



Eusko Jaurlaritzaren Informatika Elkarte
Sociedad Informática del Gobierno Vasco

UDA - Utilidades de desarrollo de aplicaciones

Guía de desarrollo

Fecha: 06/06/2011

Referencia:

EJIE S.A.
Mediterráneo, 14
Tel. 945 01 73 00*
Fax. 945 01 73 01
01010 Vitoria-Gasteiz
Posta-kutxatila / Apartado: 809
01080 Vitoria-Gasteiz
www.ejje.es



[UDA – Utilidades de desarrollo de aplicaciones](#) by [EJIE](#) is licensed under a [Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 Unported License](#).

Control de documentación

Título de documento: Guía de desarrollo

Histórico de versiones

| Código: | Versión: | Fecha: | Resumen de cambios: |
|---------|----------|------------|---------------------------|
| | 1.0.0 | 06/06/2011 | Primera versión. |
| | 1.1.0 | 17/10/2011 | Cambio del sistema de log |
| | | | |
| | | | |

Cambios producidos desde la última versión

Capítulo Logging: cambio del sistema de gestión de ficheros de logs con SLF4J y Logback

Control de difusión

Responsable: Ander Martínez

Aprobado por:

Firma:

Fecha:

Distribución:

Referencias de archivo

Autor:

Nombre archivo:

Localización:

Contenido

| | Capítulo/sección | Página |
|--------|---|--------|
| 1 | Introducción | 1 |
| 2 | Visión Global | 2 |
| 2.1 | ¿Qué es UDA en realidad? | 2 |
| 2.2 | Funcionalidad inicial de una aplicación UDA | 2 |
| 2.2.1. | Funcionalidad del backend | 3 |
| 2.3 | Arquitectura de Capas | 18 |
| 2.4 | Estructura de proyectos | 22 |
| 2.5 | Nomenclatura y paquetería | 23 |
| 2.6 | Empaquetado de librerías | 28 |
| 3 | Presentación | 30 |
| 3.1 | Vista | 30 |
| 3.2 | Control | 40 |
| 3.2.1. | Navegación | 40 |
| 3.2.2. | Peticiones de datos | 41 |
| 3.2.3. | Interacción con capa de servicios de negocio | 42 |
| 3.2.4. | Gestionar Excepciones (desactualizado) | 43 |
| 3.2.5. | Métodos generados por UDA | 45 |
| 4 | Modelo de datos | 49 |
| 4.1 | Representación del esquema relacional en Java | 51 |
| 4.1.1. | Relación OneToOne (1:1) | 53 |
| 4.1.2. | Relación ManyToOne (M:1) | 55 |
| 4.1.3. | Relación OneToMany (1:N) | 56 |
| 4.1.4. | Relación MayToMany (M:N) | 57 |

| | | |
|--------|--|----|
| 4.2 | Reglas de validación -(desactualizado) | 59 |
| 4.3 | Serialización | 60 |
| 4.3.1. | Formato String | 60 |
| 4.3.2. | Serialización binaria | 60 |
| 4.3.3. | Formato JSON | 62 |
| 4.4 | Particularidades para JPA 2.0 | 63 |
| 4.4.1. | Metamodel | 63 |
| 4.4.2. | Data Transfer Objects | 64 |
| 4.4.3. | Tips & tricks | 67 |
| 5 | Servicios de negocio | 71 |
| 5.1 | Diseño Técnico de la capa de Servicios de negocio | 71 |
| 5.1.1. | Principios | 71 |
| 5.1.2. | Patrón Fachada | 71 |
| 5.2 | Implementación de la capa de servicios | 73 |
| 5.3 | Componentes y frameworks de UDA en la capa de Servicios | 74 |
| 5.4 | Acceso a los servicios. | 74 |
| 5.5 | Invocación de EJBs sin estado. | 74 |
| 6 | Acceso a Datos | 76 |
| 6.1 | Estrategias y convenciones | 76 |
| 6.1.1. | Query by example (QBE) | 76 |
| 6.2 | Object Relational Mapping (ORM) | 76 |
| 6.3 | Estrategia de carga de datos | 78 |
| 6.4 | Paginación | 78 |
| 6.5 | Estrategia de generación de código. Tablas, sinónimos y entidades. | 79 |
| 6.5.1. | Modelo relacional | 80 |
| 6.5.2. | Sinónimos | 81 |
| 6.6 | Demarcación transaccional | 81 |
| 6.7 | Java Database Connectivity (JDBC) | 82 |

| | |
|--|----------------|
| 6.7.1 Comportamientos específicos Spring JDBC | 82 |
| 6.7.2 Métodos generados por UDA | 86 |
| 6.8 Java Persistence API (JPA) 2.0 | 90 |
| 6.8.1 Comportamientos específicos JPA 2.0 | 90 |
| 6.8.2 Métodos generados por UDA | 96 |
| 6.9 Orígenes de datos | 100 |
| 6.9.1 Uso del Data Source desde JDBC | 104 |
| 6.9.2 Uso del Data Source desde JPA 2.0 | 105 |
| 7 Remoting | 106 |
| 7.1 Configuración de los servidores de aplicaciones WebLogic 11g | 106 |
| 7.2 Consumo de una aplicación UDA remota | 115 |
| 7.3 Consumo de una aplicación Geremua 2 remota | 120 |
| 8 Transaccionalidad | 124 |
| 8.1 Transacciones con anotaciones | 124 |
| 8.2 Gestión de transacciones | 125 |
| 8.3 Drivers XA. | 126 |
| 8.4 Transaccionalidad distribuida en la invocación a EJB 2.x y EJB 3.0 | 127 |
| 9. Internacionalización (completado en Anexo) | 129 |
| 9.1 Internacionalización de los proyectos estáticos | 129 |
| 9.2 Internacionalización de los proyectos dinámicos | 130 |
| 40. Validación (desactualizado) | 134 |
| 40.1 Validación unitaria desde la vista (desactualizado) | 135 |
| 40.2 Validación masiva desde la vista (desactualizado) | 136 |
| 44 Seguridad (desactualizado) | 138 |
| 44.1 Fundamentos tecnológicos (desactualizado) | 138 |
| 44.1.1 XLS (desactualizado) | 139 |
| 44.1.2 Spring Security, Authentication and Authorization Service (desactualizado) | 140 |

| | | |
|--------|---|-----|
| 11.1.3 | Wrapper del Sistema Perimetral para Spring Security (desactualizado) | 141 |
| 11.2 | Prerrequisitos (desactualizado) | 141 |
| 11.3 | Uso del sistema de seguridad (desactualizado) | 142 |
| 11.4 | Funcionamiento conceptual | 144 |
| 12 | Logging | 146 |
| 12.1 | LogBack | 147 |
| 12.2 | Ficheros de <i>logs</i> | 151 |
| 12.3 | Formato de las trazas | 153 |
| 12.4 | Trazas de usuario | 155 |
| 12.5 | Logs de aplicación e incidencia | 156 |
| 12.6 | Auditoría de accesos a Base de Datos (jdbcsllog) | 157 |
| 13 | Glosario de anotaciones | 161 |
| 14 | Equivalencias conceptuales con Geremua 2 | 163 |
| 14.1 | Componentes proporcionados por las aplicaciones | 163 |
| 14.2 | Componentes configurables por las aplicaciones | 164 |
| 14.3 | Componentes estructurales | 165 |
| 14.4 | Custom Tags | 166 |
| 15 | Bibliografía | 167 |

1 Introducción

UDA (Utilidades de Desarrollo de Aplicaciones) es el nuevo sistema de desarrollo de aplicaciones Java EE impulsado por EJIE/EJGV. Sus principales señas de identidad frente a sus predecesores son la **simplicidad y productividad** en el desarrollo y sus capacidades de interfaz gráfico para la construcción de aplicaciones ricas (RIA, Rich Internet Applications).

Estas herramientas, están basadas en estándares de facto (Spring Framework, SLF4J, etcétera) y estándares JEE5/JEE6 (EJB 3.0, Bean Validation, etcétera) que en su gran mayoría son proporcionados por productos o librerías Open Source. No obstante, UDA está optimizado para el Servidor de Aplicaciones Oracle Weblogic 11g que requiere una licencia comercial.

A continuación, se muestra un gráfico que resume la arquitectura de componentes utilizados en UDA:

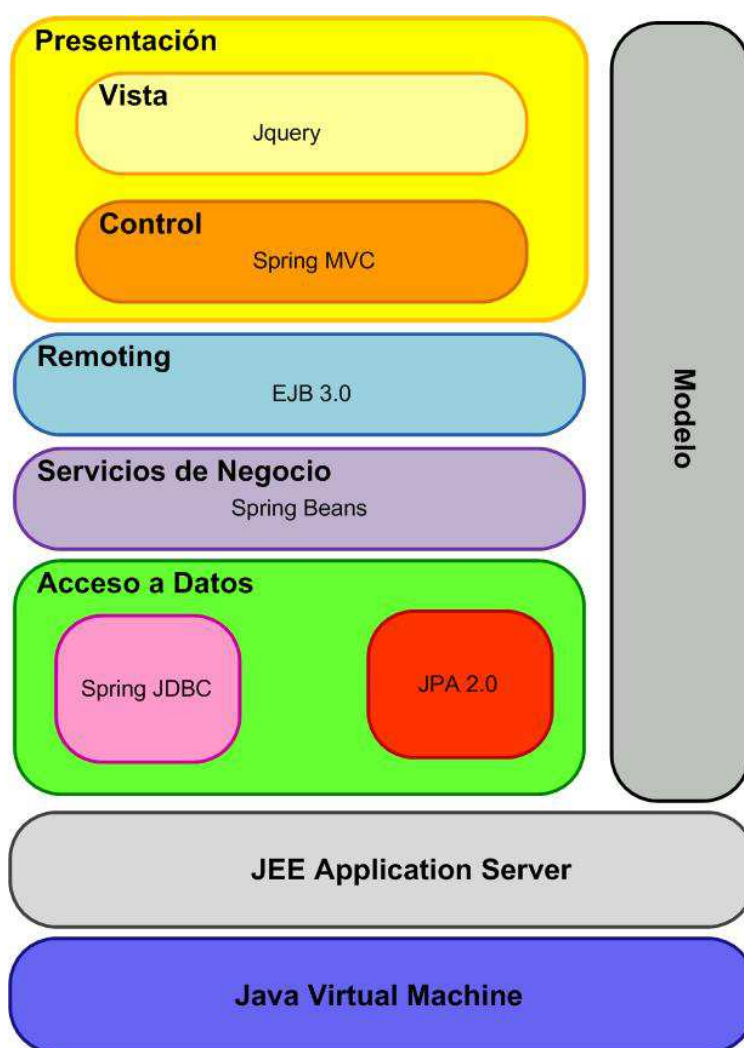


Ilustración 1. Arquitectura UDA.

Además de la arquitectura mostrada en la imagen superior, UDA también proporciona asistentes de generación de proyectos y código. Así, los desarrolladores obtendrán aplicaciones en funcionamiento que cumplen con la arquitectura homologada mediante el simple uso de dichos asistentes, integrados en el entorno de desarrollo Eclipse.

2 Visión Global

2.1 ¿Qué es UDA en realidad?

UDA es un conjunto de utilidades de desarrollo para aplicaciones Java EE y se compone de tres principales afluentes, que son:

1. **Normativa de uso de diferentes Frameworks:** UDA no es un Framework por sí mismo, sino que más bien, es un conglomerado de Frameworks, librerías y productos que han destacado por su éxito en el mundo Java EE. El verdadero valor de UDA reside en definir como encajar dichos componentes para lograr la sinergia que favorezca la simplicidad de las aplicaciones y la robustez de la arquitectura.
2. **Generadores:** UDA proporciona asistentes de generación de proyectos y código, de tal manera que el programador parte de una base que ya funciona y cumple estrictamente, como es de suponer, toda la normativa de uso de los diferentes componentes homologados en UDA.
 - a. **Asistentes de generación de proyectos:** Es posible crear proyectos de dos tipos, según el de motor de persistencia deseado, dando opción a escoger entre JDBC o JPA 2.0 (en el futuro, el menú de opciones podrá crecer). Una vez escogido el tipo de proyecto, UDA generará un proyecto de tipo EAR que estará vinculado a un proyecto JAR y a un proyecto War, obligatoriamente. Opcionalmente, se podrán añadir nuevos módulos War y EJB a esos proyectos iniciales.
Los proyectos EAR generados, están optimizados para ser desplegados en el servidor de aplicaciones Oracle WebLogic 11g automáticamente.
 - b. **Asistentes de generación de código:** Partiendo de un proyecto UDA, se podrá autogenerar el código correspondiente partiendo de un esquema relacional de base de datos. UDA detectará automáticamente si el proyecto sobre el que se desea generar el código es de tipo JDBC o JPA. Basándose en el modelo relacional, generará el código necesario para realizar la gestión CRUD (Create, Read, Update and Delete) de dicho esquema relacional desde una aplicación JEE, con interfaces REST y RMI.
 - c. **Asistentes de generación de vista:** Por último, una vez que se ha autogenerado el backend de la aplicación, UDA proporciona asistentes que generan automáticamente interfaces de usuario Web ricas (Rich Internet Application o RIA) que actúan como consumidores de servicios Web REST, proporcionando al usuario final interfaces atractivas y accesibles desde las que gestionar el esquema relacional cómodamente.
3. **Evolución:** UDA es una apuesta firme que se encuentra en constante evolución y mejora, de manera que con el tiempo, se irán proporcionando nuevas tecnologías, facilidades y funcionalidades a los programadores. Estos últimos pueden participar de dicha evolución, aportando sugerencias y mejoras que progresivamente irán reflejándose en el producto.

