marketplaceDomain, awsAccessKeyId,
         subscriptionId, associateTag, validate, xmlEscaping,
         shared, request, operationRequest, items);

  List<LibroAmazonTO> libros = new ArrayList<LibroAmazonTO>();

  List<Items> listaResultados = items.value;
  if (listaResultados == null) {
    return null;
  }
  for (Items resultado : listaResultados) {
    List<Item> listaArticulos = resultado.getItem();
    if (listaArticulos == null) {
      return null;
    }
    for (Item articulo : listaArticulos) {
      ItemAttributes atributos = articulo.getItemAttributes();
      LibroAmazonTO libro = new LibroAmazonTO();
      libro.setIsbn(isbn);
      libro.setTitulo(atributos.getTitle());
      libro.setNumPaginas(
                      atributos.getNumberOfPages().intValue());

      Image imagen = articulo.getMediumImage();
      libro.setImagen(imagen.getURL());
      libro.setImagenAlto(
                      imagen.getHeight().getValue().intValue());
      libro.setImagenAncho(
                      imagen.getWidth().getValue().intValue());

      List<String> autores = atributos.getAuthor();
      String listaAutores = "";
      for (String autor : autores) {
        listaAutores = autor + "; ";
      }
      libro.setAutor(listaAutores);
```

```
        libro.setPrecio(atributos.getListPrice() != null ?
            atributos.getListPrice().getAmount().floatValue() / 100
            : -1);

        libro.setEditorial(atributos.getPublisher());
        libro.setUrlDetalles(articulo.getDetailPageURL());
        libro.setFechaPublicacion(atributos.getPublicationDate());

        libros.add(libro);
      }
    }

    if(libros.size()>0) {
      return libros.get(0);
    } else {
      return null;
    }

  } catch (Exception ex) {
    throw new LibroException(
            "Error recuperando detalles del libro de Amazon", ex);
  }
}
```

4. Exponemos esta operación en el servicio web anteriormente creado (`LibroWS`). Con esto podremos probar que se obtienen correctamente los datos de Amazon.



Relaciones entre componentes

5. En el proyecto web (`proy-int-web-gf`) crearemos un JSP llamado `jsp/biblio/libroAmazon.jsp` que muestre los datos detallados de un libro que han sido obtenidos mediante el servicio web de Amazon. El código fuente de este JSP será el siguiente:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://struts.apache.org/tags-logic"
    prefix="logic" %>
<%@ taglib uri="http://struts.apache.org/tags-html"
    prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-bean"
    prefix="bean" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">

<%@page import="es.ua.jtech.proyint.entity.LibroEntity"%>
<%@page import="es.ua.jtech.proyint.entity.TipoOperacion"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
```

```
<title><bean:message key="libroAmazon.titulo"/></title>
<link rel="stylesheet" type="text/css" href="css/style.css"
        title="800px style" media="screen,projection" /></head>
<body>

<div id="wrap">

<%@include file="/jsp/cabecera.jspf" %>
<%@include file="/jsp/biblio/menu.jspf" %>

<div id="content">

<h2><bean:message key="libroAmazon.titulo"/></h2>

<logic:messagesPresent message="true">
        <div class="box">
        <html:messages message="true" id="msg">
        <c:out value="${msg}" /><br/></html:messages>
        </div>
</logic:messagesPresent>

<logic:messagesPresent>
        <div class="box"><html:errors /></div>
</logic:messagesPresent>

<c:choose>
<c:when test="${empty requestScope.libro}">
        <p><bean:message key="libroAmazon.noencontrado"/></p>
</c:when>
<c:otherwise>
    <table>
        <tr>
            <td valign="top">

    <img src="${libro.imagen}" width="${libro.imagenAncho}"
         height="${libro.imagenAlto}"/>

            </td>
            <td>

    <table>
        <tr>
        <td><strong><bean:message key="libro.ISBN"/></strong></td>
    <td>${libro.isbn}</td>
    </tr>
        <tr>
        <td><strong><bean:message key="libro.titulo"/>
        </strong></td>
        <td>${libro.titulo}</td>
    </tr>
        <tr>
        <td><strong><bean:message key="libro.autor"/></strong></td>
        <td>${libro.autor}</td>
    </tr>
        <tr>
        <td><strong><bean:message key="libroAmazon.editorial"/>
        </strong></td>
        <td>${libro.editorial}</td>
    </tr>
```

```
        <tr>
        <td><strong><bean:message key="libroAmazon.numPaginas"/>
        </strong></td>
        <td>${libro.numPaginas}</td>
    </tr>
        <tr>
        <td><strong><bean:message
        key="libroAmazon.fechaPublicacion"/></strong></td>
        <td>${libro.fechaPublicacion}</td>
    </tr>
        <tr>
        <td><strong><bean:message key="libroAmazon.precio"/>
        </strong></td>
        <td>$${libro.precio} (Amazon)</td>
    </tr>
        <tr>
        <td></td>
        <td><strong><a href="${libro.urlDetalles}"
        target="_blank"><bean:message
        key="libroAmazon.url"/></a></strong></td>
    </tr>
    </table>

            </td>
        </tr>
    </table>

</c:otherwise>
</c:choose>

</div>

<%@include file="/jsp/pie.jspf" %>

</div>

</body>
</html>
```

6. En este fichero se utilizan una serie de propiedades `libroAmazon.*` que deberíamos añadir al fichero `mensajes.properties` del directorio `resources` del proyecto web:

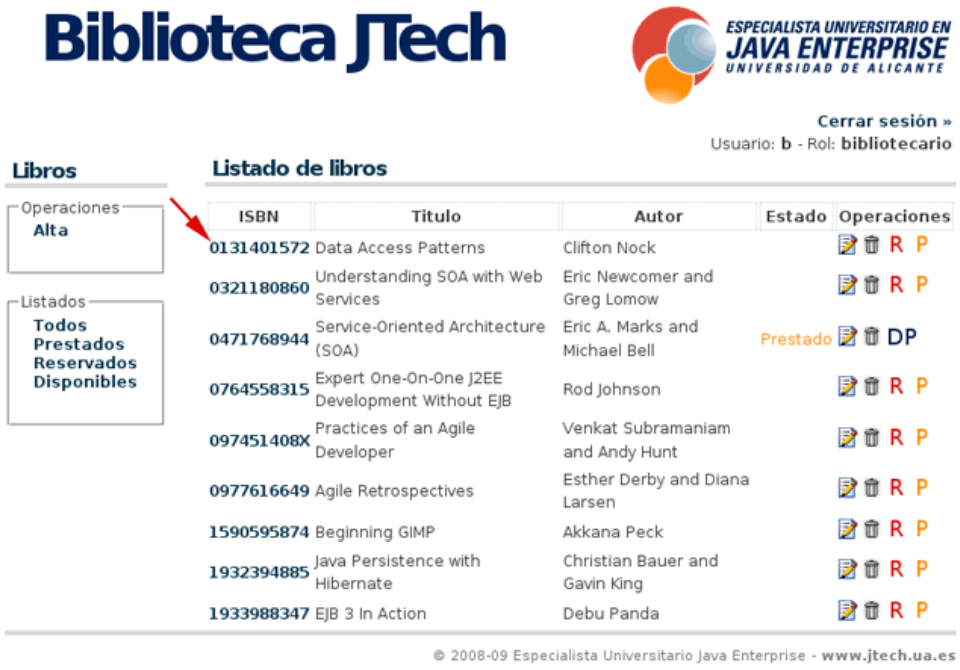
```
#libroAmazon
libroAmazon.titulo= Datos del libro
libroAmazon.noencontrado = No se ha encontrado el libro solicitado
libroAmazon.editorial= Editorial
libroAmazon.numPaginas= N\u00FAmero de p\u00E1ginas
libroAmazon.fechaPublicacion= Fecha de publicaci\u00F3n
libroAmazon.precio= Precio
libroAmazon.url= M\u00E1s informaci\u00F3n
```

7. Deberemos también crear la acción de *struts* necesaria para acceder a dicha página (`LibroAmazonAccion`) y configurar dicha acción en el fichero `struts-config.xml`. La acción podrá ser como se muestra a continuación:

```
public ActionForward execute(ActionMapping mapping,
        ActionForm form, HttpServletRequest request,
        HttpServletResponse response)
                                        throws Exception {
```

```
  try
  {
      String isbn = request.getParameter("isbn");

      if(isbn==null) {
        throw new BibliotecaException(
                    "Se debe especificar un ISBN.");
      }

      FactoriaBOBeans fd = FactoriaBOBeans.getInstance();
      LibroBOBeanLocal il = (LibroBOBeanLocal)fd.getLibroBOBean();
      LibroAmazonTO libro = il.recuperaLibroAmazon(isbn);

      request.setAttribute(Tokens.RES_LIBRO, libro);
      return mapping.findForward(Tokens.FOR_OK);

  } catch (BibliotecaException ex) {
    ActionMessages am = new ActionMessages();
    am.add(ActionMessages.GLOBAL_MESSAGE,
           new ActionMessage(Tokens.MSJ_LIBRO_GET_ERROR));
    saveErrors(request, am);
    logger.error("Error obteniendo datos del libro de Amazon. ",
                 ex);
    return mapping.findForward(Tokens.FOR_ERROR);
  }
}
```

8. En la página en la que se muestra el listado de todos los libros (`jsp/biblio/listadoLibros.jsp`) haremos que el ISBN sea un enlace a los datos detallados del correspondiente libro.



