Práctica 9

En esta práctica vamos a modificar la versión 3.3 de Notaneitor que ya incluye autenticación y separación en capas por interfaces y factorías. La modificación consistirá en hacer que la fachada de la capa de lógica esté implementada con Enterprise Java Beans (EJB).

Cómo sabes de las clases de teoría, los EJB se ejecutan dentro del contenedor específico de EJB y eso permite usar los servicios del contenedor. Entre otras cosas, ahora podremos reubicar la capa de lógica en una máquina separada (para permitir escalabilidad vertical y horizontal), controlar el acceso de usuarios a métodos de servicio o especificar el comportamiento transaccional.

Prueba de la versión Notaneitor v3.3

En esta práctica vamos a usar los dos contenedores JEE de WildFly: el contenedor Web para la parte web y el contendor de EJB para la capa de negocio y persistencia.

Antes de empezar a modificar vamos a asegurarnos de que funciona la versión de la que partimos se ejecuta correctamente.

1. **Levanta la base de datos** hsqldb. Ejecuta el bat /data/startup.bat
2. Haz el **despliegue de la aplicación en el Servidor WildFly**. Asegúrate de que la consola indica el despliegue correcto.
3. Haz una prueba desde el navegador. http://localhost:8280/Notaneitor_v3.3/

Si al arrancar la aplicación web aparece algún error en la barra de dirección "/j_security_check" o la página muestra el mensaje "El Metodo HTTP POST no es soportado por esta URL", borra las cookies y vuelve a abrir la página (también serviría abrir una nueva pestaña privada en el navegador).

Si el error persiste: vuelve a entrar a la URL **original** y pulsa el botón actualizar del navegador.

Creación de Notaneitor v4.0Web

Para la nueva versión duplicaremos el contenido del proyecto v3.3 completo en un nuevo directorio, lo renombraremos a **Notaneitor_v4.0Web** e importaremos el contenido tal cual en Eclipse.

Cambia el nombre del proyecto en la propiedad **display-name** del fichero **WebContent/WEB-INF/web.xml.** Cambia el nombre del **Context root** desde *Propiedades del proyecto -> Web Project Settings*.

Sobre este nuevo proyecto ahora haremos estas modificaciones:

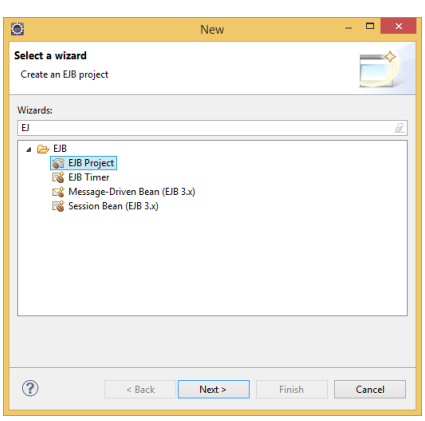
- Externalizar la lógica de negocio en otro proyecto (EJB), de forma que la aplicación esté dividida en dos proyectos: presentación y lógica de negocio.
- Hacer serializables las clases del modelo
- Creación de las nuevas factorías (Locators)
- Ajustes para el acceso a JNDI

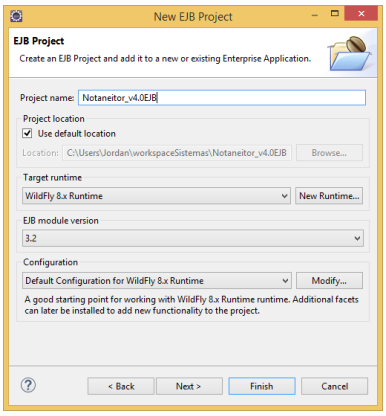
- Modificación del empaquetado creando un nuevo proyecto de tipo Enterprise Application Proyecto que agrupe el proyecto de presentación y lógica de negocio.

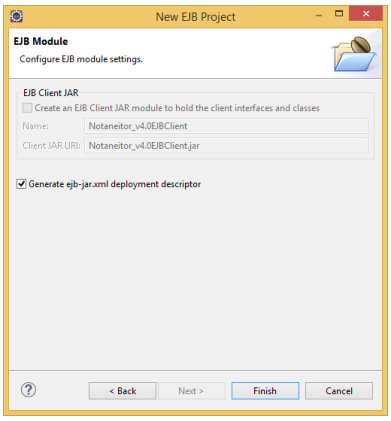
Creación de proyecto EJB

Necesitamos externalizar la lógica de negocio a otro proyecto y transformar la clase `SimpleAlumnosService` del paquete `com.sdi.business.impl` en un EJB.

Crearemos un nuevo proyecto de tipo **EJB Project** para gestionar toda la lógica de negocio del proyecto mediante el uso de EJBs. Llamamos al proyecto **Notaneitor_v4.0EJB**. En la tercera pantalla pulsamos sobre el checkbutton **"Generate ejb-jar.xml deployment description"**.