2.2 Funcionalidad inicial de una aplicación UDA

En el momento en el que se genera una nueva aplicación UDA, el backend proporciona una colección de funcionalidades que se pueden explotar a través de servicios Web RESTful o invocaciones remotas RMI.

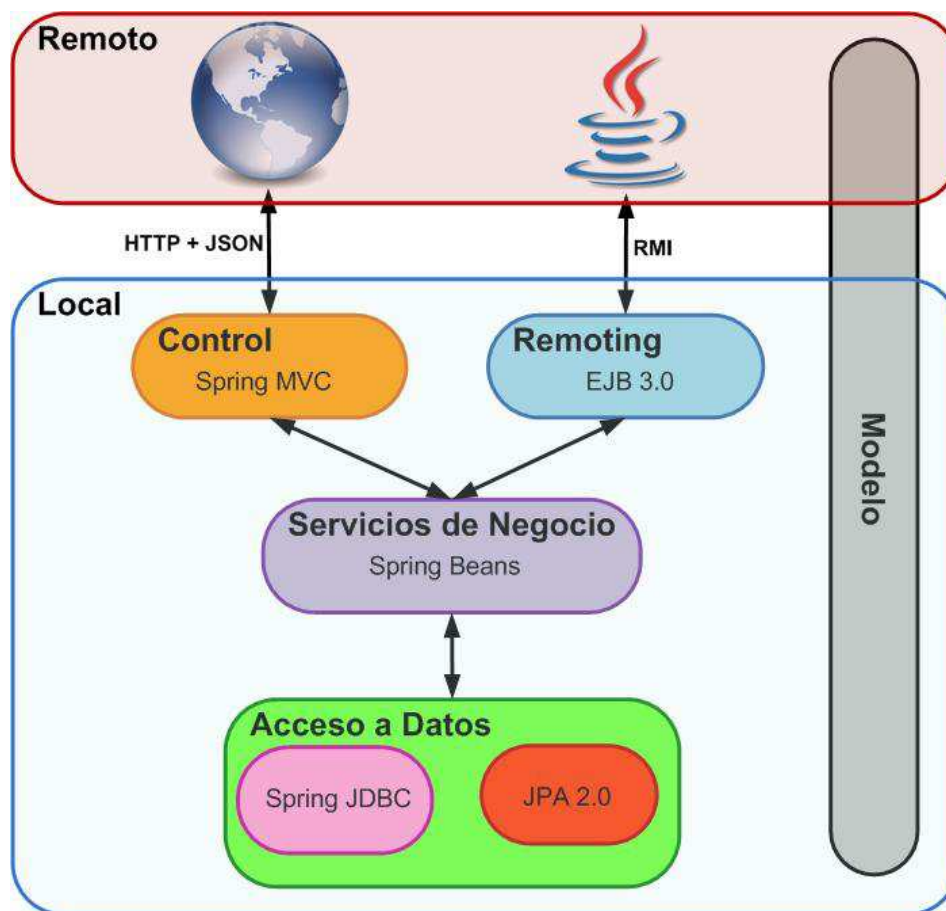


Ilustración 2. Dos maneras de consumir el Backend de UDA.

Tanto los clientes Web, como los clientes RMI son autogenerados por UDA:

- Los clientes ligeros Web que consumen los servicios Web RESTful expuestos por UDA se generan automáticamente en forma de cliente RIA. Para ello, UDA se apoya en la colección de patrones visuales denominado RUP (Rich UDA Patterns). Estos clientes Web se generan con asistentes que proporciona UDA y son altamente parametrizables.

No obstante, los Controladores de UDA (que no son más que Controllers de Spring MVC) pueden ser consumidos desde cualquier cliente Web que pueda comunicarse a través de HTTP (Hypertext Transport Protocol) y JSON (JavaScript Object Notation).

- Los clientes pesados RMI que consumen los EJBs de UDA se generan automáticamente en forma de EJB cliente. Para ello, UDA proporciona asistentes de generación de EJBs (tanto en la parte servidor como en la parte cliente).

A modo de consideración, hay que resaltar que la capa vertical que representa al modelo de datos, sube hasta la capa de cliente remoto (ya sea este de tipo ligero o pesado), lo que quiere decir que al construir los clientes, hay que reutilizar el modelo ya existente en la aplicación UDA que se desea consumir.

2.2.1. Funcionalidad del backend

Actualmente el backend que UDA genera proporciona cierta funcionalidad que sirve para realizar ciertas acciones de gestión sobre la información que se contiene en el modelo relacional de base de datos correspondiente a cada aplicación.

Esta funcionalidad queda totalmente al descubierto en el código generado por UDA y es perfectamente modificable y extensible.

Esta funcionalidad se desglosa en las siguientes acciones concretas:

1. Gestión unitaria de registros de tablas maestras:
 - a. Lectura de un registro en una tabla maestra.
 - b. Inserción de un registro en una tabla maestra.
 - c. Modificación de un registro en una tabla maestra.
 - d. Eliminación de un registro en una tabla maestra.
2. Gestión masiva de registros de tablas maestras:
 - a. Búsqueda de registros por filtrado en una tabla maestra.
 - b. Contador de registros por filtrado en una tabla maestra.
 - c. Eliminación de registros en una tabla maestra.
3. Gestión de registros de tablas maestro-detalle:
 - a. Inserción de un registro de relación entre registros existentes (sólo en relaciones M:N).
 - b. Eliminación de un registro de relación entre registros existentes (sólo en relaciones M:N).
 - c. Búsqueda de registros por filtrado de tablas detalle.
 - d. Contador de registros por filtrado de tablas detalle.

A continuación se detallará la implementación de cada uno de estos casos. Como ejemplo, se muestra el código que UDA autogenera al trabajar con el siguiente esquema entidad-relación:



UDA_Schema.jpg

Antes de profundizar en los casos de uso, a través de los diagramas de secuencia, se muestra es el diagrama de clases implicadas en el elenco de casos de uso que se extraen tomando como referencia la tabla (o entidad) llamada "Vendor".

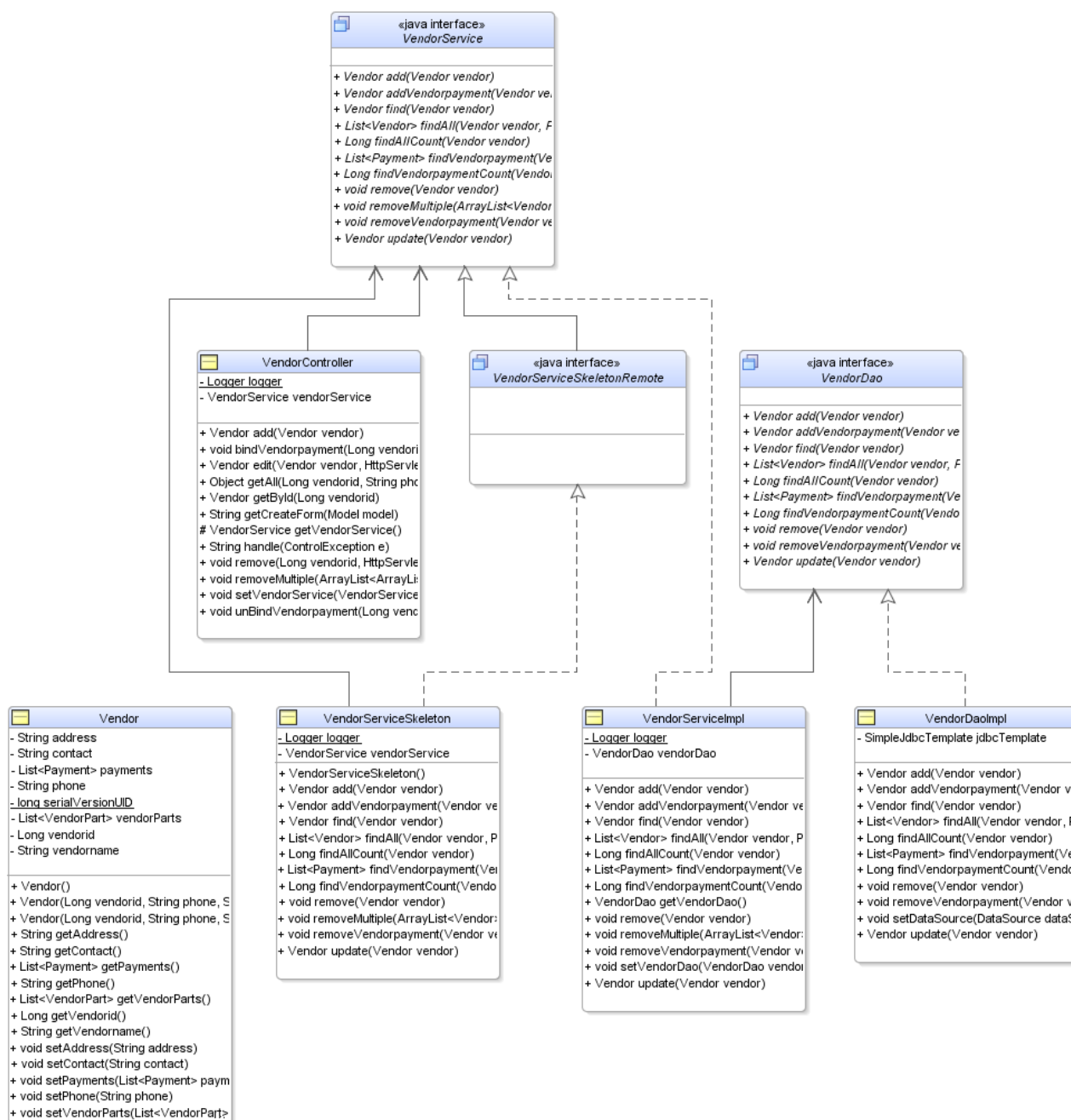


Ilustración 3. Diagrama de clases que gestionan la entidad Vendor.

NOTA: En todos los diagramas de secuencia se obviará la existencia de la capa de acceso a datos (o DAOs) para simplificar. Esta capa tiene un capítulo propio en el documento donde se explica detalladamente su funcionamiento. Se recomienda su lectura, ya que aclara aspectos relativos a la estrategia de gestión de datos.

2.1.1.1 Lectura de un registro en una tabla maestra

La secuencia de invocaciones para la lectura de un registro de la tabla "Vendor", es el siguiente:

- Web: El cliente envía una petición HTTP GET a una URL terminada en “/vendor/1”, donde el valor “1” es una variable que indica la clave primaria del registro que se desea leer. Esto provoca una invocación al componente “VendorController”, pasándole una variable de tipo “Long” con el valor “1”. El “VendorController” construye un objeto de tipo “Vendor” y le asigna el valor “1” en la propiedad “vendorid”.

Acto seguido, invoca al método “find” del componente VendorServiceImpl, al que pasa como parámetro el objeto Vendor.

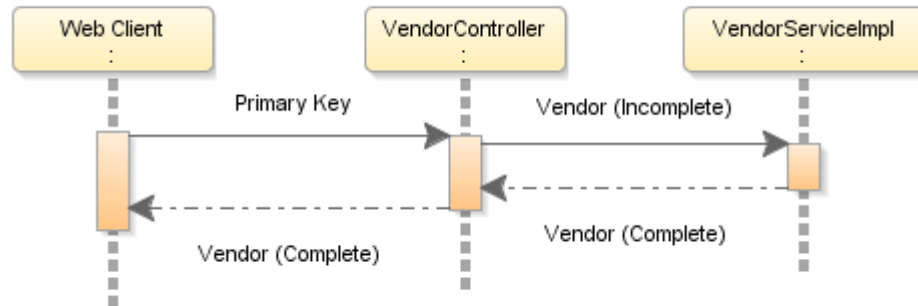


Ilustración 4. Diagrama de secuencia Web para find.

- RMI: El cliente envía una petición RMI al método “find” del EJB “VendorServiceSkeleton” y le pasa como parámetro un objeto de tipo “Vendor” que tiene la propiedad “verdorid”, de tipo “Long”, inicializada a “1”.

Acto seguido, invoca al método “find” del componente VendorServiceImpl, al que pasa como parámetro el objeto Vendor.