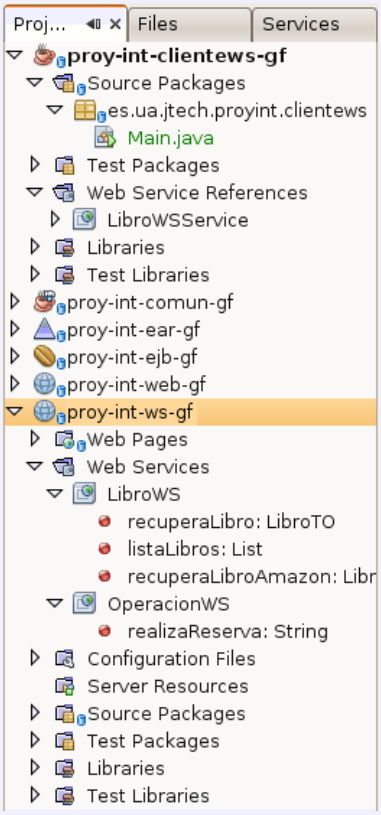
Listado de libros con enlaces

12.4. Cliente para los servicios web

Crear un cliente Java con Netbeans para nuestro servicio web de la biblioteca, en un nuevo proyecto al que llamaremos `proy-int-clientews-gf`. El programa deberá invocar la operación `listaLibros` y mostrar en la consola el listado de libros.

De cada libro se mostrará su ISBN, su título, y su autor.

12.5. Resumen



Estructura de Proyecto

Se deberá añadir un nuevo módulo `proy-int-ws-gf` con los siguientes servicios web:

- `LibroWS`: Servicio web que ofrecerá las operaciones `recuperaLibro`, `listaLibros`, y `recuperaLibroAmazon`.
- `OperacionWS`: Servicio web que ofrecerá la operación `realizaReserva`.

Se deberán añadir los siguientes componentes al módulo `proy-int-comun-gf`:

- `LibroTO`: *Transfer Object* con los datos de un libro.
- `LibroAmazonTO`: *Transfer Object* con los datos detallados de un libro proporcionados por Amazon.

Se deberá añadir al módulo `proy-int-ejb-gf`:

- `Cliente de servicios Amazon`: *Stub* para acceder a los servicios web de Amazon.
- Operación `recuperaLibroAmazon` en `LibroBOBean`: Obtiene los datos detallados de un libro mediante los servicios de Amazon.

Se deberán añadir los siguientes componentes al módulo `proy-int-web-gf`:

- `jsp/biblio/libroAmazon.jsp`: Página JSP que muestra los detalles de un libro dado su ISBN.
- `LibroAmazonAccion`: Acción de *struts* para mostrar los detalles de un libro (dado su ISBN).
- Configuración de la acción de *struts* en `struts-config.xml`, y propiedades con los textos de la página de información de Amazon en `mensajes.properties`.
- Enlaces a la página con los datos de Amazon en los ISBN del listado de todos los libros (`listadoLibros.jsp`).

Se deberá añadir un nuevo proyecto `proy-int-clientews-gf` con un cliente Java que acceda a nuestro servicio. Obtendrá la lista de todos los libros, y la mostrará por la consola.

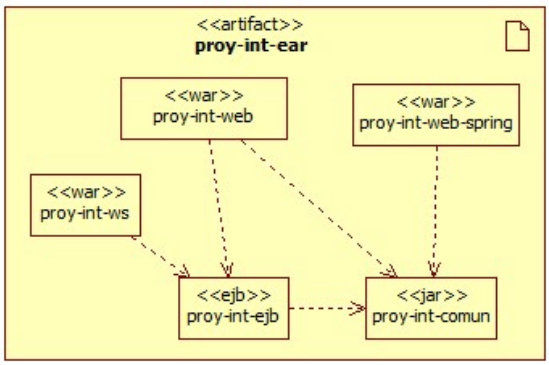
13. Proyecto Enterprise

13.1. Introducción

El objetivo del proyecto enterprise es poner en práctica la gran mayoría de las tecnologías estudiadas durante el curso, y servir como punto final donde se realicen una serie de prácticas a la finalización del proyecto.

13.1.1. ¿Qué Tenemos?

Antes de empezar esta sesión, partimos de una aplicación empaquetada dentro de un EAR (`proy-int-ear.ear`), el cual contiene una aplicación web para la administración y el bibliotecario (`proy-int-web.war`), otra aplicación web mediante *Spring* que duplica la administración (`proy-int-web-spring.war`), y una tercera para acceso a los *Web Services* que ofrecemos (`proy-int-ws.war`). Aparte, tenemos un proyecto EJB (`proy-int-ejb.jar`) y una librería de utilidades (`proy-int-comun.jar`).

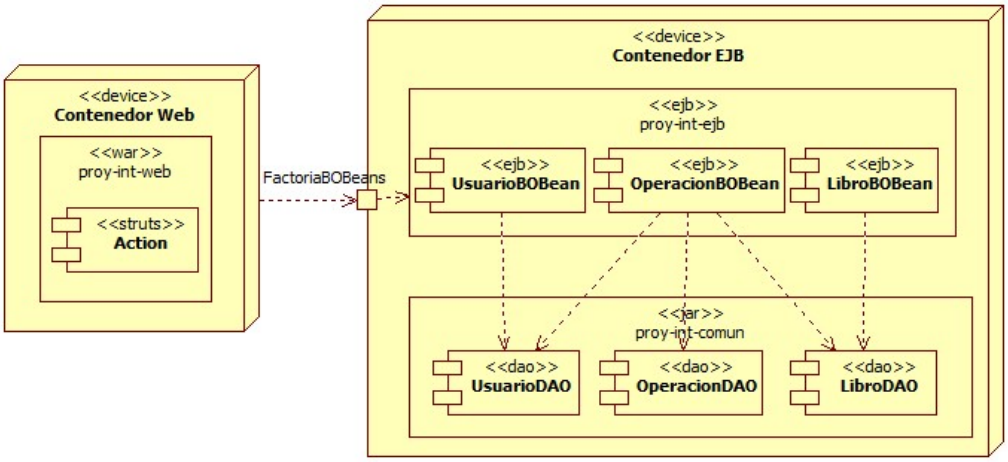


Esquema EAR

Clientes...

Los diferentes clientes EJBs y de Web Services los vamos a ignorar para simplificar los diagramas.