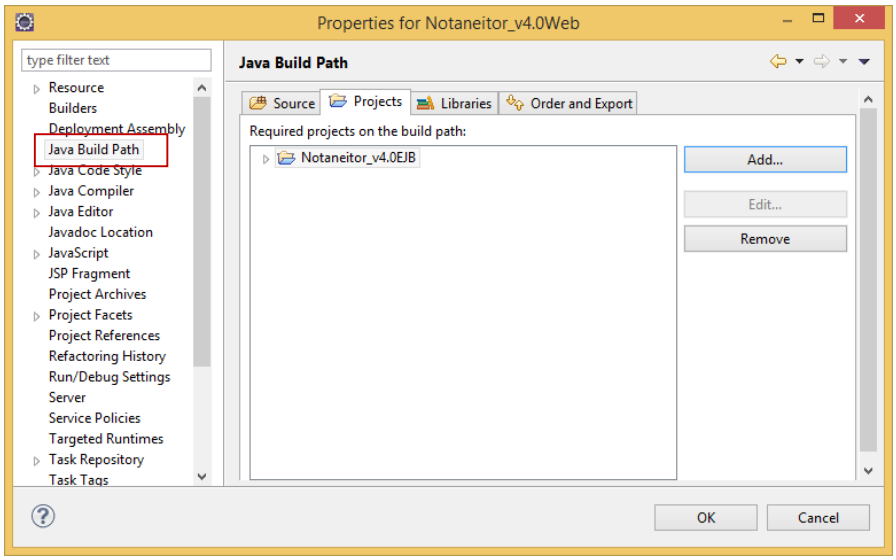


Moveremos al directorio **/ejbModule/** del nuevo proyecto **Notaneitor_v4.0EJB** los paquetes:

- com.sdi.business
- com.sdi.infraestructure
- com.sdi.model
- com.sdi.persistence

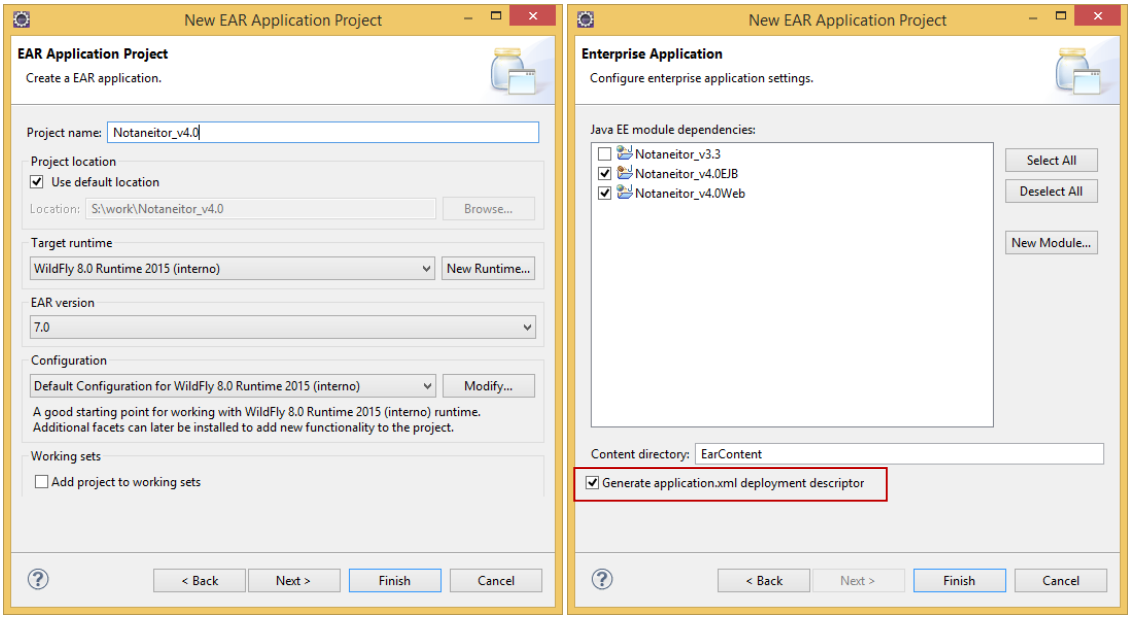
Únicamente mantendremos la capa de presentación en el proyecto original, de esta forma comenzamos a separar la presentación de la lógica de negocio. También moveremos al proyecto EJB la carpeta de la base de datos, ya que no tiene sentido que esté ubicada en la capa de presentación.

Agregamos una referencia al proyecto **Notaneitor_v4.0WEB,** *Properties* > *Java Build Path* > *Proyects* > *Add***,** agregamos el proyecto **Notaneitor_v4.0EJB** como referencia, con esta acción se deberían eliminar los errores por clases no encontradas.



Para desplegar de forma conjunta los dos proyectos debemos crear un tercer proyecto llamado **Notaneitor_v4.0** que será de tipo **Enteprise Application Project.**

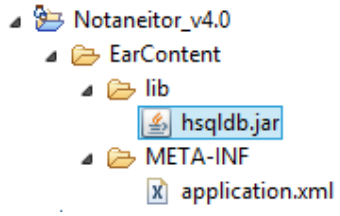
Seleccionamos los dos proyectos anteriores, y marcamos el CheckBox **Generate application.xml deployment descriptor.**



Comprobamos que el fichero **application.xml** localizado en **EarContent/META-INF** le está dando un nombre lógico a la aplicación web (correspondiente con el nombre del proyecto).

```
<module>
    <web>
      <web-uri>Notaneitor_v4.0Web.war</web-uri>
      <context-root>Notaneitor_v4.0Web</context-root>
    </web>

  </module>
```

Para asegurarnos que el proyecto EJB encontrará el driver de la base de datos creamos la carpeta **EarContent/lib/** y copiamos dentro de ella el fichero **hsqldb.jar**