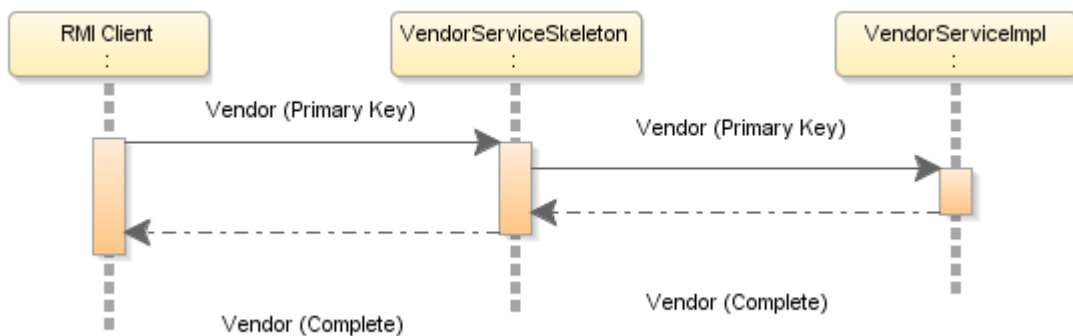


Ilustración 5. Diagrama de secuencia RMI para find.

En ambos casos, el servicio devuelve un objeto de tipo “Vendor” con todos los valores que tiene el registro correspondiente de base de datos.

2.1.1.2 Inserción de un registro en una tabla maestra

La secuencia de invocaciones para la inserción de un registro de la tabla “Vendor”, es el siguiente:

- Web: El cliente envía una petición HTTP POST a una URL terminada en “/vendor”. En el cuerpo de la petición, viajarán todos los campos correspondientes a la clase “Vendor”, en formato JSON. El

“VendorController” construye un objeto de tipo “Vendor” y le asigna todos los valores obtenidos de la petición HTTP.

Acto seguido, invoca al método “add” del componente VendorServiceImpl, al que pasa como parámetro el objeto Vendor.

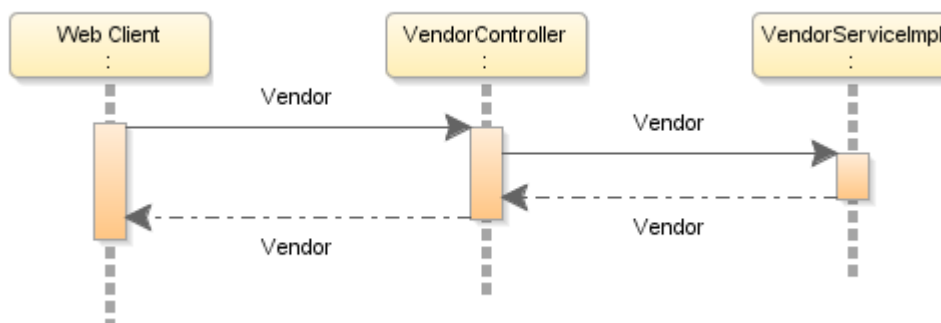


Ilustración 6. Diagrama de secuencia Web para add.

- RMI: El cliente envía una petición RMI método “add” del EJB “VendorServiceSkeleton” y le pasa como parámetro un objeto de tipo “Vendor” con todos los campos necesarios inicializados. Acto seguido, invoca al método “add” del componente VendorServiceImpl, al que pasa como parámetro el objeto Vendor.

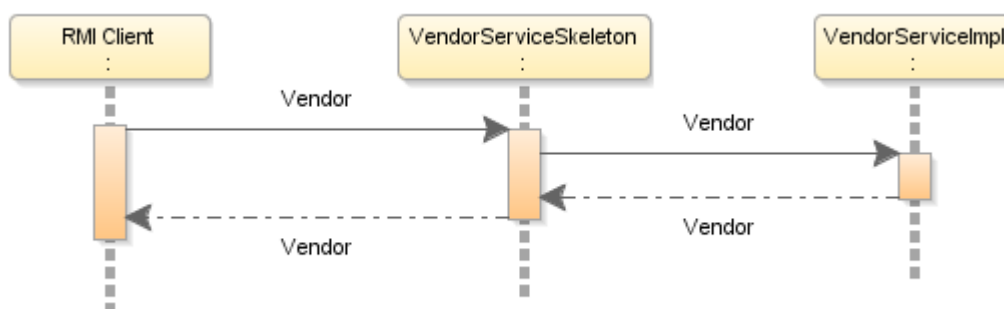


Ilustración 7. Diagrama de secuencia RMI para add.

En ambos casos, el servicio devuelve un objeto de tipo “Vendor” sincronizado con los valores que tiene el registro correspondiente de base de datos.

2.1.1.3 Modificación de un registro en una tabla maestra

La secuencia de invocaciones para la modificación de un registro de la tabla “Vendor”, es el siguiente:

- Web: El cliente envía una petición HTTP PUT a una URL terminada en “/vendor”. En el cuerpo de la petición, viajarán todos los campos correspondientes a la clase “Vendor”, en formato JSON. El “VendorController” construye un objeto de tipo “Vendor” y le asigna todos los valores obtenidos de la petición HTTP.

Acto seguido, invoca al método “update” del componente VendorServiceImpl, al que pasa como parámetro el objeto Vendor.

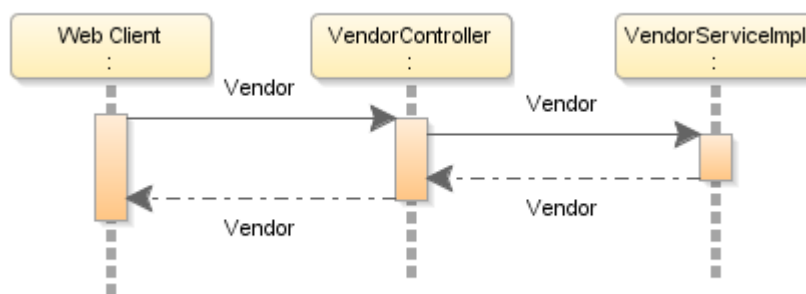


Ilustración 8. Diagrama de secuencia Web para update.

- RMI: El cliente envía una petición RMI al método “update” al EJB “VendorServiceSkeleton” y le pasa como parámetro un objeto de tipo “Vendor” con todos los campos necesarios inicializados. Acto seguido, invoca al método “update” del componente VendorServiceImpl, al que pasa como parámetro el objeto Vendor.

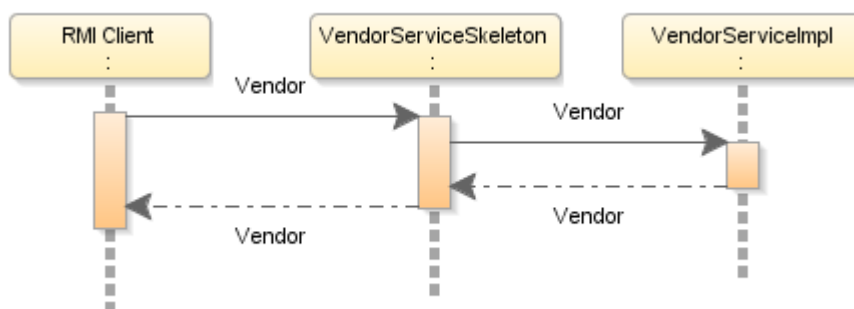


Ilustración 9. Diagrama de secuencia RMI para update.

En ambos casos, el servicio devuelve un objeto de tipo “Vendor” sincronizado con los valores que tiene el registro correspondiente de base de datos.

2.1.1.4 Eliminación de un registro en una tabla maestra

La secuencia de invocaciones para la eliminación de un registro de la tabla “Vendor”, es el siguiente:

- Web: El cliente envía una petición HTTP DELETE a una URL terminada en “/vendor/1”, donde el valor “1” es una variable que indica la clave primaria del registro que se desea eliminar. Esto provoca una invocación al componente “VendorController”, pasándole una variable de tipo “Long” con el valor “1”. El “VendorController” construye un objeto de tipo “Vendor” y le asigna el valor “1” en la propiedad “vendorid”. Acto seguido, invoca al método “remove” del componente VendorServiceImpl, al que pasa como parámetro el objeto Vendor.

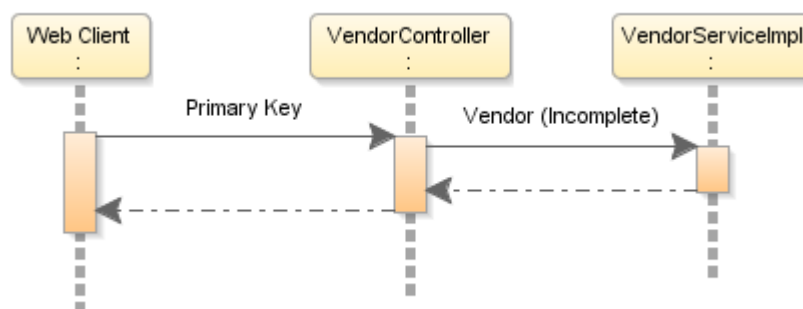


Ilustración 10. Diagrama de secuencia Web para remove.

- RMI: El cliente envía una petición RMI al método “remove” del EJB “VendorServiceSkeleton” y le pasa como parámetro un objeto de tipo “Vendor” que tiene la propiedad “vendorid”, de tipo “Long”, inicializada a “1”.

Acto seguido, invoca al método “remove” del componente VendorServiceImpl, al que pasa como parámetro el objeto Vendor.

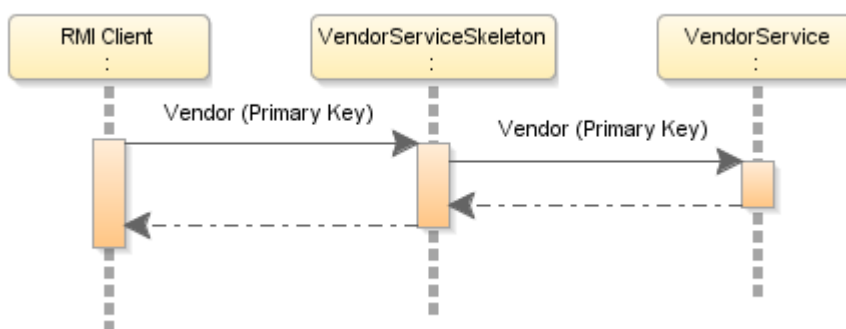


Ilustración 11. Diagrama de secuencia RMI para remove.

En ambos casos, el servicio devuelve “void”.

2.1.1.5 Búsqueda de registros por filtrado en una tabla maestra

En este caso de uso se pretende realizar una búsqueda de registros de una única tabla maestra, donde es posible personalizar la consulta a base de datos mediante ciertos parámetros de filtrado.

Esta búsqueda se realiza a partir de un objeto “Vendor” con las propiedades (todas, algunas o ninguna) inicializadas con ciertos valores. Dicho objeto, se utiliza para aplicar el valor de sus propiedades en el filtrado.

Además, opcionalmente, se podrá personalizar aún más el filtrado, utilizando otro objeto de tipo “Pagination” perteneciente a UDA, a través del cual se indican los siguientes parámetros al filtrado:

- El número máximo de registros a obtener → “rows”.
- El número de página a obtener → “page”. Sabiendo en número de registros y la pagina deseada, se obtienen los registros comprendidos entre $page \cdot rows + 1$ y $page \cdot rows + rows$.
- Ordenación ascendente o descendente → “ascDsc”.
- Campo por el que ordenar → “sort”.

La secuencia de invocaciones para el filtrado de los registros de una tabla es la siguiente:

- Web: El cliente envía una petición HTTP GET a una URL terminada en “/vendor”, donde envía una serie de parámetros relativos al filtrado. Esto provoca una invocación al componente “VendorController”, pasándole como variable los parámetros recibidos en la petición. El “VendorController” construye un objeto “Vendor” donde mapea los parámetros que coinciden con las propiedades de “Vendor”. Además, mapea los parámetros de paginación en un objeto “Pagination”.

Todos estos parámetros son opcionales. Cuantos menos parámetros se reciban, más ambiguo será el filtrado.

A continuación, el “VendorController” invoca al método “findAll” del “VendorServiceImpl” pasándole como parámetro los objetos “Vendor” y “Pagination”.

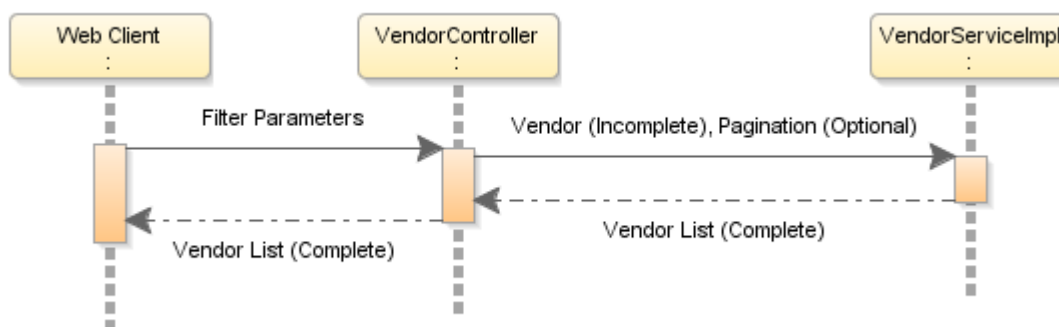


Ilustración 12. Diagrama de secuencia Web para findAll.