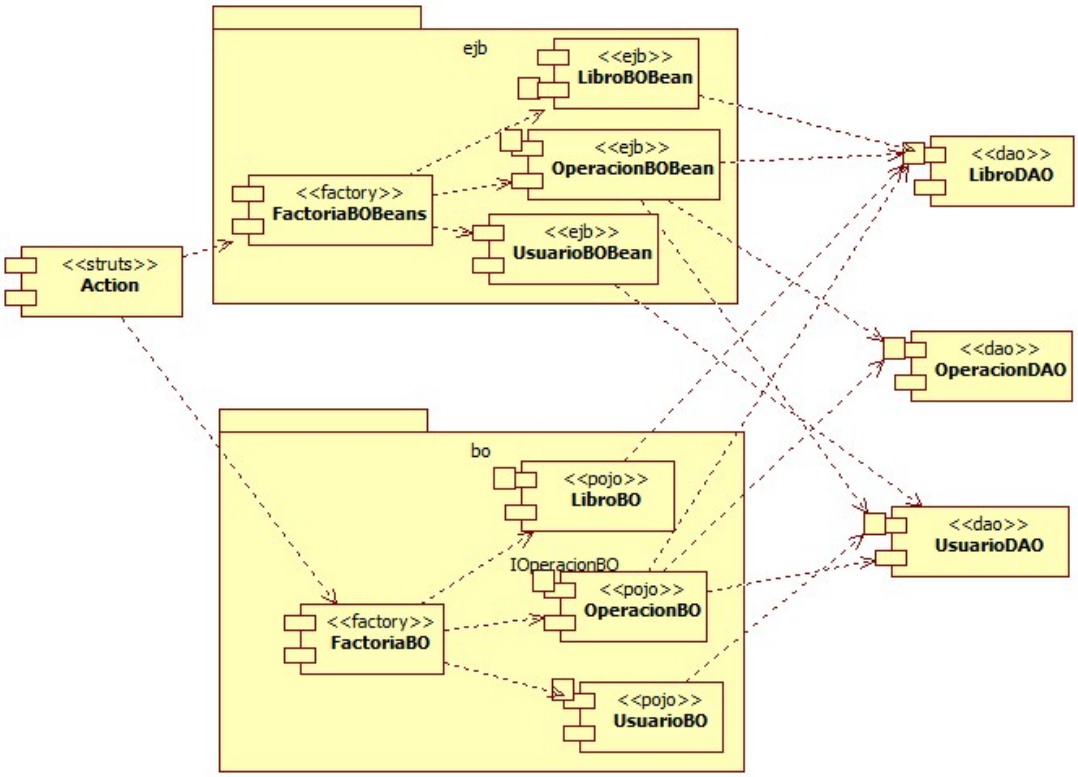
A nivel lógico, la aplicación está dividida en 3 capas: **presentación, negocio y datos**. En el módulo de EJB, la parte de negocio se refactorizó a un EJB para aprovechar la gestión de transacciones y su escalabilidad. De este modo, el EJB de Operación coordina las transacciones a nivel de negocio, en vez de realizarse a nivel de datos mediante JDBC. Del mismo modo, por ejemplo, respecto al subsistema de Libro, también hemos creado un EJB para integrar la aplicación con Amazon via *Web Services*.



Arquitectura de Despligue

Todas estas divisiones ha provocado que tengamos duplicada la lógica de negocio tanto en los EJBs como en nuestros BOs, y que también hayamos duplicado la `FactoriaBOs` (para los EJBs `FactoriaBOBeanss`) lo que nos ha provocado tener que modificar todos los `Actions` que hacen uso de los EJBs.

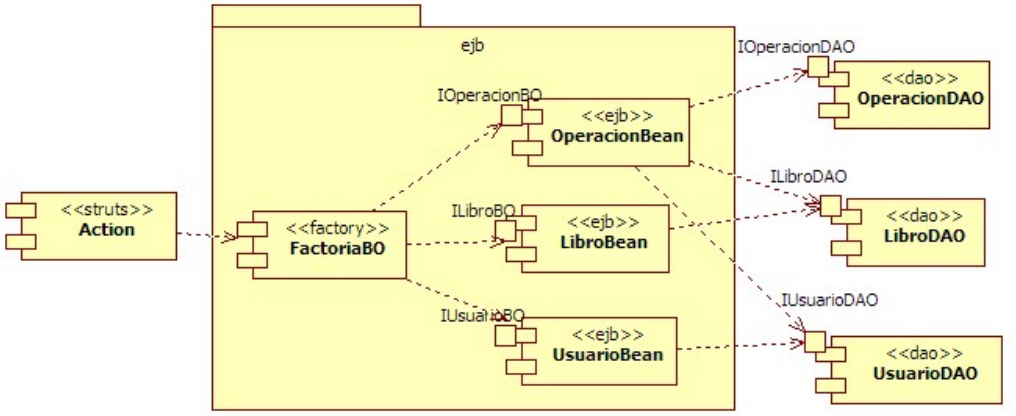
Como podemos observar en el siguiente gráfico, todo el negocio se ha refactorizado a EJBs, con su consiguiente duplicidad de lógica y dispersión del negocio. La decisión de tener en la aplicación partes implementadas mediante EJB y otras sin ellos es más que discutible, ya que provoca inconsistencia en el código, así como duplicidad de código (tenemos 2 factorías para BOs).



Arquitectura EJB y BO

13.1.2. Donde Queremos Llegar

Cómo queremos una aplicación homogénea, y por cuestiones del guión hemos decidido utilizar EJBs, vamos a acceder a la lógica de negocio mediante EJBs de sesión sin estado. La lógica de negocio está definida en las distintas interfaces IOperacionBO, ILibroBO y IUsuarioBO. Estas interfaces están implementadas por los EJBs y por los BOs.



Arquitectura con EJB

Para llegar a este diseño, necesitamos reorganizar nuestra separación lógica mediante los proyectos de Netbeans. Los pasos que vamos a seguir son:

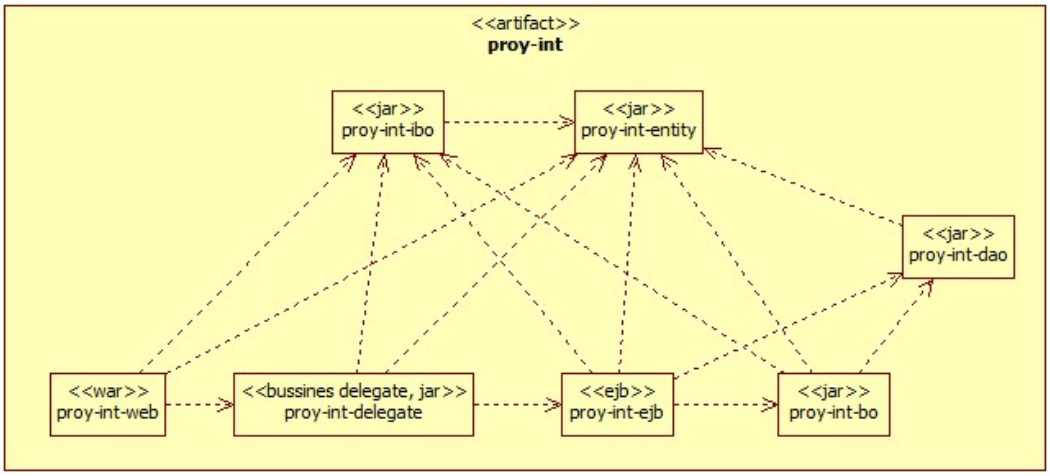
1. Dividir el proyecto común en tantos proyectos como capas arquitectónicas contenga. Es decir, vamos a crear:
   - un proyecto para guardar los DAOs (`proy-int-dao`)
   - otro para guardar las entidades (`proy-int-entity`), que además contendrá la excepción padre de la aplicación (`BibliotecaException`)
   - vamos a separar a su vez el negocio en 2 proyectos:
     - uno para guardar las interfaces de negocio y las excepciones (`proy-int-ibo`)
     - y otro para las implementaciones que ya tenemos (`proy-int-bo`).