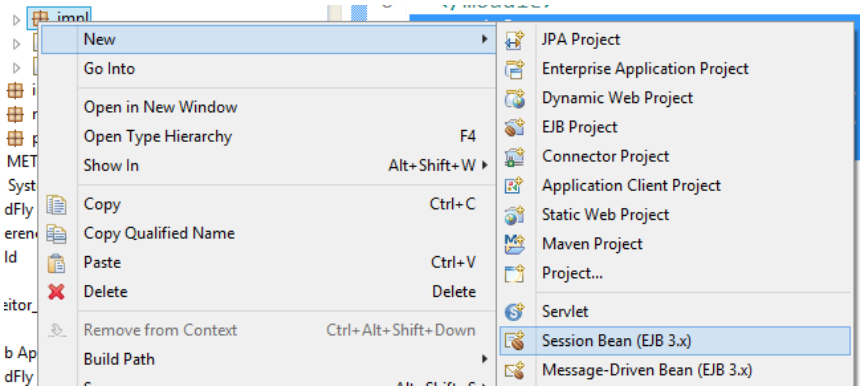
Comprobamos que todo el proceso hasta ahora ha sido correcto, ejecutamos el proyecto **Notaneitor_v4.0** y accedemos a la aplicación, no debería haber ningún cambio en el funcionamiento respecto a la versión anterior.

Crear los EJB

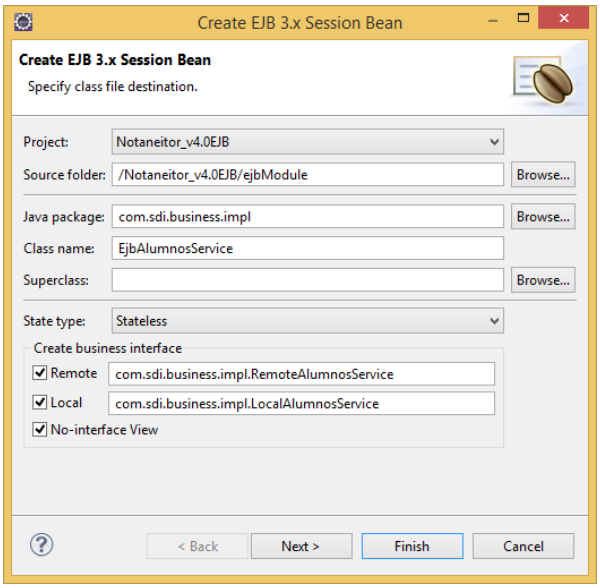
A continuación comenzaremos a utilizar los EJB en el proyecto **Notaneitor_v4.0EJB**, la primera funcionalidad a transformar será la relativa a los **Alumnos.**

Comenzamos creando el EJB **"EjbAlumnosService"** en el paquete **com.sdi.business.impl,** en este caso el servicio que prestará el EJB será sin estado ya que cada llamada al método que hagamos será independiente de las demás. Por lo tanto necesitamos un EJB de sesión sin estado "Stateless".

Creamos un EJB de nueva clase en el paquete **com.sdi.business.impl** utilizando el asistente, botón derecho, *New -> Session Bean (EJB 3.x).*



Marcamos que cree interfaces Remotos y Locales, con los nombre: **RemoteAlumnosService** y **LocalAlumnosService** respectivamente.



Continuamos y finalizamos la creación del EJB. Una vez creado hacemos que las dos interfaces extiendan de la interfaz **AlumnosService**.

```
@Remote
public interface RemoteAlumnosService extends AlumnosService {}

@Local
public interface LocalAlumnosService  extends AlumnosService {}
```

Implementamos los métodos de **EjbAlumnosService**, se trata de los mismos métodos que teníamos en su versión anterior **SimpleAlumnosService**, podemos copiarlos y pegarlos.

Observamos que en la declaración de la clase de **EjbAlumnosService** el asistente ha incluido por defecto una etiqueta **@LocalBean**, la eliminamos.

Hacer serializables las clases del modelo

Al permitir acceso remoto a la capa de lógica, los objetos que se intercambian esta capa y la de presentación deben viajar por la red. El mecanismo de serialización de java permite hacer eso automáticamente por nosotros pero las clases que necesiten este tipo de transporte deben **implementar la interfaz Serializable**. Es una interfaz que no añade métodos pero que le indica a la JVM que los objetos de esa clase son aptos para ser enviados a través de streams. De momento en esta mini-aplicación sólo tenemos una clase en el modelo.

```
public class Alumno implements Serializable {
	private String nombre;
	...
}
```

Creación de las nuevas factorías

A continuación vamos a modificar las antiguas factorías y transformarlas en "localizadores" de EJBs.

Podemos eliminar las clases **SimpleAlumnosService y SimpleServicesFactory** puesto que no vamos a volver a utilizarlas.

Accedemos a la clase **com.sdi.bussines.ServicesFactory** y renombramos el método createAlumnosService, ahora pasara a llamarse **getAlumnosService**, para que resulte más correcto conceptualmente.

```
package com.sdi.business;

public interface ServicesFactory {

	AlumnosService getAlumnosService();

}
```

Al cambiar la implementación de la clase que da soporte a la interfaz de la capa de lógica tenemos crear un Locator que encuentre y devuelva el EJB. Dado que vamos a permitir acceso local y remoto a la implementación de la capa de lógica, crearemos dos Locators.