- RMI: El cliente envía una petición RMI al método “findAll” del EJB “VendorServiceSkeleton” y le pasa como parámetro un objeto de tipo “Vendor” que tiene las propiedades por las que aplicar el filtro inicializadas. Lo mismo sucede con el objeto “Pagination” que el EJB recibe.

Acto seguido, invoca al método “findAll” del componente VendorServiceImpl, al que pasa como parámetro el objeto “Vendor” y el objeto “Pagination”.

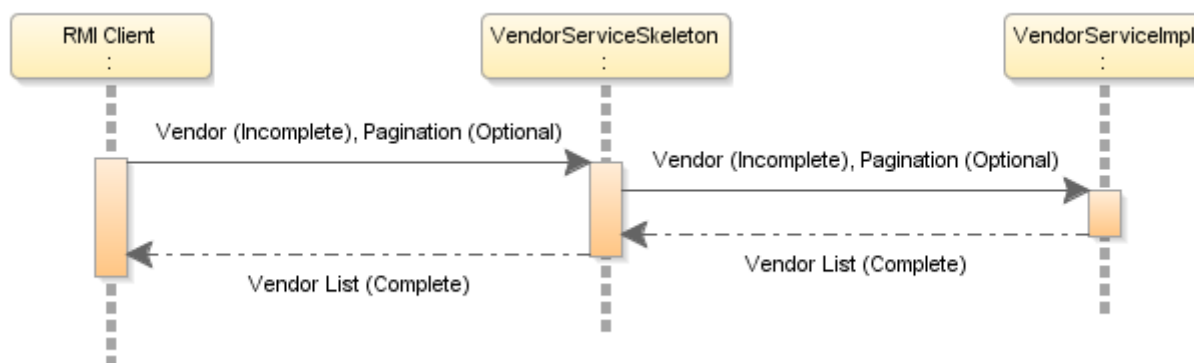


Ilustración 13. Diagrama de secuencia RMI para findAll.

En ambos casos, el servicio devuelve una lista de objetos “Vendor” que coinciden con los parámetros de filtrado.

2.1.1.6 Contador de registros por filtrado en una tabla maestra

En este caso de uso se pretende obtener el número de registros que coinciden con la búsqueda sobre una única tabla maestra, donde es posible personalizar la consulta a base de datos mediante ciertos parámetros de filtrado.

Esta búsqueda se realiza a partir de un objeto “Vendor” con las propiedades (todas, algunas o ninguna) inicializadas con ciertos valores.

La obtención del número de registros es muy útil, por ejemplo, para gestionar la paginación de los resultados.

La secuencia de invocaciones para la obtención del número de registros que responden al filtrado de los registros de una tabla es la siguiente:

- Web: El cliente envía una petición HTTP GET a una URL terminada en “/vendor/count”, donde envía una serie de parámetros relativos al filtrado. Esto provoca una invocación al componente “VendorController”, pasándole como variable los parámetros recibidos en la petición. El “VendorController” construye un objeto “Vendor” donde mapea los parámetros que coinciden con las propiedades de “Vendor”.

Todos estos parámetros son opcionales. Cuantos menos parámetros se reciban, más ambiguo será el filtrado.

A continuación, el “VendorController” invoca al método “findAllCount” del “VendorServiceImpl” pasándole como parámetro el objeto “Vendor”.

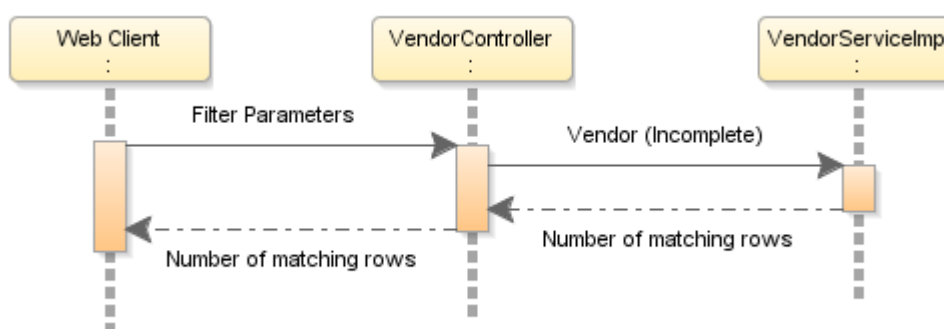


Ilustración 14. Diagrama de secuencia Web para findAllCount.

- RMI: El cliente envía una petición RMI al método “findAllCount” del EJB “VendorServiceSkeleton” y le pasa como parámetro un objeto de tipo “Vendor” que tiene las propiedades por las que aplicar el filtro inicializadas.

A continuación, el Skeleton invoca al método “findAllCount” del componente VendorServiceImpl, al que pasa como parámetro el objeto “Vendor”.

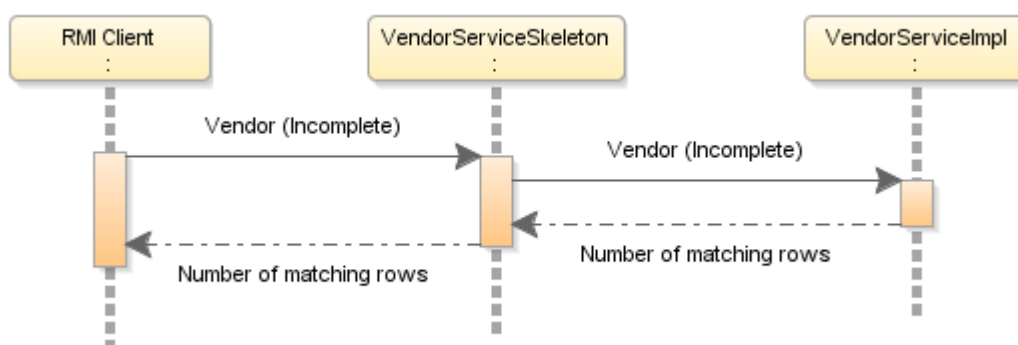


Ilustración 15. Diagrama de secuencia RMI para findAllCount.

En ambos casos, el servicio devuelve un objeto de tipo Long con el número de registros que coinciden con el filtrado realizado.

2.1.1.7 Búsqueda de registros por filtrado por semejanzas en una tabla maestra

En este caso de uso se pretende realizar una búsqueda de registros de una única tabla maestra, donde es posible personalizar la consulta a base de datos mediante ciertos parámetros de filtrado que se aplicaran por semejanza (operador Like).

Esta búsqueda se realiza a partir de un objeto "Vendor" con las propiedades (todas, algunas o ninguna) inicializadas con ciertos valores. Dicho objeto, se utiliza para aplicar el valor de sus propiedades en el filtrado.

Además, opcionalmente, se podrá personalizar aún más el filtrado, utilizando otro objeto de tipo "Pagination" perteneciente a UDA, a través del cual se indican los siguientes parámetros al filtrado:

- El número máximo de registros a obtener → "rows".
- El número de página a obtener → "page". Sabiendo en número de registros y la pagina deseada, se obtienen los registros comprendidos entre $page \times rows + 1$ y $page \times rows + rows$.
- Ordenación ascendente o descendente → "ascDsc".
- Campo por el que ordenar → "sort".

Además, también se utiliza el Flag "startsWith", que indica si la semejanza ha de aplicarse comparando que el valor proporcionado sea contenido por el valor de base de datos desde el inicio, o si por el contrario el valor proporcionado puede ser contenido por el valor en base de datos en cualquier posición.

La secuencia de invocaciones para el filtrado de los registros de una tabla es la siguiente:

- Web: El cliente envía una petición HTTP GET a una URL terminada en "/vendor", donde envía una serie de parámetros relativos al filtrado. Esto provoca una invocación al componente "VendorController", pasándole como variable los parámetros recibidos en la petición. El "VendorController" construye un objeto "Vendor" donde mapea los parámetros que coinciden con las propiedades de "Vendor". Además, mapea los parámetros de paginación en un objeto "Pagination" y mapea el Flag que indica el tipo de filtrado a una variable booleana llamada "startsWith".

Todos estos parámetros son opcionales. Cuantos menos parámetros se reciban, más ambiguo será el filtrado.

A continuación, el "VendorController" invoca al método "findAllLike" del "VendorServiceImpl" pasándole como parámetro los objetos "Vendor", "Pagination" y el booleano "startsWith".

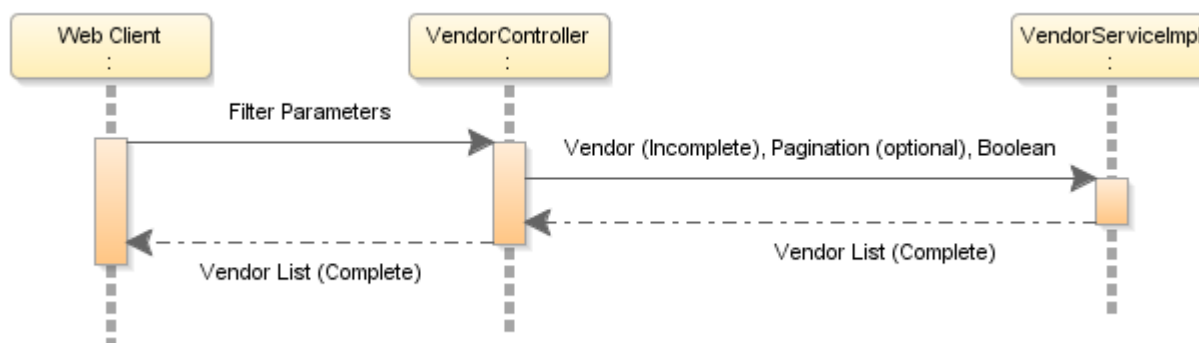


Ilustración 16. Diagrama de secuencia Web para findAllLike.

- RMI: El cliente envía una petición RMI al método “findAllLike” del EJB “VendorServiceSkeleton” y le pasa como parámetro un objeto de tipo “Vendor” que tiene las propiedades por las que aplicar el filtro inicializadas. Lo mismo sucede con los objetos “Pagination” y Boolean que el EJB recibe.

Acto seguido, invoca al método “findAllLike” del componente VendorServiceImpl, al que pasa como parámetro el objeto “Vendor”, el objeto “Pagination” y el Booleano.

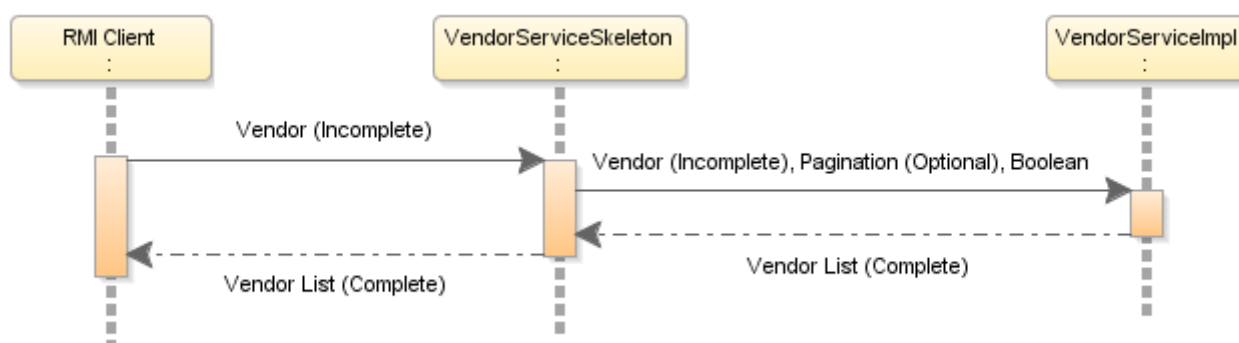


Ilustración 17. Diagrama de secuencia RMI para findAllLike.

En ambos casos, el servicio devuelve una lista de objetos “Vendor” que coinciden de manera similar con los parámetros de filtrado.

2.1.1.8 Eliminación de registros en una tabla maestra

La secuencia de invocaciones para la eliminación de varios registros de la tabla “Vendor”, es la siguiente:

- Web: El cliente envía una petición HTTP POST a una URL terminada en “/vendor/deleteAll”, donde se envía una colección con las claves primarias de todos los registros que se quieren eliminar. Esto provoca una invocación al componente “VendorController”, pasándole como variable una lista de claves primarias, pudiendo ser simples o compuestas. El “VendorController” construye una lista de objetos “Vendor” a los que asigna valores en consonancia con las claves primarias recibidas.

Acto seguido, invoca al método “removeMultiple” del componente VendorServiceImpl, al que pasa como parámetro la lista de objetos “Vendor” con clave primaria asignada.