Refactorización proyecto comun

2. Como el proyecto web utiliza los interfaces de negocio y la factoría de BOs, estos elementos deben colocarse en un proyecto independiente de la implementación que luego hagamos. Lo ideal es crear un proyecto (`proy-int-delegate`) que haga la función del patrón *Business Delegate* y que oculte la implementación del negocio. Este proyecto contendrá únicamente la factoría de negocio (`FactoriaBOs`), y una implementación del negocio que cederá el control al proyecto EJB. De este modo, los Action que solicitan realizar una tarea a un BO no saben que realmente detrás hay un EJB.
   A su vez, como ya hemos colocado los interfaces de negocio en un proyecto aparte, tanto el proyecto web, el business delegate, como el ejb y la implementación de negocio pueden reutilizar estos interfaces que serán comunes a todas las implementaciones. De este modo nos aseguramos que desde un *Action* lo único que puede hacer un programador es instanciar una `FactoriaBOs` que le devuelve

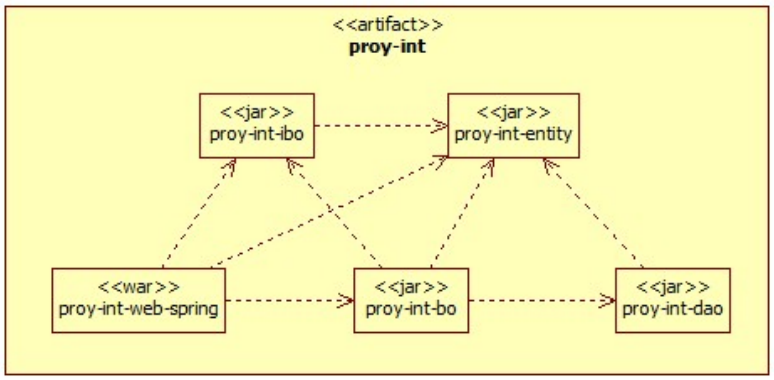
interfaces de negocio, y evitamos que pueda instanciar directamente un EJB o un DAO.

3. Una vez creados todos los proyectos, definiremos correctamente todas sus interdependencias. Desde el punto de vista del proyecto web con *Struts* que utiliza EJBs tenemos:



Refactorización EAR

Del mismo modo, con el proyecto web de *Spring* necesitamos menos proyectos, ya que eliminamos las dependencias con los EJBs, lo que nos permitiría desplegar este aplicación dentro de un contenedor web:



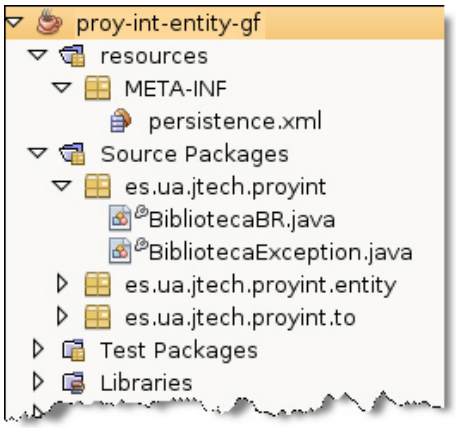
Refactorización EAR Spring

13.2. Refactorizando Paso a Paso

Para empezar, tal como hemos comentado previamente, vamos a dividir el proyecto común.

13.2.1. proy-int-entity

Nuestro primer nuevo proyecto será `proy-int-entity`, siendo un proyecto de librería java que contendrá las entidades, los *Transfer Objects*, la excepción padre de la aplicación (`BibliotecaException`) y las reglas de negocio (`BibliotecaBR`). El hecho de incluir tanto la excepción como las reglas de negocio se debe a que este proyecto va a ser utilizado por todos, y es el lugar idóneo para colocar ambas clases. Idealmente estas dos clases deberían ir en otro proyecto (tipo util) para contener aquellos elementos comunes a toda la aplicación.



Proyecto con los Entities

A su vez, este proyecto necesitará definir en sus librerías el uso de las librerías de Hibernate JPA, ya que nuestras entidades están anotadas.

Muy Importante

Todos aquellos proyectos que utilicen este proyecto (que van a ser todos), también deben incluir la librería de Hibernate JPA para poder interpretar las anotaciones, aunque no las usen.

13.2.2. proy-int-dao

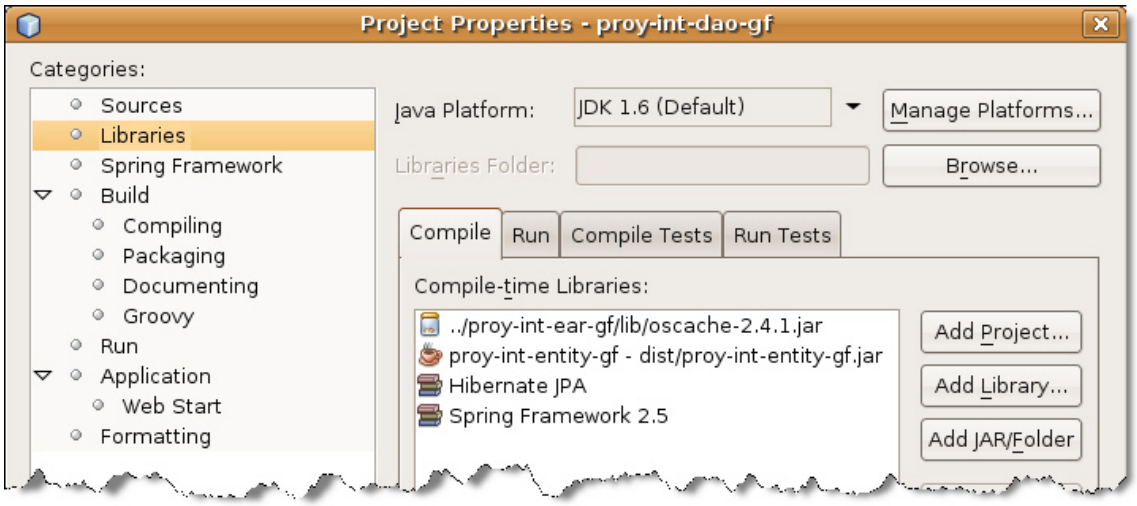
Este proyecto contendrá todas las clases que teníamos en el subpaquete dao, incluyendo el `CacheManager` que utilizabamos para acceder a la caché de *OSCache*. Por lo tanto, también necesitaremos su librería.

Recordad que dentro del proyecto común también teníamos una serie de pruebas sobre los DAOs, y por tanto, también las moveremos a este proyecto.

Colocando las Librerías

En este momento tenemos todas las librerías dentro del proyecto común, aunque luego desde el EAR estamos accediendo a ellas. Vamos a mejorar esto y vamos a colocar todas las librerías de terceros dentro de una carpeta `lib` dentro del EAR.

Por lo tanto, si accedemos a la vista de librerías de las propiedades del proyecto obtendremos algo semejante a esto:



Librerias del proyecto DAO

13.2.2.1. DAOs con Spring

Para minimizar el código que tenemos que modificar para poder usar nuestros DAOs con *Spring* y no tener que reescribir lo que ya tenemos, podemos reutilizar las clases `xxxxxJPAEEDAO`, a las cuales deberemos:

- Anotar con `@Repository`
- Anotar nuestra instancia de `EntityManager` con `@PersistenceContext`
- Crear un constructor por defecto para permitir la inyección de dependencias via Spring

De este modo, por ejemplo, el DAO de Operación quedaría similar a:

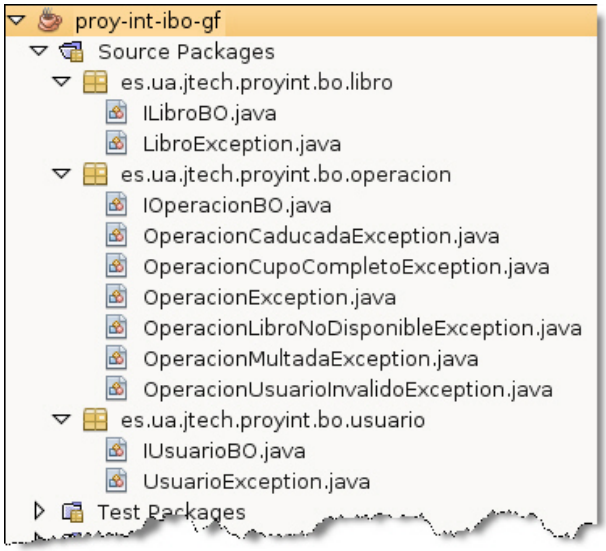
```
@Repository
public class OperacionJPAEEDAO implements IOperacionDAO {

    @PersistenceContext
    private EntityManager em;

    public OperacionJPAEEDAO() {}

    public OperacionJPAEEDAO(EntityManager em) {
        this.em = em;
    }

    // Métodos del DAO
```

Posteriormente, en el archivo de configuración de *Spring* (que nosotros hemos colocado en el proyecto bo, donde reside el negocio con *Spring*), definiremos tanto el `EntityManagerFactory` encargado de crear los `EntityManager` como un procesador de

estas etiquetas JPA en clases POJO (recordad que la anotación `@PersistenceContext` solo la podemos usar en componentes gestionados por el servidor de aplicaciones, ya sean *Servlets*, *EJBs*, etc...).