Estos Locators podrán ser utilizados por la clase **com.sdi.infraestructure.Factories** para devolver el EJB apropiado en función de la configuración de la aplicación.

Creamos el Locator para acceso local en el paquete **com.sdi.business.impl** :

```
package com.sdi.business.impl;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import com.sdi.business.AlumnosService;
import com.sdi.business.ServicesFactory;

public class LocalEjbServicesLocator implements ServicesFactory {

	private static final String ALUMNOS_SERVICE_JNDI_KEY =
			"java:global/"
			+ "Notaneitor_v4.0/"
			+ "Notaneitor_v4.0EJB/"
			+ "EjbAlumnosService!"
			+ "com.sdi.business.impl.LocalAlumnosService";

	@Override
	public AlumnosService getAlumnosService() {

		try {
			Context ctx = new InitialContext();
			return (AlumnosService) ctx.lookup(ALUMNOS_SERVICE_JNDI_KEY);

		} catch (NamingException e) {
			throw new RuntimeException("JNDI problem", e);
		}
	}
}
```

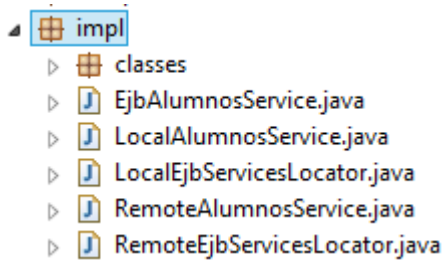
Lo único que hace es acceder al registro JNDI y obtener un objeto que implementa el servicio creado y gestionado por el contenedor. Como recordarás de teoría, el registro JNDI es una estructura de almacenamiento de objetos con organización jerárquica (algo similar a un árbol de directorios) que gestiona el contenedor por nosotros. El objeto de la clase *InitialContext* que creamos nos "conecta" con el registro y nos trae el objeto asociado a la clave "`java:global/Notaneitor_v4.0/Notaneitor_v4.0EJB/EjbAlumnosService!com.sdi.business.impl.LocalAlumnosService`".

Creamos el Locator para acceso remoto en el paquete **com.sdi.business.impl**:

```
public class RemoteEjbServicesLocator implements ServicesFactory {

	private static final String ALUMNOS_SERVICE_JNDI_KEY =
```

```
                        "Notaneitor_v4.0/"
                        + "Notaneitor_v4.0EJB/"
                        + "EjbAlumnosService!"
                        + "com.sdi.business.impl.RemoteAlumnosService";

        @Override
        public AlumnosService getAlumnosService() {
                System.out.println("Using remote services locator");
                try {
                        Context ctx = new InitialContext();
                        return (AlumnosService) ctx.lookup(ALUMNOS_SERVICE_JNDI_KEY);

                } catch (NamingException e) {
                        throw new RuntimeException("JNDI problem", e);
                }
        }
}
```

Observa que la clave de la interfaz remota no comienza por "`java:global/`". Eso es por cuestiones de configuración de Wildfly[1]. El paquete de implementación de la capa de negocio queda ahora así:



Para que la clase **com.sdi.infrastructure.Factories** devuelva la clase factoría adecuada es necesario hacerle una pequeña modificación:

```
public class Factories {
//      public static ServicesFactory services = new SimpleServicesFactory();
        public static ServicesFactory services = new LocalEjbServicesLocator();

        public static PersistenceFactory persistence = new SimplePersistenceFactory();
}
```

Para especificar el uso de beans en el **Notaneitor_v4.0Web** debemos crear un nuevo documento llamado **beans.xml** en la carpeta **/WebContent/WEB-INF/** con el siguiente contenido:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                       http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
   version="1.1"
   bean-discovery-mode="none">

</beans>
```

[1] Esta versión de Wildfly tiene una configuración de seguridad que impide acceder a claves bajo la rama java:global/ desde conexiones externas a la JVM que ejecuta el contenedor. Para poder hacer estas conexiones externas la interfaz remota también aparece en la rama java:jboss:exported/ que es mapeada directamente a / desde el exterior y a la que se accede con login y password (especificada en el jndi.properties).

Revisión de comunicación entre Web y EJBs

Si en el proyecto **Notaneitor_v4.0Web** estas utilizando la clase **BeanAlumno** desarrollada en proyectos anteriores te encontrarás con un pequeño error de comunicación entre el proyecto Web y el EJB.

Cuando la clase **BeanAlumnos** llama al método salvar(), obtiene el servicio AlumnosService del EJB correspondiente, utiliza este servicio para llamar al método saveAlumno() y updateAlumno(), pero en ambos casos no les envía el tipo de dato que realmente requieren.

SaveAlumno() y updateAlumno() esperan recibir un **Alumno** no un **BeanAlumno**. El proyecto EJB no está preparado para manejar **BeanAlumno**, tampoco debería ya que estos objetos pertenecen a la capa de presentación.

Para arreglarlo creamos un nuevo método en **BeanAlumno** el cual genere una copia de tipo **Alumno** con sus mismas propiedades:

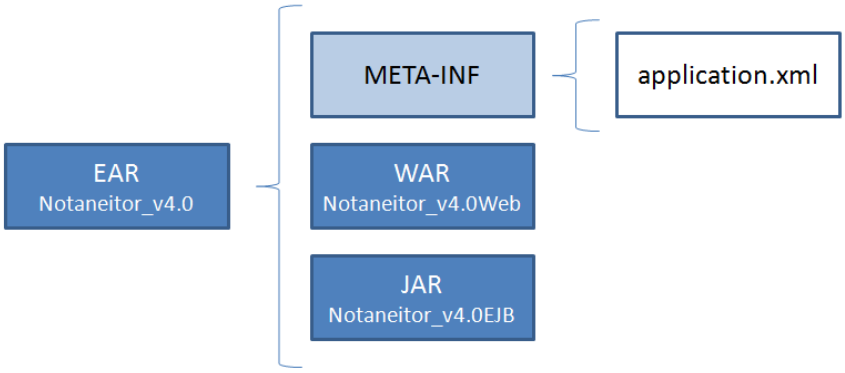
```
public Alumno getAlumnoBase(){
   Alumno base = new Alumno();
   base.setApellidos(getApellidos());
   base.setEmail(getEmail());
   base.setId(getId());
   base.setIduser(getIduser());
   base.setNombre(getNombre());
   return base;