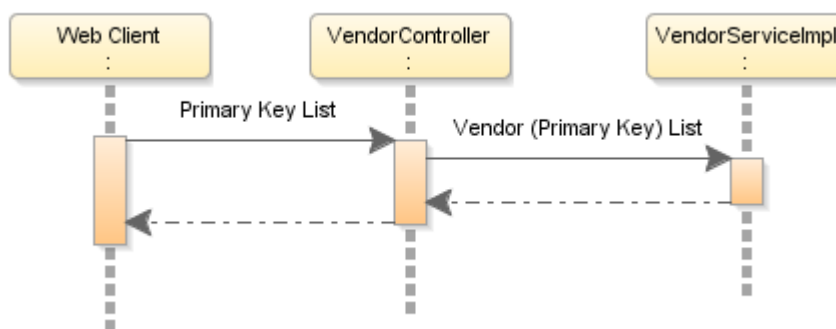


Ilustración 18. Diagrama de secuencia Web para delete multiple.

- RMI: El cliente envía una petición RMI al método “removeMultiple” del EJB “VendorServiceSkeleton” y le pasa como parámetro una lista de objetos de tipo “Vendor” que tienen la propiedad que representa a la clave primaria, es decir “verdorid” inicializada al valor correspondiente.

Acto seguido, invoca al método “removeMultiple” del componente VendorServiceImpl, al que pasa como parámetro la lista de objetos “Vendor”

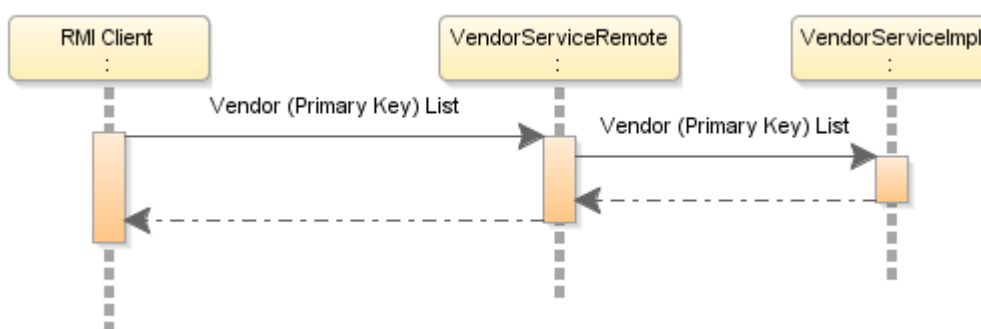


Ilustración 19. Diagrama de secuencia RMI para delete multiple.

En todos los casos, el servicio no devuelve nada.

2.1.1.9 Inserción de un registro de relación entre registros existentes (sólo en relaciones M:N)

Para ilustrar este ejemplo se utilizará la relación existente entre las tablas “Vendor” y “Payment”, ya que al ser esta una relación de tipo Many To Many (o M a N) es necesario hacer uso de una tabla intermedia denominada “Vendor_Payment”, en la cual se establecen relaciones entre registros de la tabla “Vendor” y registros de la tabla “Payment”. Sin embargo, esta tabla intermedia no representa a ninguna entidad, y por lo tanto, no tiene un objeto específico en el modelo de datos, y ha de ser gestionada desde el DAO de alguna de las tablas relacionadas.

En este ejemplo, se hará uso del Servicio y el DAO de la entidad “Vendor”.

Partiendo de que existen ciertos registros en la tablas “Vendor” y “Payment” que se quieren relacionar (es decir, crear nuevos registros en la tabla intermedia que los relacione), esta es la secuencia de pasos que hay que realizar.

- Web: El cliente envía una petición HTTP POST a una URL terminada en “/vendor/bindVendorpayment”, donde se envían las claves primarias de los registros “Vendor” y “Payment” que se desean vincular.

Esto provoca una invocación al método “bindVendorpayment” del “VendorController”, pasándole como parámetro las claves primarias de los objetos “Vendor” y “Payment” a vincular. En este caso, se le pasan un objeto de tipo Long que representa el “vendorid” y un objeto de tipo BigDecimal que representa el “paymentid”.

El “VendorController” crea una instancia del objeto “Vendor” y le asigna el valor correspondiente en la propiedad “vendorid”. Después, hace lo propio con un objeto “Payment” y el valor “paymentid”. Después, accede a la propiedad “payments” del objeto “Vendor” (que es una colección de objetos “Payment”) y añade ahí el objeto “Payment”.

Por último, invoca al método “addVendorpayment” del objeto “VendorServiceImpl” y le pasa el objeto “Vendor”, tal y como muestra el siguiente diagrama.

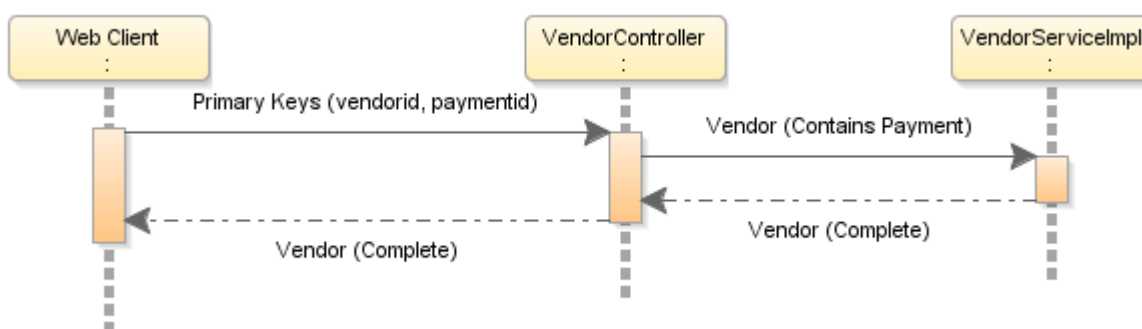


Ilustración 20. Diagrama de secuencia Web para addVendorpayment.

- RMI: El cliente envía una petición RMI al método “addVendorpayment” del EJB “VendorServiceSkeleton” y le pasa como parámetro un objeto “Vendor” que contiene una colección de objetos “Payment”. Todos estos objetos han de tener la primary key establecida.

Después el Skeleton invoca al método “addVendorpayment” del objeto “VendorServiceImpl” y le pasa el objeto “Vendor”, tal y como muestra el siguiente diagrama.

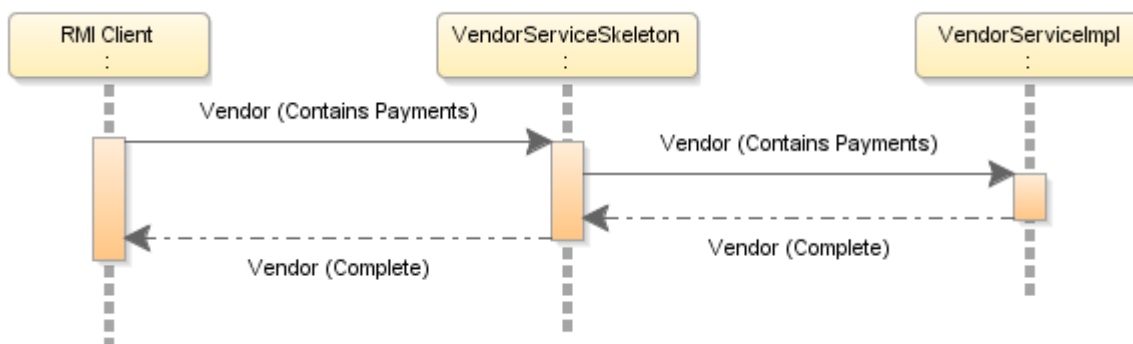


Ilustración 21. Diagrama de secuencia RMI para addVendorpayment.

El servicio siempre devolverá el objeto “Vendor” sincronizado con la base de datos.

2.1.1.10 Eliminación de un registro de relación entre registros existentes (sólo en relaciones M:N)

En este ejemplo se muestra como desvincular registros existentes en las tablas "Vendor" y "Payment" relacionados en la tabla "Vendor_Payment". Esta es la secuencia de acciones que hay que realizar:

- Web: El cliente envía una petición HTTP POST a una URL terminada en "/vendor/unBindVendorpayment", donde se envían las claves primarias de los registros "Vendor" y "Payment" que se desean vincular.

Esto provoca una invocación al método "unBindVendorpayment" del "VendorController", pasándole como parámetro las claves primarias de los objetos "Vendor" y "Payment" a vincular. En este caso, se le pasan un objeto de tipo Long que representa el "vendorid" y un objeto de tipo BigDecimal que representa el "paymentid".

El "VendorController" crea una instancia del objeto "Vendor" y le asigna el valor correspondiente en la propiedad "vendorid". Después, hace lo propio con un objeto "Payment" y el valor "paymentid". Después, accede a la propiedad "payments" del objeto "Vendor" (que es una colección de objetos "Payment") y añade ahí el objeto "Payment".

Por último, invoca al método "removeVendorpayment" del objeto "VendorServiceImpl" y le pasa el objeto "Vendor", tal y como muestra el siguiente diagrama.

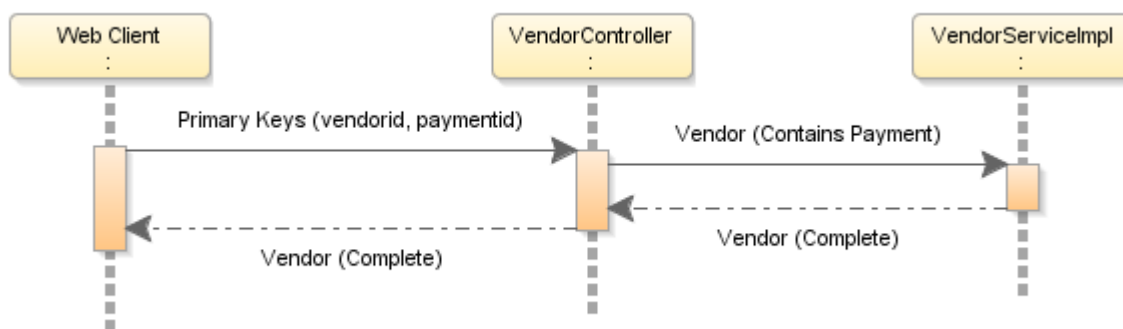


Ilustración 22. Diagrama de secuencia Web para removeVendorpayment.

- RMI: El cliente envía una petición RMI al método "removeVendorpayment" del EJB "VendorServiceSkeleton" y le pasa como parámetro un objeto "Vendor" que contiene una colección de objetos "Payment". Todos estos objetos han de tener la primary key establecida.

Después el Skeleton invoca al método "removeVendorpayment" del objeto "VendorServiceImpl" y le pasa el objeto "Vendor", tal y como muestra el siguiente diagrama.

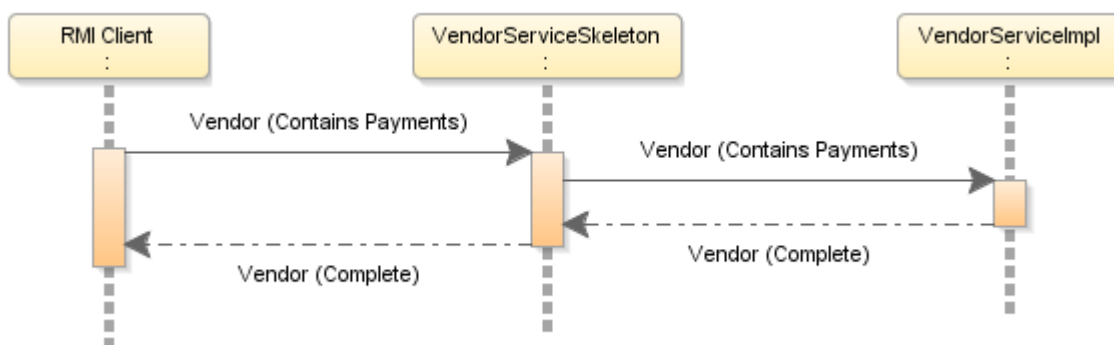


Ilustración 23. Diagrama de secuencia RMI para removeVendorpayment.

El servicio siempre devuelve un objeto Vendor sincronizado con la base de datos.

2.1.1.11 Búsqueda de registros por filtrado de tablas detalle

En este caso se trata de realizar un filtrado basándose en los registros de las tablas relacionadas (o hijas). Este escenario aplica en los casos en los que la relación sea una One To Many (1 a N) o una Many To Many (M a N). Para ello se han de proporcionar, por un lado, la clave primaria de la tabla padre y por otro lado las propiedades de la tabla relacionada por las que se quiere realizar la búsqueda.

De momento, esta funcionalidad solo se ofrece a nivel de servicios de negocio con sus respectivos envoltorios RMI. A nivel Web, los patrones implementados hasta el momento no hacen uso de esta funcionalidad, aunque en cualquier caso queda en manos del desarrollador el utilizar esta funcionalidad que UDA proporciona.

En el caso de RMI, la secuencia es la siguiente:

El cliente envía una petición RMI al método "findVendorpayment" del EJB "VendorServiceSkeleton" y le pasa como parámetro un objeto "Vendor", que contiene la primary key, un objeto "Payment" que contiene los parámetros de filtrado y un objeto "Pagination" donde se añaden parámetros de filtrado extra, como por ejemplo, el número de registros a obtener (el propósito de este objeto ya se ha explicado anteriormente).