13.2.3. proy-int-ibo

Dentro de este proyecto vamos a colocar tanto las interfaces como las interfaces de negocio. De este modo, diferentes implementaciones de negocio, ya sea mediante POJOs o EJBs, podrán compartir los interfaces sin crear dependencias circulares.

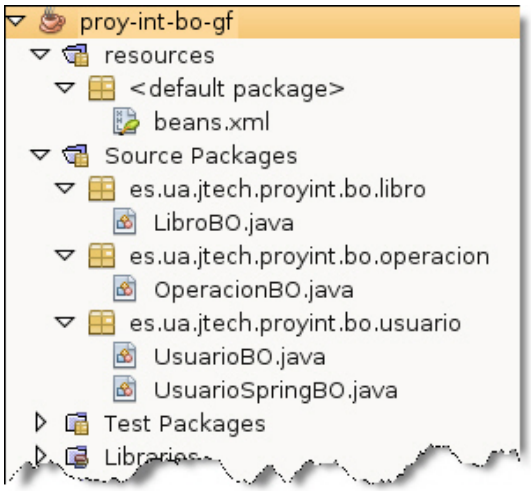


Contenido de los interfaces de negocio

En cuanto a las dependencias, solo dependerá de las entidades, y por tanto también deberá incluir las librerías de JPA

13.2.4. proy-int-bo

Esté proyecto contendrá las implementaciones de negocio, ya sean mediante POJOs simples o *Spring*, es decir, aquellas implementaciones que no requieran de un servidor de aplicaciones.



Contenido de negocio

En cuanto a los proyectos con lo que tiene dependencias, necesita a las entidades, los interfaces de negocio y los DAOs. Recordar que al tener una implementación de *Spring* necesitamos las librerías de *Spring*, así como aquellas librerías de terceros que hayáis podido utilizar (en la solución hemos utilizado `commons-lang`, la cual hemos colocado en la carpeta `lib` del EAR).

13.2.4.1. BOs con Spring

Para dar un mayor soporte a la lógica de negocio con *Spring*, vamos a anotar los POJOs *Spring* con anotaciones transaccionales, anotando los servicios con `@Transactional(propagation=Propagation.REQUIRED)`, y configuraremos el gestor transaccional con JTA (porque usamos *Spring* dentro de *Glassfish*).

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
       xmlns:jee="http://www.springframework.org/schema/jee"
       xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
       http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd
       http://www.springframework.org/schema/jee
       http://www.springframework.org/schema/jee/spring-jee-2.5.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">

    <jee:jndi-lookup id="miEmf" jndi-name="BibliotecaEE"
resource-ref="true"/>
    <jee:jndi-lookup id="miDS" jndi-name="jdbc/biblioteca"
resource-ref="false"/>
```

```
    <context:component-scan base-package="es.ua.jtech.proyint"/>

    <bean id="miTxManager"
class="org.springframework.transaction.jta.JtaTransactionManager"
/>

    <!-- Para que Spring interprete las anotaciones JPA e inyecte
el EntityManager -->
    <bean
class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor"
/>

    <tx:annotation-driven transaction-manager="miTxManager"/>
</beans>
```

Para hacer uso de las anotaciones JPA dentro de DAOs de *Spring*, tal como habíamos comentado, necesitamos definir la factoría de `EntityManager`, la cual no podemos obtenerla vía JNDI y por tanto necesitaremos su referencia como recurso en las aplicaciones web que utilicen este proyecto. Además, hemos de definir el procesador de etiquetas JPA (`PersistenceAnnotationBeanPostProcessor`).

En cuanto al gestor de transacciones, si quisiéramos usar esta capa de negocio en un proyecto web Spring fuera de un servidor de aplicaciones, nuestro gestor de transacciones deberá ser JPA o JDBC, dependiendo de la tecnología de acceso a datos que queramos usar.

13.2.5. proy-int-ejb

Este proyecto contendrá las implementaciones mediante EJB de las interfaces de negocio, es decir, lo mismo que teníamos hasta ahora menos la factoría. Si quisieramos, ahora podríamos simplificar estos componentes cediendole el control a los POJOs de aquellos métodos que no cambian (todos aquellos que hacen uso de JMS).

Gracias a la especificación EJB 3.0, la decisión de incluir EJBs en nuestra aplicación no es tan intrusiva. Dejando de lado los recursos hardware necesarios, la necesidad de desplegar toda la aplicación en cada cambio y de un servidor de aplicaciones, en lo referente al código a implementar, los cambios necesarios son mínimos.

Al introducir los EJBs, el interfaz de negocio se mantiene, pero la implementación de los BOs cambia, ya que damos todo el control al contenedor para que gestione las transacciones de la aplicación. Al compartir los interfaces, podemos hacer que un EJB no sea más que un delegador, que con el solo hecho de ceder el control a la implementación POJO del BO ya le está otorgando las características de los EJBs. Con poco código, muchas ventajas.

Por ejemplo, vamos a suponer que centralizamos toda la lógica en los BOs mediante POJOs, y que los EJBs les ceden el control, pero cubriéndolos con el paraguas transaccional que ofrecen. Así pues, si antes, dentro del EJB de libro teníamos esto:

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
```

```
public void actualizaLibro(LibroEntity libro) throws LibroException
{
    if (libro == null) {
        context.setRollbackOnly();
        throw new IllegalArgumentException("Se esperaba un libro");
    }

    ILibroDAO dao = FactoriaDAOs.getInstance().getLibroDAO(em);

    try {
        int numAct = dao.updateLibro(libro);
        if (numAct == 0) {
            throw new LibroException("No ha actualizado ningun
libro");
        }
    } catch (DAOException daoe) {
        context.setRollbackOnly();
        throw new LibroException("Error actualizando libro", daoe);
    }
}
```

Ahora tendremos:

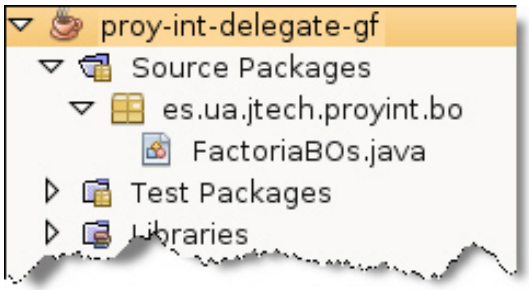
```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void actualizaLibro(LibroEntity libro) throws LibroException
{
    ILibroBO bo = new LibroBO();
    try {
        bo.actualizaLibro(libro);
    } catch (LibroException le) {
        context.setRollbackOnly();
        throw new LibroException("Error actualizando libro", le);
    }
}
```

Respecto a las dependencias, aquellos proyectos que son comunes al EAR, ya sean librerías de terceros o proyectos de aplicación no se archivan con el proyecto ejb, ya que formarán parte del EAR, por lo tanto dentro de la configuración de librerías del proyecto, no haremos el *package* de ningún proyecto dependendiente.

13.2.6. proy-int-delegate

El primer objetivo es que la capa de presentación no se vea modificada por la inclusión de EJBs. Cuando en la sesión de EJB se incluyeron los EJBs tuvimos que modificar todas nuestras acciones para referenciar a la nueva factoría. Vamos a arreglar esto creando el proyecto delegate, el cual incluirá una factoría que decidirá sin instancia EJBs o POJOs.

Esta factoría puede ser tanto la que teníamos en el proyecto común como en el ejb. Para que la aplicación siga usando los EJBs, moveremos la factoría del proyecto EJB a este nuevo proyecto, y renombraremos los métodos para que no haga referencia a los EJBs.