}
```

Modificamos el parámetro que enviamos a los métodos saveAlumno y updateAlumno dentro de la clase **BeanAlumnos**.

```
public String salva() {

        AlumnosService service;
        try {
                service = Factories.services.getAlumnosService();
                if (alumno.getId() == null) {
                        service.saveAlumno(alumno.getAlumnoBase());
                } else {
                        service.updateAlumno(alumno.getAlumnoBase());

                }
```

Modificación del empaquetado

Ya hemos realizado las modificaciones oportunas en el proyecto, a continuación desplegaremos el proyecto **Notaneitor_v4.0.** Al ser un proyecto de tipo EAR, contiene la parte Web y la lógica de negocio EJB.



Despliegue y prueba

Si todo está en su sitio ahora bastaría con desplegar la aplicación **Notaneitor_v4.0**. Observamos en la consola del servidor que los EJB se están desplegando.

```
java:global/Notaneitor_v4.0/Notaneitor_v4.0EJB/EjbAlumnosService!com.sdi.business.impl.LocalAlumnosService
java:app/Notaneitor_v4.0EJB/EjbAlumnosService!com.sdi.business.impl.LocalAlumnosService
java:module/EjbAlumnosService!com.sdi.business.impl.LocalAlumnosService
java:global/Notaneitor_v4.0/Notaneitor_v4.0EJB/EjbAlumnosService!com.sdi.business.impl.RemoteAlumnosService
java:app/Notaneitor_v4.0EJB/EjbAlumnosService!com.sdi.business.impl.RemoteAlumnosService
java:module/EjbAlumnosService!com.sdi.business.impl.RemoteAlumnosService
java:jboss/exported/Notaneitor_v4.0/Notaneitor_v4.0EJB/EjbAlumnosService!com.sdi.business.impl.RemoteAlumnosService
```

La consola nos está diciendo que hay un EJB desplegado, que tiene varias entradas en el registro JNDI y cuáles son las claves bajo las que aparecerá. Cómo puedes comprobar son las que piden las dos factorías.

Prueba la aplicación en la misma URL de antes y la salida debería ser idéntica. Aparentemente no ha cambiado nada pero la nueva arquitectura tiene un mundo nuevo de posibilidades.

Modificación del DAO para unirse a la transacción

Haz una pequeña modificación al método de dar alta en **EjbAlumnosService**. Algo así:

```
@Override
public void saveAlumno(Alumno alumno) throws EntityAlreadyExistsException {
        new AlumnosAlta().save(alumno);
        throw new RuntimeException("forzamos un problema");
}
```

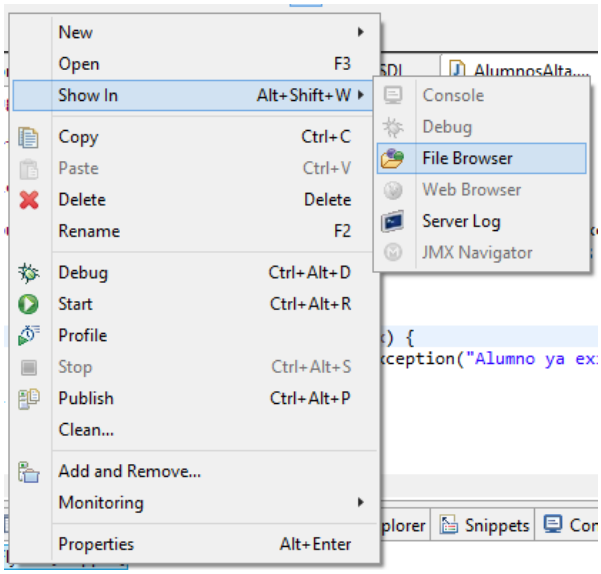
Si todo fuese realmente transaccional el nuevo alumno no debería estar en la base de datos. ¿Es realmente así? El contenedor funciona correctamente, la base de datos también, ¿dónde puede estar el problema? ¿Cómo se puede solucionar?

El problema viene de que el contendor y la base de datos no están coordinados. Los DAO abren **conexiones independientes** a la base de datos y por tanto el contenedor no controla la transacción. Para solucionarlo debemos hacer que los DAO obtengan la conexión de un Datasource servido por el contenedor.

Un Datasource es algo así como una factoría de conexiones y necesita configuración. En nuestro caso WildFly será el que creará los Datasources. Hay que especificarle información de la

conexión: a qué BDD, con qué driver, qué usuario y password y algún refinamiento extra (pool de conexiones, etc.). En esta versión de JBoss esto se resuelve con un descriptor en .xml.

Crea un fichero de nombre **notaneitor-ds.xml** con el contenido mostrado a continuación. **Cópialo** en la carpeta `<WildFly>\standalone\deployments` junto con el driver la BDD. Podemos acceder directamente a esta carpeta seleccionando el servidor desde la vista Servers del Eclipse, *botón derecho sobre el servidor -> Show in -> File Browser.*



Contenido de **notaneitor-ds.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources xmlns="http://www.jboss.org/ironjacamar/schema">
        <datasource
                jndi-name="java:jboss/datasources/NotaneitorDS"
                enabled="true"
                use-java-context="true"
                pool-name="NotaneitorDS">

                <connection-url>jdbc:hsqldb:hsql://localhost/localDB</connection-url>
                <driver>hsqldb.jar</driver>
                <pool></pool>
                <security>
                        <user-name>sa</user-name>
                        <password></password>
                </security>

        </datasource>
</datasources>
```