Después el Skeleton invoca al método "findVendorpayment" del objeto "VendorServiceImpl" y le pasa el objeto "Vendor", el objeto "Payment" y el objeto "Pagination", tal y como muestra el siguiente diagrama.

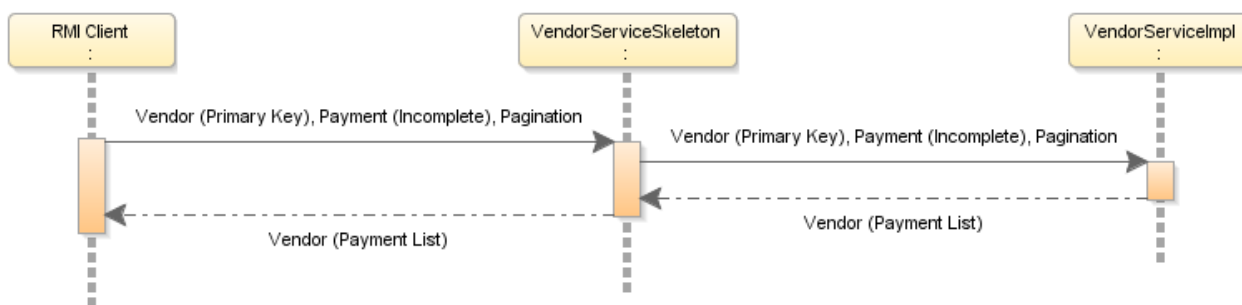


Ilustración 24. Diagrama de secuencia RMI para findVendorpayment.

El servicio devuelve la entidad "Vendor" con todas las entidades "Payment" que coinciden con el filtro vinculadas en una lista interna.

2.1.1.12 Contador de registros por filtrado de tablas detalle

En este caso se trata de obtener el número de registros que obedecen a un filtrado basándose en los registros de las tablas relacionadas (o hijas). Este escenario aplica en los casos en los que la relación sea una One To Many (1 a N) o una Many To Many (M a N). Para ello se han de proporcionar, por un lado, la clave primaria de la tabla padre y por otro lado las propiedades de la tabla relacionada por las que se quiere realizar la búsqueda.

De momento, esta funcionalidad solo se ofrece a nivel de servicios de negocio con sus respectivos envoltorios RMI. A nivel Web, los patrones implementados hasta el momento no hacen uso de esta funcionalidad, aunque en cualquier caso queda en manos del desarrollador el utilizar esta funcionalidad que UDA proporciona.

En el caso de RMI, la secuencia es la siguiente:

El cliente envía una petición RMI al método “findVendorpaymentCount” del EJB “VendorServiceSkeleton” y le pasa como parámetro un objeto “Vendor”, que contiene la primary key y un objeto “Payment” que contiene los parámetros de filtrado.

Después el Skeleton invoca al método “findVendorpaymentCount” del objeto “VendorServiceImpl” y le pasa el objeto “Vendor” y el objeto “Payment”, tal y como muestra el siguiente diagrama.

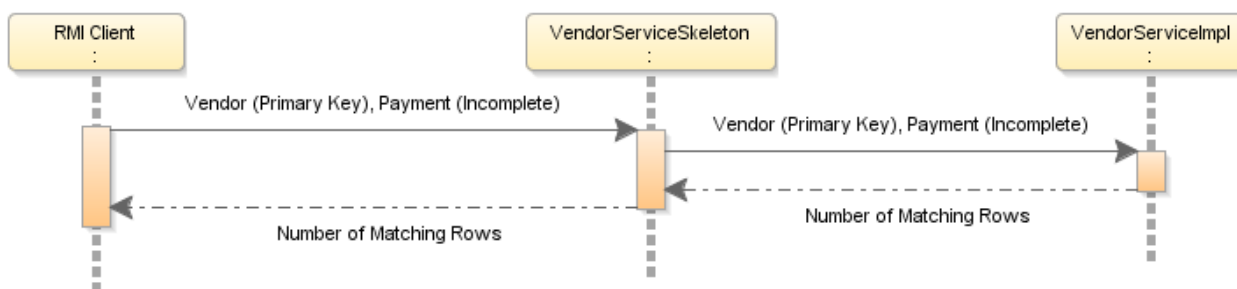


Ilustración 25. Diagrama de secuencia RMI para findVendorpaymentCount.

El servicio devuelve el número de registros que casan con el filtrado realizado.

2.3 Arquitectura de Capas

El diseño de UDA está basado en una arquitectura de capas: presentación, remoting, servicios de negocio y acceso a datos. La ventaja principal del desarrollo en capas consiste en poder distribuir el trabajo de creación de una aplicación por niveles, de este modo, cada grupo de trabajo está totalmente abstraído del resto, de manera que cada grupo sólo necesita conocer la API que existe en su nivel anterior, y los futuros cambios, en la mayoría de casos, sólo afectan al nivel requerido.

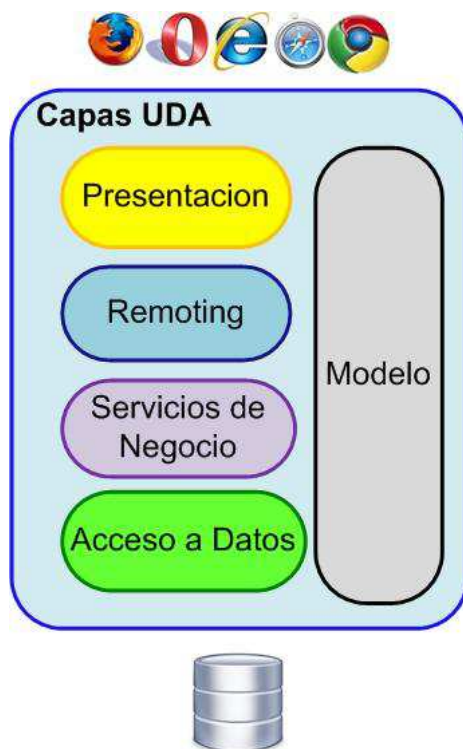


Ilustración 26. Arquitectura de capas.

Las capas que componen una aplicación UDA son las siguientes:

- **Presentación:** Es la responsable de ofrecer a los usuarios la posibilidad de interactuar con la aplicación de forma sencilla e intuitiva, proporcionando una experiencia de usuario Web satisfactoria.

La capacidad de desacoplar una interfaz de usuario y reemplazarla con otra suele ser un requerimiento clave en todas las capas de presentación.

A grandes rasgos se podría decir que la capa de presentación se compone de:

1. La interfaz de usuario: ofrece a los usuarios información, sugerencias, acciones y captura los datos de entrada a través del teclado y el ratón.
 2. La lógica de presentación: se ejecuta en cliente y hace referencia a todo el procesamiento requerido para mostrar datos y transformar los datos de entrada en acciones que podemos ejecutar contra el servidor.
 3. La lógica de control de peticiones: reside en el servidor y se encarga de interactuar con lógica de presentación, encargándose principalmente de transformar el modelo de datos para la recepción/envío, gestionar las reglas de navegación y enlazar con la capa de servicios de negocio.
- **Servicios de negocio:** En esta capa es donde se deben implementar todas aquellas reglas obtenidas a partir del análisis funcional del proyecto. Un servicio se compone de funciones sin estado, auto-contenidas, que aceptan una(s) llamada(s) y devuelven una(s) respuesta(s) mediante una interfaz bien definida. Los servicios no dependen del estado de otras funciones o procesos. Los servicios pueden también enmarcarse en unidades discretas de trabajo, como son las transacciones, que coordinan el uso de la capa de acceso a datos.
 - **Acceso a datos:** La capa de acceso a datos es la responsable de la gestión de la persistencia de la información manejada en las capas superiores. En un modelo académicamente purista, la interfaz de esta capa estaría compuesta por vistas de las entidades a persistir (una vista de “factura”, otra de “cliente”), pero a efectos prácticos, y con objeto de aprovechar la habitual potencia de los gestores de bases de datos, la interfaz muestra una serie de métodos que pueden agrupar operaciones en lo que se puede denominar “lógica de persistencia”, como insertar cliente o

inserción factura, en la que podrían darse de alta al mismo tiempo una factura y todos las entidades que dependan de dicha factura (porque no, el mismo cliente).

- Remoting: Su propósito será el de permitir ofrecer un acceso a los servicios de negocio debidamente empaquetados a aplicaciones diferentes a la que contiene dichos servicios. Para ello, se utilizan técnicas de Remoting o invocación remota a Objetos. En general en las aplicaciones que no necesiten accesos remotos esta capa no es necesaria. Por esto último se deduce que la capa de Remoting **no puede contener lógica de negocio** y ha de actuar como Proxy (consultar patrón Proxy más adelante) de la capa de servicio.
- Modelo: representa las entidades de la aplicación y típicamente es utilizada por las capas formando parte de los parámetros de sus interfaces a modo de Value Objects.

Las funcionalidades de Transaccionalidad declarativa y Trazabilidad para auditoria (Logging) son proporcionadas utilizando Programación mediante Aspectos (AOP).

La seguridad ha sido implementada mediante Spring Security, y para la validación se ha implementado un sistema que es utilizado por la vista, aunque para el resto de capas de utiliza directamente el estándar de Bean Validation proporcionado por Hibernate Validator.

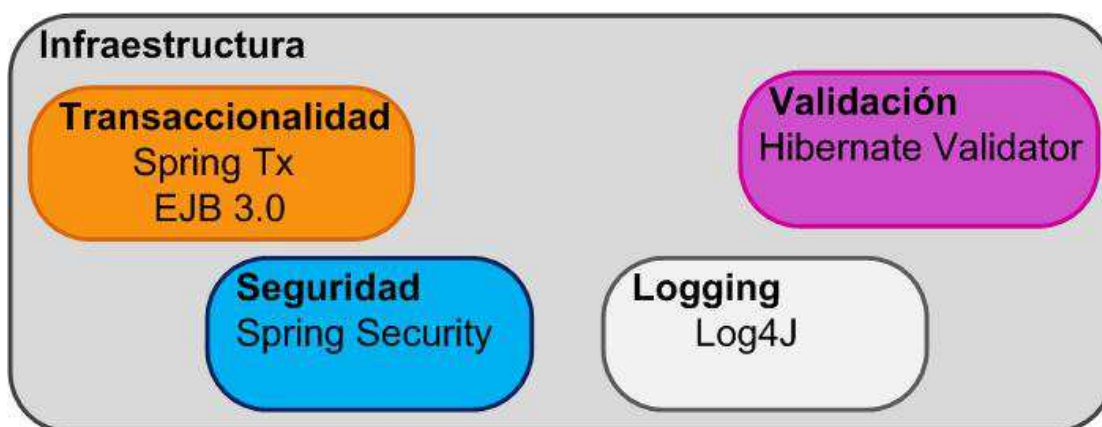


Ilustración 27. Componentes de Infraestructura.

Cada uno de estos módulos que proporciona la infraestructura son implementados utilizando tecnologías no propietarias. Esta es una de las **premisas de las nuevas utilidades de desarrollo**: se escogen las que se consideran las mejores tecnologías para cada caso en lugar de hacer desarrollos a medida siempre que sea posible.

Sin embargo, es evidente que en algunos casos existe la obligación de hacer desarrollos a medida. Principalmente en los casos de integración con otros sistemas que ya existen en la infraestructura donde se vaya a utilizar UDA, como es el sistema de seguridad corporativa del Gobierno Vasco (XL-NetS) o la escritura de Logs con formatos específicos. Por tanto, existen ciertos desarrollos propietarios puntuales y basados en estándares y/o estándares de facto, que se resumen a continuación:

- Seguridad: Se proporciona un Wrapper de XLNets en la librería x38ShLibClasses. Se trata de una implementación basada en Spring Security. El uso de dicho Wrapper solo es obligatorio en el ámbito de EJIE/EJGV.
- Validación: La arquitectura se apoya en Hibernate Validator, implementación de referencia de JSR-303 y parte de JEE6, para proporcionar un sistema de validación estándar multicapa. Sin embargo, se ha desarrollado una solución que posibilita la validación automática de campos y formularios pertenecientes a la capa de vista, para así, conseguir doble objetivo de, por un lado, las reglas de validación residan exclusivamente en el modelo de datos, y por otro lado, la vista pueda utilizar dichas validaciones mediante peticiones asíncronas Ajax.
- Logging: Los desarrolladores tendrán la posibilidad de escribir trazas que les puedan ser de utilidad. Para ello, se dispone directamente del Logger de SLF4J. No obstante, existen una serie de trazas que la arquitectura se encarga de generar automáticamente, utilizando interceptores AOP que captan y

registran las peticiones realizadas a Filtros, Servicios y Objetos de Acceso a datos (Data Access Objects en inglés, **DAOs en adelante**).