Proyecto Delegate

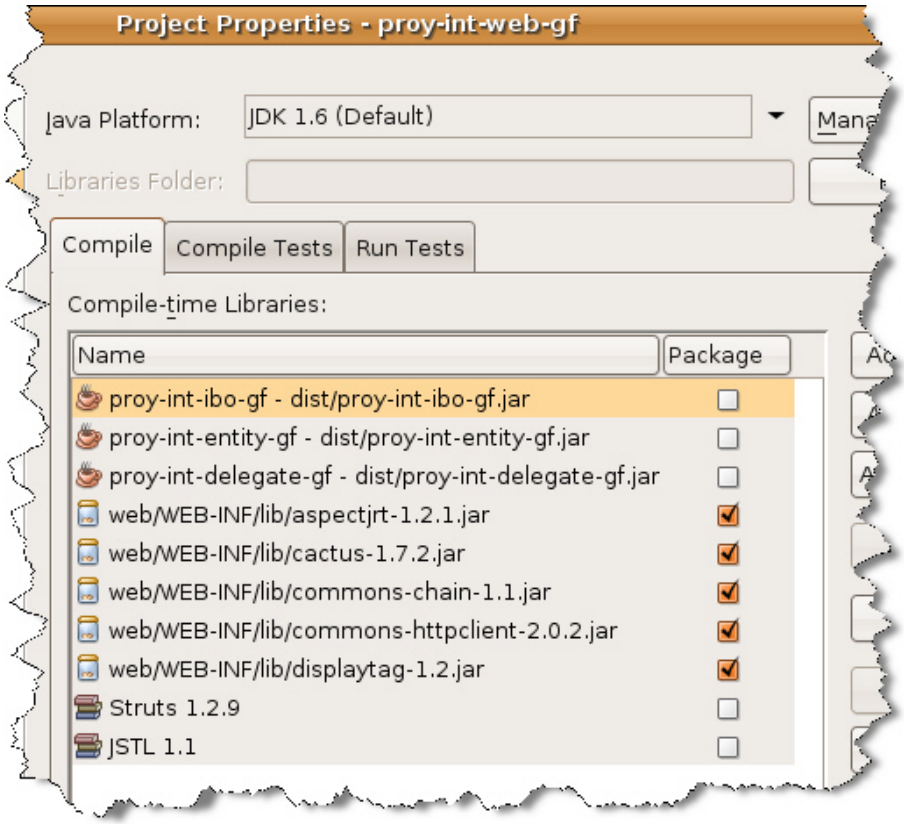
Antes de EJB 3.0

Antes de los EJBs 3.0, al crear un EJB había que llamar al método `create` que te devolvía el interfaz `Home` y luego al `remove`, esto para cada llamada a un método del EJB. Para poder encapsular estas llamadas y abstraerlas de la capa web, para interfaz de negocio se crea una nueva implementación (por ejemplo, `UsuarioDelegateBO`), que para cada método creaba y devolvía al pool el EJB en uso. Además, se hacía un uso intenso del patrón *ServiceLocator* para aislar en un único punto todos los accesos JNDI.

13.2.7. proy-int-web

Este proyecto web contiene la implementación *Struts* de los casos de uso del administrador y del bibliotecario. Además, vamos a dejar que esta implementación haga uso del delegate para que decida si utiliza EJBs o no. Por lo tanto, contendrá el mismo código que teníamos antes de refactorizar.

En cuanto a las dependencias, destacar que como librerías propias a exportar solo contiene las de Cactus y las displaytags (para la parte optativa de la administración).

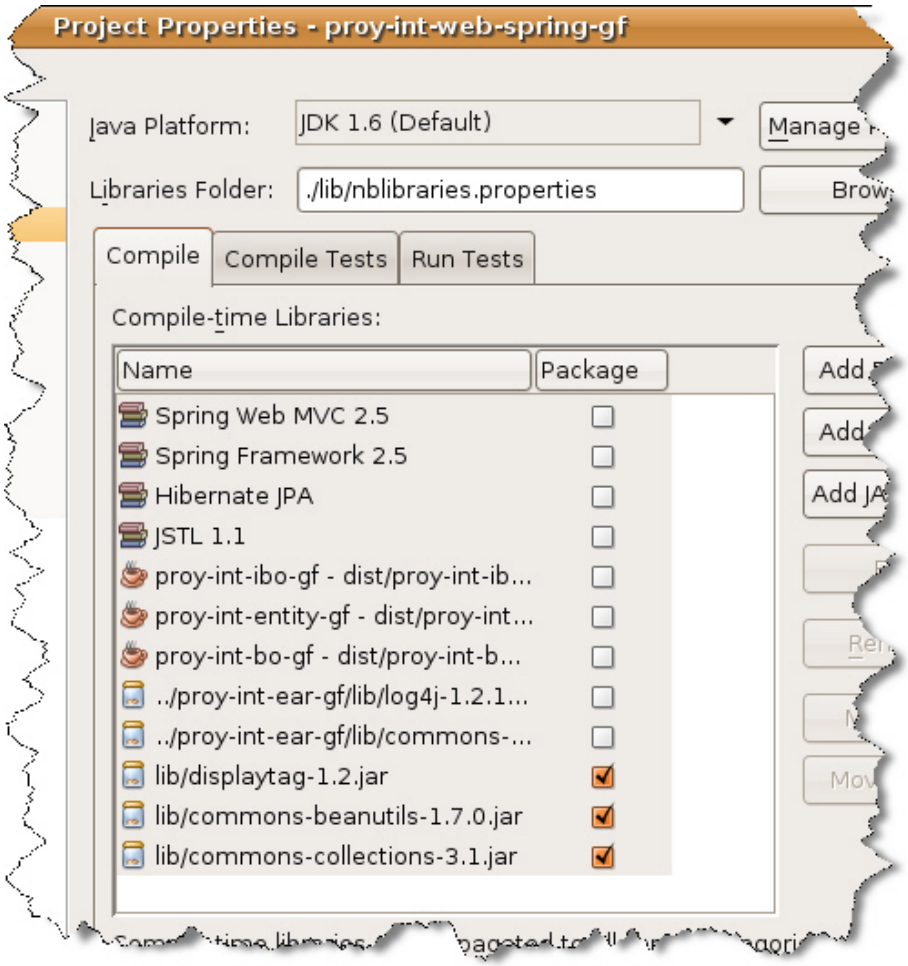


Proyecto Web

Igual que con los EJBS, destacar que aquellos proyectos que son comunes al EAR, ya sean librerías de terceros o proyectos de aplicación no se archivan con la aplicación web, ya que formarán parte del EAR.

13.2.8. proy-int-web-spring

Para terminar, veremos el proyecto con Spring. Igual que en el proyecto web, solo empaquetamos aquellas librerías especificas de este proyecto:



Dependencias Proyecto Spring

Para permitir el acceso al `EntityManager` gestionado por *Glassfish*, hemos de crear una referencia al mismo dentro de nuestro `web.xml`:

```
<persistence-unit-ref>
<persistence-unit-ref-name>BibliotecaEE</persistence-unit-ref-name>
    <persistence-unit-name>BibliotecaEE</persistence-unit-name>
</persistence-unit-ref>
```

13.2.8.1. i18n

Para poder acceder a los mensajes vía i18n antes de cargar ningún controlador, hemos de crear otro archivo de configuración (`beans-web.xml`), el cual define la fuente de mensajes del mismo modo que hacíamos en `dispatcher-servlet.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ....>
    <bean id="messageSource"
```

```
class="org.springframework.context.support.ResourceBundleMessageSource">
            <property name="basename" value="mensajes"/>
    </bean>
</beans>
```

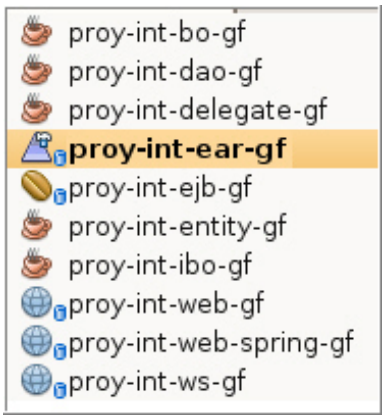
Y lo definiremos en el web.xml para que al cargar la aplicación, también cargue este archivo:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:beans.xml, /WEB-INF/beans-web.xml
    </param-value>
</context-param>
```