Observa el contenido del fichero. Está aportando la información necesaria para poder conectar a la BDD. La información que especifica este fichero de configuración es la misma que teníamos en los atributos de la clase AlumnoJdbcDAO.

Nota: asegurate de que el valor del elemento <connection-url> jdbc:hsqldb:hsql://localhost/**localDB**</connection-url> coincide con el alias de tu base de datos.

Puedes consultar el alias de la base de datos en el fichero **server.properties** de la carpeta **/data/**

Según indica el fichero, el driver de la base de datos (fichero **hsqldb.jar**) debe situarse en su mismo directorio `\standalone\deployments`.

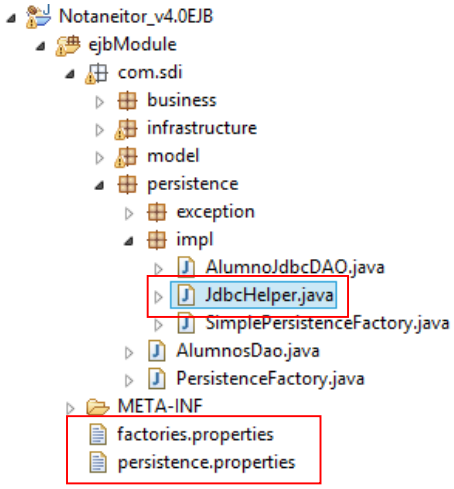
Nos resta modificar las clases DAO de forma que no abran la conexión como hasta ahora. La cuestión es ¿cómo obtienen el DataSource las clases DAO? Del registro JNDI, bajo la clave especificada en el fichero de configuración anterior

```
jndi-name="java:jboss/datasources/NotaneitorDS"
```

Como esto será código repetitivo en todos los métodos del DAO (todos abren y cierran conexiones) aprovecharemos para factorizar ese código añadiendo una clase Helper llamada **JdbcHelper** en el paquete **com.sdi.persistence.impl**.

Copia la clase JdbcHelper desde la carpeta de material al paquete com.sdi.persistence.impl.

Introduciremos todos los parámetros de configuración en un documento **persistence.properties**. Este fichero está en la carpeta material, cópialo en la carpeta **ejbModule** para que quede en el classpath.



Los parámetros de configuración deberían estar siempre definidos en ficheros de configuración, de forma que se pudieran modificar sin recompilar.

Si observas la implementación actual del DAO de Alumno verás que tiene varios string incrustados (los de conexión a la BDD y los de la única query que hace). Extrae todos ellos a un fichero de propiedades y usando como apoyo la clase JdbcHelper modifica adecuadamente la implementación de forma que se tomen esos datos desde un fichero de propiedades.

Modifica el código del **AlumnoJdbcDao** localizada en el paquete **com.sdi.persistence.impl** para comenzar a utilizar el Helper.

```
public class AlumnoJdbcDao implements AlumnoDao {
        private static String CONFIG_FILE = "/persistence.properties";

        private JdbcHelper jdbc = new JdbcHelper(CONFIG_FILE);

        public List<Alumno> getAlumnos() {
                PreparedStatement ps = null;
                ResultSet rs = null;
                Connection con = null;

                List<Alumno> alumnos = new LinkedList<Alumno>();
```

```
		try {
			con = jdbc.createConnection();
			ps = con.prepareStatement(jdbc.getSql("ALUMNOS_SELECT_ALL"));

			rs = ps.executeQuery();

			while (rs.next()) {
				Alumno alumno = new Alumno();
				alumno.setId(rs.getLong("ID"));
				alumno.setNombre(rs.getString("NOMBRE"));
				alumno.setApellidos(rs.getString("APELLIDOS"));
				alumno.setEmail(rs.getString("EMAIL"));
				alumno.setIduser(rs.getString("IDUSER"));

				alumnos.add(alumno);
			}
		} catch (SQLException e) {
			throw new PersistenceException("Invalid SQL or database schema", e);
		} finally {
			jdbc.close(ps, rs, con);
		}

		return alumnos;

	}
```

Como ves obtiene el datasource del JNDI. Y la clave la lee de un fichero de propiedades denominado ***persistence.properties***. Esto permite adaptar la aplicación en explotación sin recompilar.

```
#Clave JNDI bajo la cual se recupera el datasource
#	Si usamos EJB las conexiones a BDD nos las da el contenedor
JNDI_DATASOURCE = java:jboss/datasources/NotaneitorDS
```

Con esto se solventa el problema de la coordinación de la transacción. Despliega el proyecto **Notaneitor_v4.0** y prueba la aplicación.