De esa manera, se proporciona la capacidad de seguir la traza de la ejecución de las diferentes operaciones de manera automática, ya que cada hilo de ejecución dispondrá de un identificador único, asignado automáticamente por la arquitectura, que se verá reflejado en las trazas, dando lugar a la posterior trazabilidad de las transacciones, ya que dicho identificador se propaga en peticiones remotas.

En cuanto a la transaccionalidad de las operaciones, se delega todo el peso de este aspecto opcional sobre la implementación de Java Transaction API que proporciona el servidor de aplicaciones (como por ejemplo, Weblogic 11g). Tanto Spring como los EJBs hacen uso de dicha transaccionalidad proporcionada por el contenedor, de manera declarativa, utilizando exclusivamente anotaciones.

La siguiente tabla refleja las tecnologías más destacables sobre las que se basa UDA:

| Tecnología | Utilización en UDA | Versión |
|-----------------------|---|---------|
| JQuery | Componentes visuales, lógica de presentación y comunicaciones Ajax. | 1.7.2 |
| Spring MVC | Controlador y gestión de navegación | 3.1.1 |
| Jackson | Procesador de JSON | 1.9.7 |
| Apache Tiles | Plantillado de páginas dinámicas | 2.2.2 |
| Spring Framework | Inversion of Control, Dependency Injection, AOP,...→ Núcleo de UDA | 3.1.1 |
| Spring JDBC | Utilidades para facilitar el uso de JDBC | 3.1.1 |
| JPA | Mapeo de objetos y base de datos. | 2.0 |
| Eclipselink | Implementación JPA | 2.1.2 |
| Slf4j | Escritura de logs y trazas | 1.6.1 |
| Logback | Escritura de logs y trazas. | 1.0.6 |
| Hibernate Validator | Validaciones de campos anotados directamente en el modelo. | 4.2 |
| Spring Security | Seguridad declarativa integrada con Spring | 3.1.0 |
| Enterprise Java Beans | Invocaciones remotas transaccionales | 3.0 |

2.4 Estructura de proyectos

Los proyectos que componen las aplicaciones UDA se dividen en diferentes grupos físicos de ficheros o proyectos Eclipse.

Toda aplicación ha de estar compuesta de la siguiente manera (“xxx” representa al código de aplicación) :

- Proyecto EAR, con la nomenclatura xxxEAR → Contiene librerías, Wars y EJBs.
 - Proyecto xxxzzzWar → Contiene las clases de la capa de control y JSPs. Puede haber uno o varios Wars en un EAR.
 - Proyecto xxxuuuEJB → Contiene los EJBs de la capa de Remoting. Es opcional, pudiendo haber de ninguno a varios dentro del mismo EAR.
 - Proyecto xxxEARClasses → Contiene las clases de la capa de servicios de negocio, acceso a datos y modelo. Tiene que haber uno por aplicación.
- Proyecto xxxConfig → Contiene la configuración de la aplicación y de LogBack. Tiene que haber uno por proyecto.
- Proyecto xxxStatics → Contiene el contenido estático Web de la aplicación. Tiene que haber uno por proyecto. Estos proyectos no se despliegan en servidores de aplicaciones, sino en servidores Web como Apache o Interwoven.

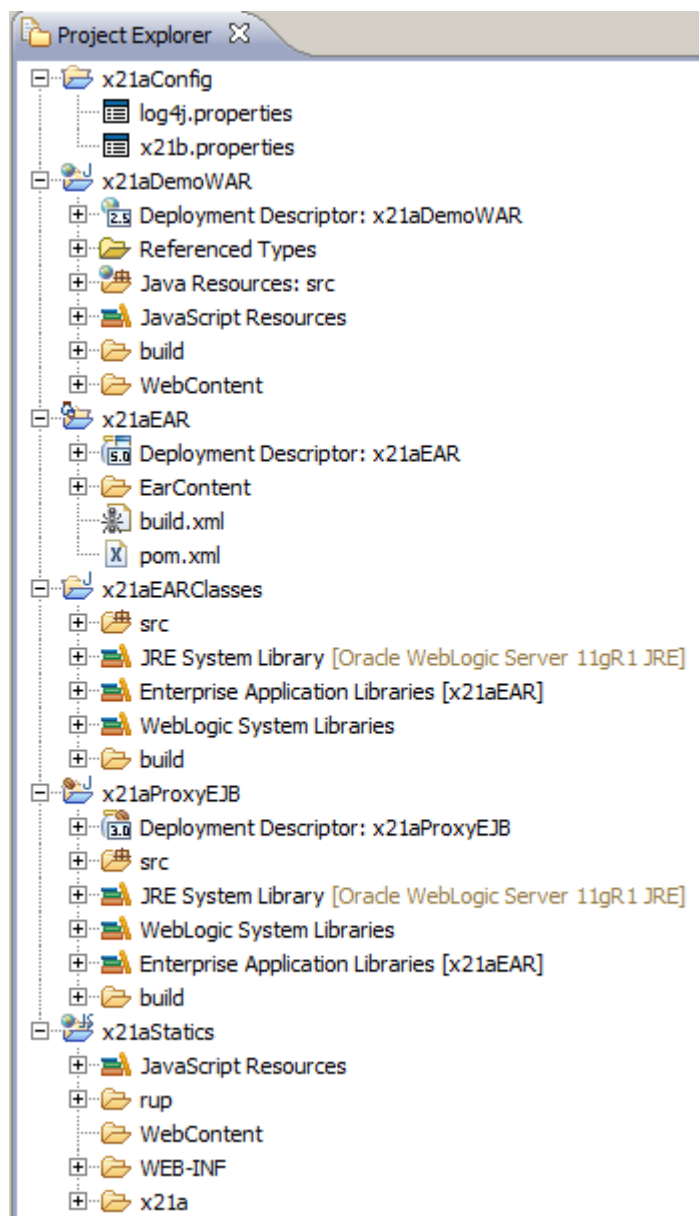


Ilustración 28. Proyectos de una aplicación UDA.

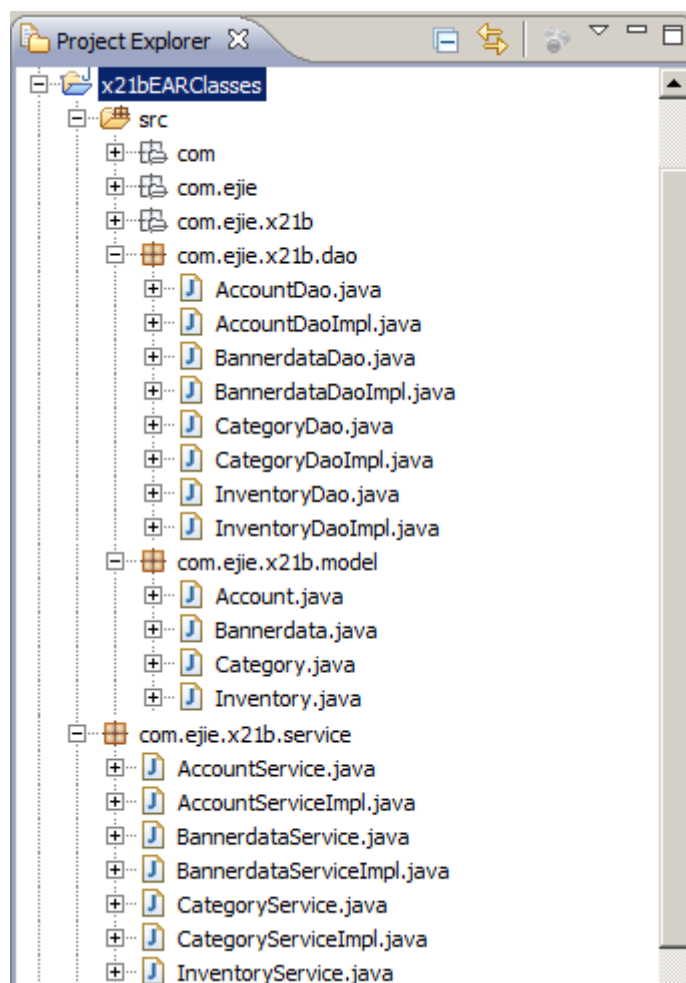
2.5 Nomenclatura y paquetería

La siguiente tabla recoge la nomenclatura, de estricto cumplimiento, de paquetes y clases acordada para los elementos de las diferentes capas.

| Capa | Nomenclatura de paquete |
|------------|--|
| Controller | <code>com.ejje.codigoAplicacion.control.[agrupacionOpcional].</code> |

| | |
|-------------|---|
| | NombredebeanBean.java |
| Data Access | Interfaces: com.ejje.codigoAplicacion.dao.[agrupaciónOpcional]. NombreInterfaceDAO |
| | Implementaciones: com.ejje.codigoAplicacion.dao.[agrupaciónOpcional]. NombreInterfaceDAOImpl |
| Service | Interfaces: com.ejje. codigoAplicacion.service.[grupoFuncionalOpcional]. NombreInterfaceService |
| | Implementaciones: com.ejje. codigoAplicacion.service.[grupoFuncionalOpcional]. NombreInterfaceServiceImpl |
| Model | com.ejje.codigoAplicacion.model.[agrupacionOpcional]. NombreClase |

Un ejemplo de uso de esta nomenclatura puede observarse en la imagen siguiente:



Adicionalmente, habrá que crear un subpaquete bajo el paquete “model” cuando se trate de aplicaciones JPA (en JDBC no existe tal subpaquete). Este subpaquete obedece a la siguiente normativa:

| | |
|-----------------------------|---|
| Data Transfer Objects | <code>com.ejje.codigoAplicacion.model.[agrupacionOpcional].dto. NombreClaseDto</code> |
|-----------------------------|---|

Este es un ejemplo de nomenclatura del paquete model al completo, en aplicaciones JPA:

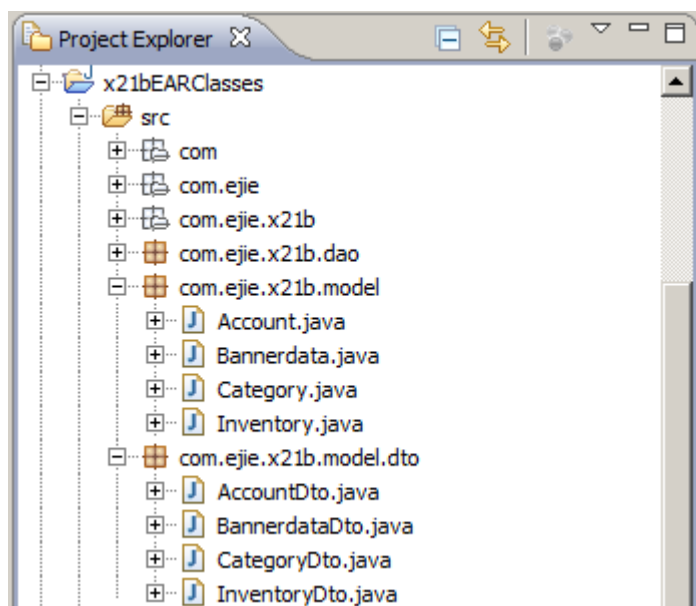


Ilustración 29. Ejemplo de nomenclatura en DTOs.

En los casos en los que la aplicación realice invocaciones remotas, deberá disponer de un EJB 3.0 cliente (Stub) por cada EJB 3.0 servidor (Skeleton) que se desee invocar. A continuación, la normativa de nomenclatura tanto para Stubs como para Skeletons:

| | |
|----------|--|
| Remoting | Interfaces Skeleton: com.ejje. codigoAplicacion.remoting.[grupoFuncionalOpcional]. NombreInterfaceServiceSkeletonRemote |
| | Implementaciones Skeleton: com.ejje. codigoAplicacion.remoting.[grupoFuncionalOpcional]. NombreInterfaceServiceSkeleton |
| | Interfaces Stub: com.ejje. codigoAplicacion.remoting.[grupoFuncionalOpcional]. NombreInterfaceServiceStubRemote |
| | Implementaciones Stub: com.ejje. codigoAplicacion.remoting.[grupoFuncionalOpcional]. NombreInterfaceServiceStub |

A continuación sendos ejemplos de Skeletons y Stubs:

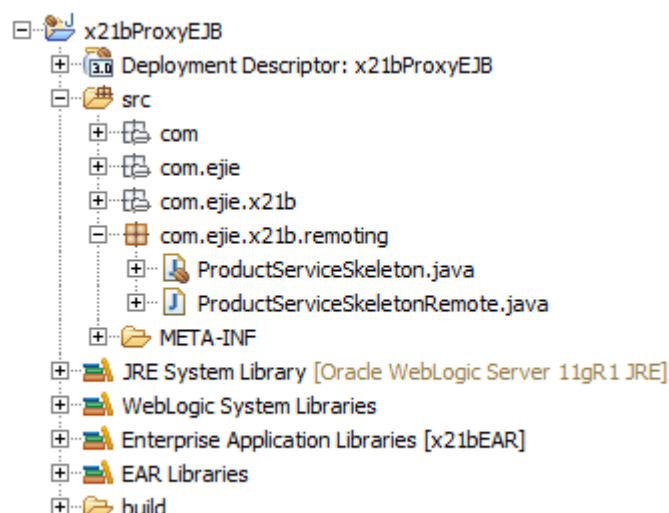


Ilustración 30. Ejemplo de nomenclatura de Skeletons.