13.3. Resultado de la Refactorización

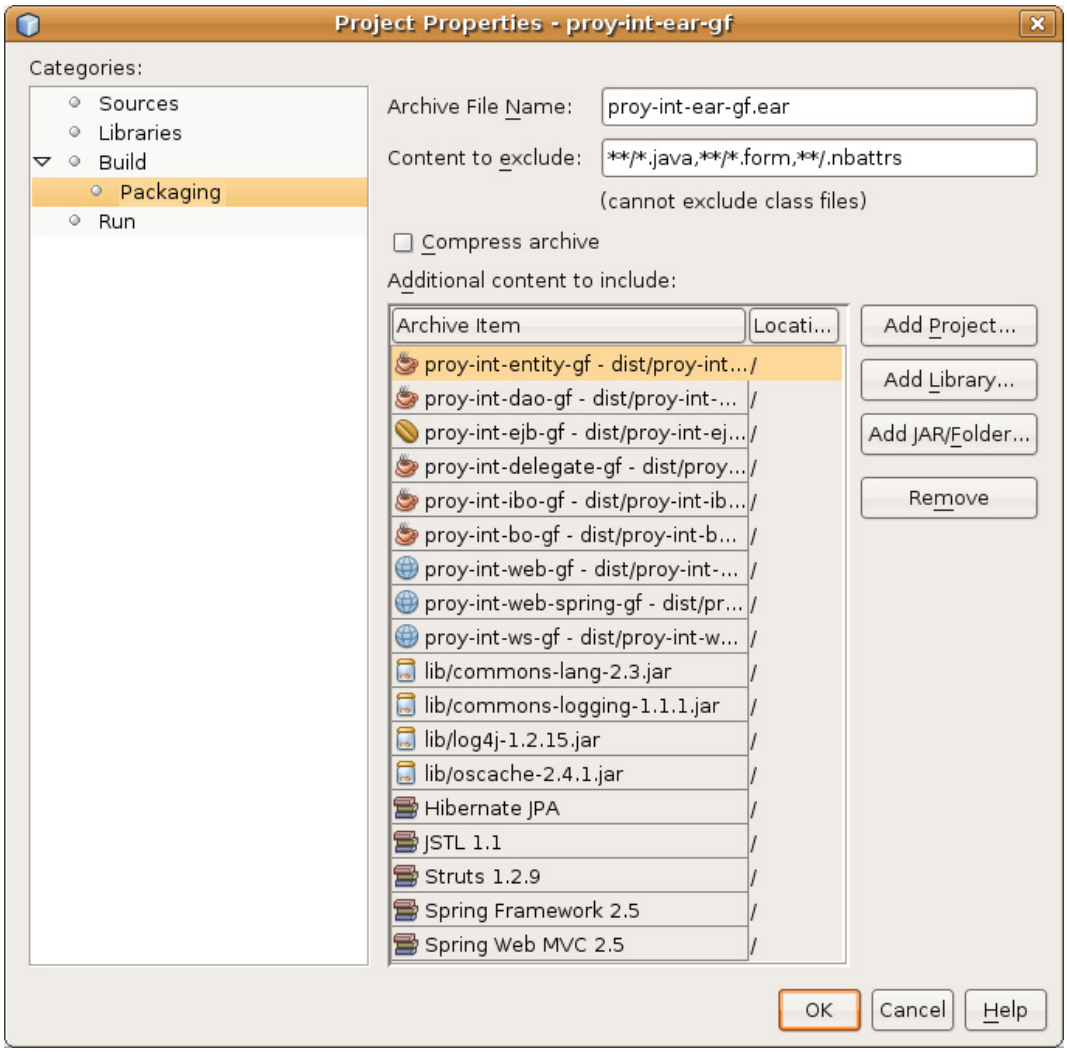
Al definir las interdependencias entre todos estos proyectos, aseguramos la calidad de código necesaria, de modo que cada capa accede a los objetos que debe. Mediante este tipo de arquitectura y la separación lógica de los proyectos evitamos errores de codificación y mejoramos la cohesión del código. Si anteriormente teníamos algún *Action* que instanciaba un DAO, ahora tendremos un error a resolver.

En resumen, tras refactorizar la aplicación tenemos que tener los siguientes proyectos:



Proyectos Netbeans

Además, en el caso de la aplicación EAR, deberá archivar todos los proyectos y librerías que utiliza.



Proyectos del EAR

Biblioteca en Spring

Si quisiéramos desplegar la aplicación basada en *Spring* en *Tomcat*, el proyecto web de *Spring* debería archivar todos las librerías, así como adjuntar el proyecto con los DAOs. Además, deberíamos modificar la gestión de las transacciones (para no usar JPA).

13.4. Web del Socio

Para terminar el proyecto, nos queda realizar la web socio/profesor. Vamos a intentar reutilizar gran parte del código que ya tenemos y realizar un interfaz adecuado al rol del usuario. **Se deja a vuestra elección la tecnología a utilizar**, ya sea *Struts* con *POJOs* o *EJBs*, o mediante *Spring*. Los casos de uso que puede realizar un socio son:

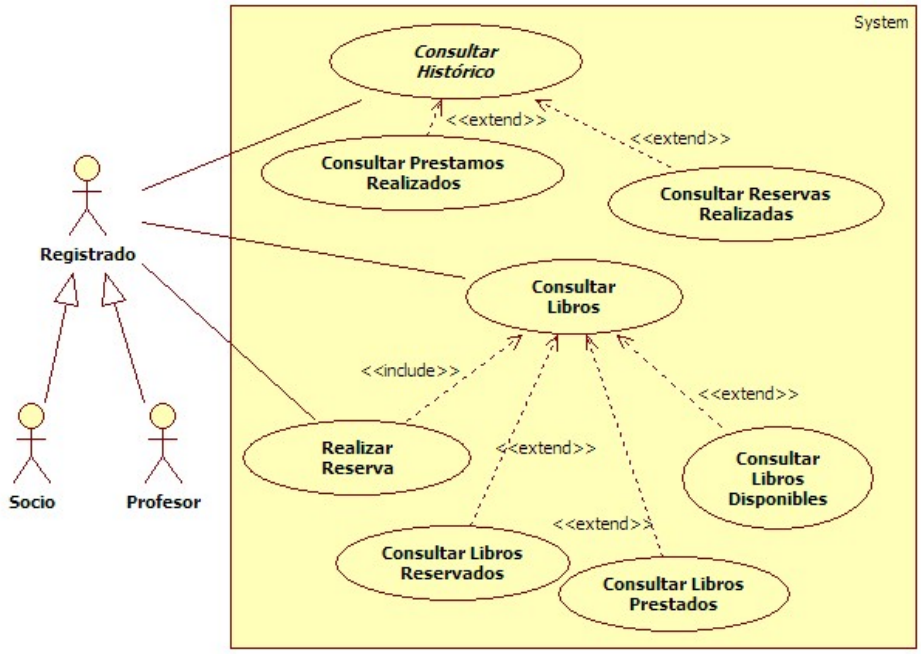


Diagrama de Casos de Uso detallado para el Usuario Registrado

Nuevo...

Podemos observar que únicamente tendremos que codificar desde 0 las operaciones de consultas al histórico.

Así pues, cuando un usuario de este tipo entra en la aplicación se encontrará con un interfaz similar al siguiente (fijaos como **no puede editar ni borrar libros, y únicamente puede realizar reservas sobre los libros disponibles, y anular las reservas que él mismo haya realizado**):

Web del Socio

13.4.1. Login de un Socio/Profesor

Nada más entrar a la aplicación, con un usuario cuyo rol sea socio o profesor, hemos de comprobar si el usuario es moroso, para así evitar que realice más operaciones. Si no es moroso, además comprobaremos si tiene préstamos pendientes de devolver:

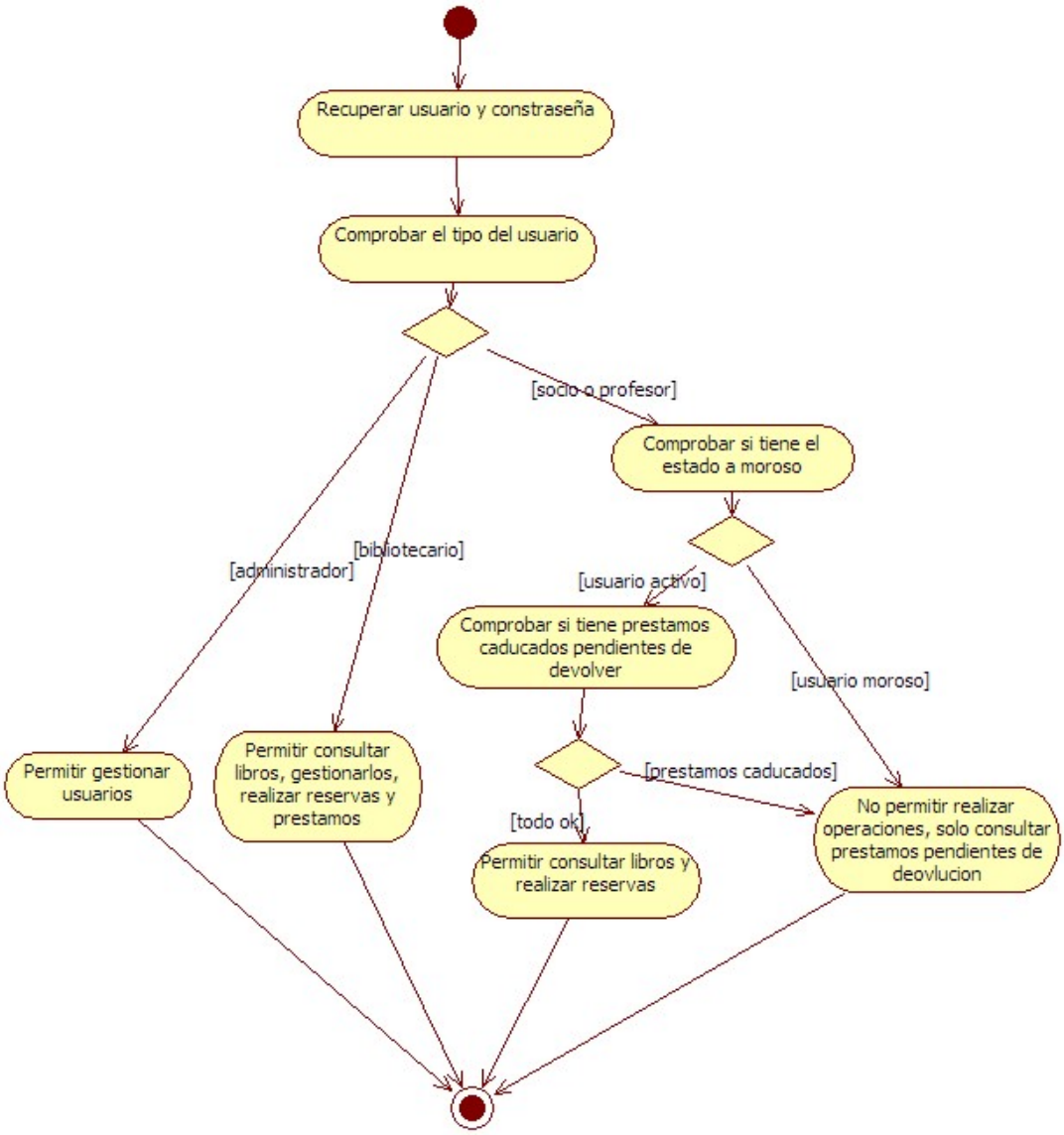


Diagrama de Actividad para la entrada al sistema

En el caso de que el usuario sea moroso o tenga libros pendientes de devolver, el interfaz impedirá que:

- en el caso de **moroso**, no puede hacer nada, sólo averiguar que fecha finaliza la multa
- en el caso de **pendiente de devolución**, averiguar que libros tiene en préstamo y que conllevarán un multa

Optativo I - Cerrando las Multas

En la sesión de JMS creamos un timer EJB que se encarga de caducar las reservas. A continuación, vamos a crear otro proceso automático pero que se encargará de pasar a histórico aquellas multas que hayan caducado. Al cerrar una multa, también se debe modificar el estado del usuario de `moroso` a `activo`.
Implementar vía *EJB Timer* o *Spring*

Por ejemplo, en el caso de que el usuario sea moroso, se le mostrará un interfaz similar a este:

Socio Moroso

Optativo II - Comprobando Devoluciones Pendientes

Se plantea como optativa la segunda posibilidad. Es decir, es obligatorio comprobar si el usuario tiene el estado de moroso, y se plantea como optativa la comprobación de si tiene préstamos pendiente de devolución, y en ese caso, mostrarle un listado con todos aquellos libros que le faltan por devolver

13.4.2. Listados de Libros con Struts

Para realizar los listados de libros (disponibles, reservados, prestados, todos), vamos a reutilizar los `Action` que ya tenemos implementados. Lo que vamos a modificar son los `ActionMapping`, de modo que el menú del socio tenga sus propios `ActionMapping` que referenciarán a los jsp dentro de la carpeta socio. A continuación mostramos como ejemplo la definición de dos `ActionMappings` que referencian al mismo `Action`, pero que pueden ejecutar diferente roles y que ceden el control a jsps diferentes.

```
<action path="/libroList"
type="es.ua.jtech.proyint.struts.acciones.libro.LibroListTodosAccion"
        roles="bibliotecario">
    <forward name="OK" path="/jsp/biblio/listadoLibros.jsp" />
</action>
<action path="/libroListSocio"
type="es.ua.jtech.proyint.struts.acciones.libro.LibroListTodosAccion"
        roles="socio, profesor">
    <forward name="OK" path="/jsp/socio/listadoLibros.jsp" />
```

```
</action>
```

Finalmente nos queda crear los jsp que contengan los listados de los libros. Para ello, nos basaremos en los listados del usuario bibliotecario, pero eliminando las opciones de editar/borrar libro, las operaciones de prestar y devolver préstamo, así como cambiando la referencia al menú del socio.

Web Service Amazon

Igual que con los listados, tendremos que crear un nuevo `ActionMapping` con el `LibroAmazonAccion` para que nos lleve a un jsp donde el menú sea el del socio, no el del bibliotecario

13.4.3. Reserva de Libros

Destacar que a la hora de realizar una reserva, ya no necesitamos ninguna pantalla intermedia, así pues, cuando el usuario pulse sobre el icono de **R**eserva (ya sea en el listado de todos los libros, o en de libros disponibles), se realizará la reserva y a continuación se le mostrará el listado de libros reservados.

Además, del mismo modo que hacíamos desde el punto de vista del bibliotecario, es necesario saber el rol del usuario que realiza la reserva (socio/profesor), para calcular los días de duración de la misma.

Mejora en el rendimiento

Como al hacer el login ya sabemos el rol del usuario, el cual queda almacenado en la sesión, las operaciones de reserva y préstamo podrían optimizarse de modo que recibieran un parámetro más, el del rol del usuario, y así evitar una consulta a la base de datos para averiguar dicho rol

13.4.4. Listado de Mis Libros

En este apartado se deben listar las reservas y préstamos que ha realizado un usuario, mediante la consulta al histórico. Destacar que ambos listados devuelven `OperacionHistoricoEntity`, y no `OperacionActivaEntity` como hasta ahora. A continuación, se muestra un ejemplo de listado de reservas realizadas por un determinado usuario.

Cerrar la sesión »

Usuario: **patricia** - Rol: **socio**

Listado de Mis Reservas (2)

isbn	titulo	autor	fecha inicio	fecha fin	fecha fin real
0131401572	Data Access Patterns	Clifton Nock.	23/11/06	27/11/06	26/11/06
0471768944	Service-Oriented Architecture (SOA):	Eric A. Marks and Michael Bell	04/06/08	09/06/08	04/06/08

Listado de Mis Reservas

Optativo III

Se plantea como ejercicio optativo adjuntar a este listado tanto las reservas/préstamos que tiene activas, como las que tiene en el histórico. En el caso de las reservas activas, debe ofrecer la posibilidad de Anular la Reserva

13.5. Entrega

La parte obligatoria a entregar es la siguiente:

- Refactorizar la aplicación de modo que su arquitectura sea homogenea, todos los subsistemas sean operativos tanto vía EJBs como POJOS con o sin Spring. Es imprescindible que desde un componente web (ya sea un `Action` o un `Controller`) no se puedan instanciar EJBs ni DAOs y que se minimice la redundancia de código.
- Implementar la web del socio, tanto los listados generales, como los listado sobre el histórico del usuario

La parte optativa se centra en:

- Cerrar las multas que estando activas deberían estar en histórico.
- El login del usuario cuando, sin ser moroso, tiene préstamos pendientes de devolver.
- En los listados históricos, mostrando las operaciones activas.