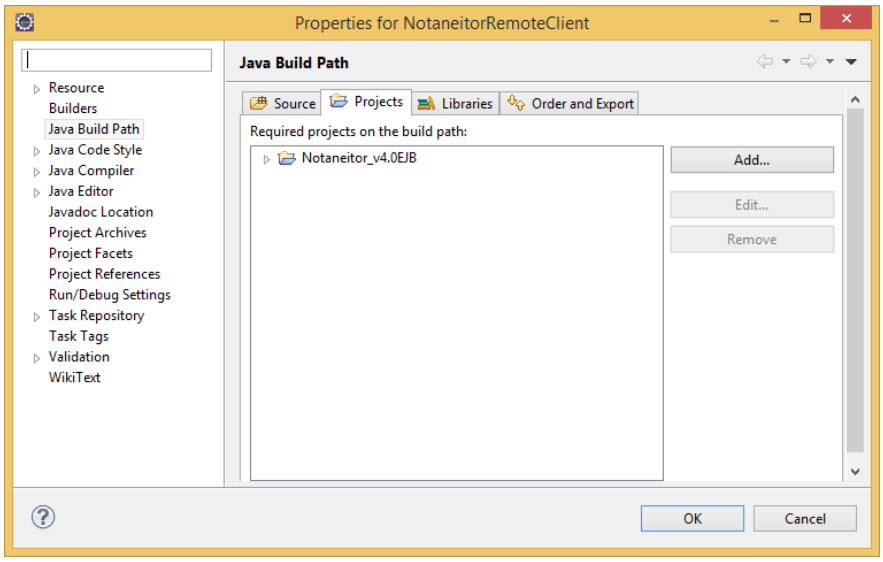
Acceso remoto al EJB desde Notaneitor Remote Client

Para comprobar la posibilidad de acceso a la capa de negocio desde programas externos desarrollaremos ahora un sencillo cliente que liste los alumnos registrados.

Creamos un nuevo proyecto Java llamado **NotaneitorRemoteClient**.

Agregamos una referencia al proyecto **Notaneitor_v4.0EJB** para poder utilizar sus clases (aunque habría otras posibilidades para lograr esto mismo)

Properties -> Java Build Path -> Projects -> Add.



Creamos el paquete **com.sdi.client** y dentro de él la clase **Main.**

```
private static final String ALUMNOS_SERVICE_JNDI_KEY =
                  "Notaneitor_v4.0/"
                + "Notaneitor_v4.0EJB/"
                + "EjbAlumnosService!"
                + "com.sdi.business.impl.RemoteAlumnosService";

private void run() throws Exception {
        Context ctx = new InitialContext();
        AlumnosService service = (AlumnosService) ctx.lookup(ALUMNOS_SERVICE_JNDI_KEY);
        List<Alumno> alumnos = service.getAlumnos();

        printHeader();
        for (Alumno a : alumnos) {
                printLine(a);
        }
}
```

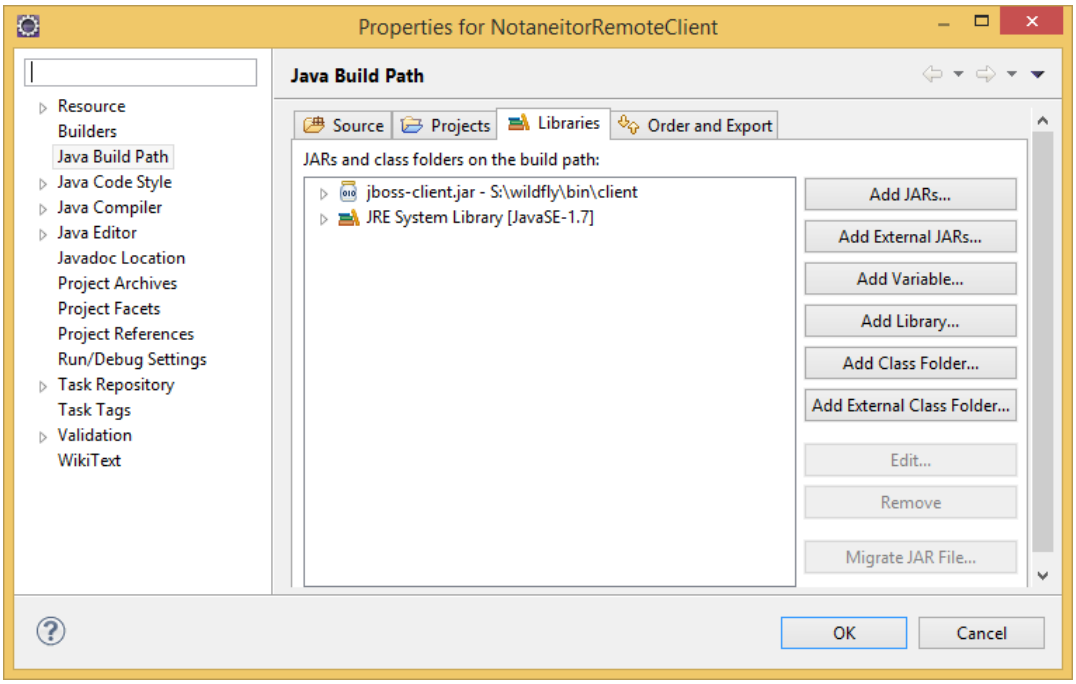
Llamaremos a este método run() desde el método Main.

Esta clase se ejecutará en una máquina Java distinta de la del servidor, por lo tanto necesitará hacer acceso remoto al servidor. Para ello necesita conectar con el registro JNDI (remoto) y obtener el stub al EJB. Para que todo eso funcione se necesitan bibliotecas con el código necesario para conectar al JNDI y hacer marshalling/unmarshalling de los datos. Esas bibliotecas forman lo que se llama el **contenedor del cliente**.

Ajuste del classpath

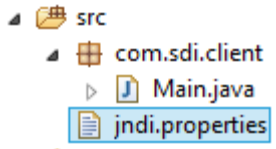
El cliente debe tener en el classpath las bibliotecas del contenedor del cliente y las clases que necesita ver del lado servidor (Alumno, AlumnosService, etc.). Para ello **añadiremos al classpath:**

- el jar `<WildFly>\bin\client\jboss-client.jar`
- el proyecto **Notaneitor_v4.0EJB** agregado anteriormente.



Ajustes para el acceso al servidor JNDI

Sólo nos resta añadir al classpath un fichero específico para WildFly que le indica a la clase InitialContext (la que nos conecta con el registro JNDI) en qué dirección y puerto está escuchando el servidor JNDI (en este caso es un servidor interno gestionado por WildFly). Para ello debemos añadir al directorio ./src el fichero **jndi.properties**.




```
#jboss JNDI properties
java.naming.factory.initial=org.jboss.naming.remote.client.InitialContextFactory
java.naming.provider.url=http-remoting://localhost:8280
java.naming.factory.url.pkgs=org.jboss.ejb.client.naming
java.naming.security.principal=user
java.naming.security.credentials=password
jboss.naming.client.ejb.context=true
```

La clase InitialContext está diseñada para buscar en el classpath un fichero con el nombre `jndi.properties` para inicializar las propiedades del contexto.

Refactorizaciones

Modificar la clase Factories

Según vimos en clase de teoría no debería ser necesario recompilar nada por hacer un cambio de configuración. En la práctica eso supondría que si quisiéramos hacer una reconfiguración de los servidores y necesitásemos poner la capa de lógica en otra máquina habría que usar la factoría de

RemoteEjbServicesFactory en vez de la local. Tal como están las cosas ahora habría que retocar, y recompilar, la clase **Factories**. Esta clase debería leer de un fichero de configuración, como el que se adjunta, cuál es la clase real que implementa la factoría y crear un objeto usando reflectividad.

factories.properties

```
SERVICES_FACTORY = com.sdi.business.impl.LocalEjbServicesLocator
PERSISTENCE_FACTORY = com.sdi.persistence.impl.SimplePersistenceFactory
```

Coloca el fichero de **factories.properties** en la carpeta **ejbModule** para que quede en el classpath.

Haz las modificaciones necesarias en la clase **com.sdi.infraestructure.Factories** y usa como apoyo la clase del fichero **FactoriesHelper.java**.

```
public class Factories {
	private static String CONFIG_FILE = "/factories.properties";

	public static ServicesFactory services = (ServicesFactory)
		FactoriesHelper.createFactory(CONFIG_FILE, "SERVICES_FACTORY");

	public static PersistenceFactory persistence = (PersistenceFactory)
		FactoriesHelper.createFactory(CONFIG_FILE, "PERSISTENCE_FACTORY");

	}
```

```
public class FactoriesHelper {

	/**
	 * Devuelve instancia de la clase factory deseada. Crea un objeto a partir
	 * del nombre de la clase
	 *
	 * @param file
	 *		El fichero de propiedades
	 * @param factoryType
	 *		El nombre de la propiedad en el fichero de proerties
	 * @return
	 *		El objeto de la clase factoria adecuada
	 */
	static Object createFactory(String file, String factoryType) {

		String className = loadProperty(file, factoryType);
		try {

			Class<?> clazz = Class.forName(className);
			return clazz.newInstance();

		} catch (ClassNotFoundException e) {
			throw new RuntimeException(e);
		} catch (InstantiationException e) {
			throw new RuntimeException(e);
		} catch (IllegalAccessException e) {
			throw new RuntimeException(e);
		}
	}

	/**
	 * Carga un propiedad desde fichero de propiedades
	 * Lanza runtime exception si no existe la propiedad o el fichero
	 *
	 * @param file
	 * @param property
```

```
	 * @return
	 */
	static String loadProperty(String file, String property) {

		Properties p = loadPropertiesFile(file);

		String value = p.getProperty(property);
		if (value == null){
			throw new RuntimeException("Property not found in " + file);
		}
		return value;
	}

	private static Properties loadPropertiesFile(String file) {
		Properties p = new Properties();
		try {
			InputStream is = Factories.class.getResourceAsStream(file);

			p.load(is);

			is.close();
		} catch (IOException e) {
			throw new RuntimeException(e);
		}
		return p;

	}
```