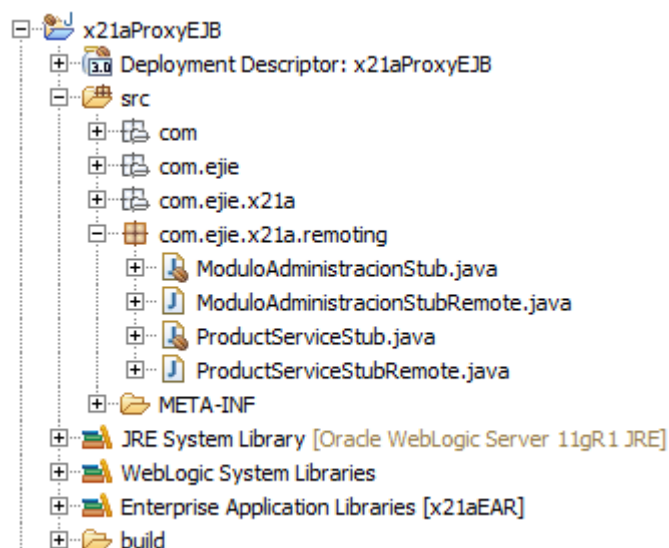


Ilustración 31. Ejemplo de nomenclatura de Stubs.

Leyenda:

Negrita: indica que ha de escribirse tal cuál.

codigoAplicacion: sustituir por el código real de la aplicación. La primera letra será minúscula cuando se utilice para dar nombre al paquete y mayúscula cuando se use para nombrar clases.

[]: Indican opcionalidad. Se emplea para en caso de existir numerosas clases, poder hacer subgrupos. Se trata únicamente de una cuestión organizativa y de legibilidad.

NombreClase o NombreInterfaz: son de escritura libre. Son los que realmente identifican a la clase.

2.6 Empaquetado de librerías

UDA utiliza muchas librerías de terceros. En la siguiente tabla se listan todas ellas:

| Librerías UDA |
|--|
| aopalliance-1.0.jar |
| aspectjrt-1.6.8.jar |
| aspectjweaver-1.6.9.jar |
| commons-beanutils-1.8.0.jar |
| commons-digester-2.0.jar |
| eclipselink-2.1.3.jar (*) |
| hibernate-validator-4.2.0.Final.jar |
| jackson-core-asl-1.9.7.jar |
| jackson-mapper-asl-1.9.7.jar |
| javax.persistence-2.0.1.jar (*) |
| jcl-over-slf4j-1.6.1.jar |
| joda-time-1.6.jar |
| jstl-1.2.jar |
| logback-classic-1.0.6.jar |
| logback-core-1.0.6.jar |
| slf4j-api-1.6.1.jar |
| slf4j-ext-1.6.1.jar |
| spring-aop-3.1.1.RELEASE.jar |
| spring-asm-3.1.1.RELEASE.jar |
| spring-beans-3.1.1.RELEASE.jar |
| spring-context-3.1.1.RELEASE.jar |
| spring-context-support-3.1.1.RELEASE.jar |
| spring-core-3.1.1.RELEASE.jar |

| Librerías UDA |
|---|
| spring-expression-3.1.1.RELEASE.jar |
| spring-jdbc-3.1.1.RELEASE.jar |
| spring-orm-3.1.1.RELEASE.jar |
| spring-security-acl-3.1.0.RELEASE.jar |
| spring-security-config-3.1.0.RELEASE.jar |
| spring-security-core-3.1.0.RELEASE.jar |
| spring-security-taglibs-3.1.0.RELEASE.jar |
| spring-security-web-3.1.0.RELEASE.jar |
| spring-tx-3.1.1.RELEASE.jar |
| spring-web-3.1.1.RELEASE.jar |
| spring-webmvc-3.1.1.RELEASE.jar |
| tiles-api-2.2.2.jar |
| tiles-core-2.2.2.jar |
| tiles-jsp-2.2.2.jar |
| tiles-servlet-2.2.2.jar |
| tiles-template-2.2.2.jar |
| validation-api-1.0.0.GA.jar |
| xercesImpl-2.9.1.jar |

(*): Librerías solo requeridas en aplicaciones JPA.

A la hora de desarrollar con UDA, estas librerías, incluyendo la propia de UDA x38ShLibClasses deben localizarse dentro de la carpeta APP-INF/lib de la aplicación EAR correspondiente.

3 Presentación

Antes de profundizar en detalles sobre la implementación de la capa de presentación de UDA, conviene hacer un breve repaso de lo que representa este grupo de componentes de la arquitectura, desde el punto de vista del paradigma MVC (Model View Control):

- **Modelo:** Esta es la representación específica de la información con la cual el sistema opera. En resumen, el modelo se limita a lo relativo de la vista y su controlador facilitando las presentaciones visuales complejas. El sistema también puede operar con más datos no relativos a la presentación, haciendo uso integrado de otras lógicas de negocio y de datos afines con el sistema modelado.
- **Vista:** Este presenta el modelo en un formato adecuado para interactuar, usualmente la interfaz de usuario.
- **Controlador:** Este responde a eventos, usualmente acciones del usuario, e invoca peticiones al modelo y, probablemente, a la vista.

Muchos de los sistemas informáticos utilizan un Sistema de Gestión de Base de Datos para gestionar los datos: en líneas generales del **MVC** corresponde al modelo. La unión entre capa de presentación y capa de negocio conocido en el paradigma de la Programación por capas representaría la integración entre **Vista** y su correspondiente **Controlador** de eventos y acceso a datos, MVC no pretende discriminar entre capa de negocio y capa de presentación pero si pretende separar la capa visual gráfica de su correspondiente programación y acceso a datos, algo que mejora el desarrollo y mantenimiento de la Vista y el Controlador en paralelo, ya que ambos cumplen ciclos de vida muy distintos entre sí.

Aunque se pueden encontrar diferentes implementaciones de **MVC**, el flujo que sigue el control en UDA es el siguiente:

1. El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, el usuario pulsa un botón, enlace, etc.)
2. El controlador recibe (mediante peticiones Ajax) la notificación de la acción solicitada por el usuario. El controlador gestiona el evento que llega, frecuentemente a través de un gestor de eventos (handler) o callback.
3. El controlador accede al modelo, actualizándolo, posiblemente modificándolo de forma adecuada a la acción solicitada por el usuario (por ejemplo, el controlador actualiza el carro de la compra del usuario). Los controladores complejos están a menudo estructurados usando un patrón de comando que encapsula las acciones y simplifica su extensión.
4. El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario. La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se reflejan los cambios en el modelo (por ejemplo, produce un listado del contenido del carro de la compra). El modelo no debe tener conocimiento directo sobre la vista. La vista no tiene acceso directo al modelo, dejando que el controlador envíe los datos del modelo a la vista.
5. La interfaz de usuario espera nuevas interacciones del usuario, comenzando el ciclo nuevamente.

3.1 Vista

En la capa de vista se optado por seguir las tecnologías del proyecto anterior de RIA jQuery y la filosofía de creación de plugins.

A continuación se muestra el catálogo de patrones desarrollados hasta la fecha, explicado de una forma funcional, si se desea mas información sobre su implementación véase los manuales de cada patrón de RUP.

- Patrón **Idioma**: permite al usuario poder elegir entre los diferentes idiomas de los que dispone la aplicación.

A continuación se muestra una maqueta típica del patrón:



Los casos de uso del patrón son:

Cuando tengamos varios idiomas disponibles en el sitio web y queramos que los usuarios puedan elegir fácilmente el que prefieran.

Por qué.

- Ocupa poco espacio en la interfaz.
- Es fácil de comprender y utilizar.
- Es escalable: se pueden incluir varios idiomas en el combo desplegable sin afectar al resto de la interfaz.
- Patrón **ToolBar**: barra de herramientas, donde agrupar las funcionalidades de la entidad con la que se está trabajando. Esta barra de botones agrupará las acciones principales de la pantalla, tales como Nuevo Elemento, Editar, Borrar, Filtrar, Imprimir ... haciendo uso de botones fijo o de agrupación de botones (patrón mbuttons) como se puede ver en la imagen inferior.

A continuación se muestra un ejemplo del patrón:



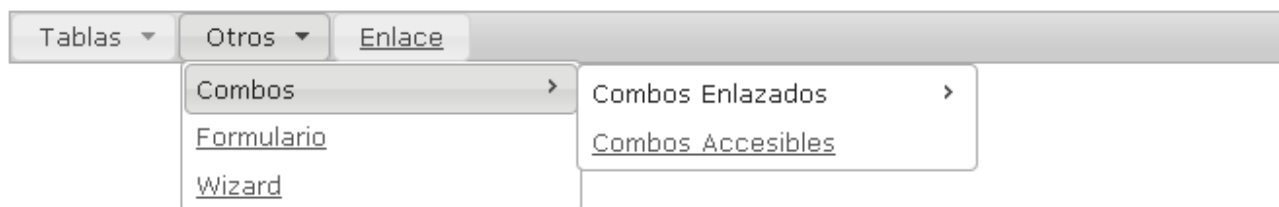
- Patrón **Menú**: este patrón muestra al usuario el menú principal de la aplicación y se mantiene a lo largo de todas las páginas de forma consistente. Muestra entradas directas a secciones clave de la aplicación, los puntos fuertes de la aplicación. Este patrón dispone de dos formas de mostrar dicho menú, horizontal o vertical.

Se recomienda el uso del patrón:

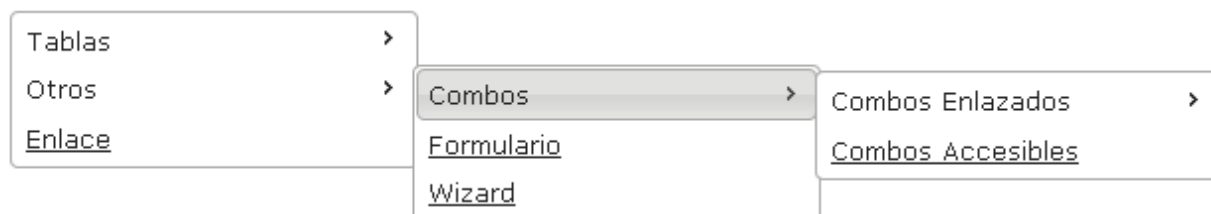
- Cuando exista un sitio web estructurado jerárquicamente y relativamente amplio.
- Cuando se desee queramos facilitar y agilizar la navegación y dar contexto al usuario.

A continuación se muestra un ejemplo del patrón para las dos posibles alternativas del patrón:

▪ Menú horizontal:



▪ Menú vertical:



- Patrón **Migas**: muestra a los usuarios la ruta de navegación que ha seguido por la aplicación permitiéndole volver a niveles superiores hasta la página de inicio. Los casos de uso de este patrón son los siguientes:

- Cuando queramos mostrar al usuario la ruta de navegación que ha seguido, en qué página está y permitirle volver fácilmente a niveles superiores.
- Se recomienda usarlas en aplicaciones web con tres o más niveles de jerarquía ya que es en este tipo de aplicaciones donde son realmente útiles.

Por qué



- Es una solución conocida por los usuarios y fácil de comprender.
- Aporta flexibilidad a la navegación al permitir volver a niveles superiores.
- Ocupa poco espacio en la interfaz.
- Refuerza el contexto y la navegación ya que muestra a los usuarios dónde están y cómo está estructurada la información.
- Ayuda a los usuarios a comprender la estructura de la aplicación.

A continuación se muestra una maqueta típica del patrón:

Usted está en: [Inicio](#) ▶ [RUP Patrones](#)

- Patrón **Mantenimiento**: este patrón facilita la lógica necesaria en las acciones básicas sobre una tabla, denominadas crud (create, read, update y delete) y el comportamiento de la página. Se aconseja la utilización de este patrón, cuando se realicen mantenimientos de tablas haciendo uso de las especificaciones establecidas en la guía de desarrollo de UDA. A continuación se enumeran las funcionalidades que proporciona el patrón para más detalle ver documentos de Patrones RUP de CAC.
 - Creación automática del formulario de detalle.
 - Añade los botones del formulario de detalle.
 - Añade el área de paginación al formulario de detalle.
 - Añade los botones del formulario de búsqueda.
 - Creación de la toolbar y adición de los botones por defecto.
 - Obtener los datos del servidor para recargar el formulario de detalle.
 - Posibilidad de crear mantenimientos con multiselección.
 - Edición en línea.
 - Validaciones automáticas y control de cambios en el formulario de detalle.
 - Mantenimientos maestro detalle.

A continuación se muestra una maqueta típica del patrón:

Castellano [Cambiar Idioma](#)

[Inicio](#)
Fase 1
Fase 2
Fase 3
Fase 4
Fase 5
Fase 6

Usted está en: [Inicio](#) > RUP Patrones

Mantenimiento (Selección simple)

[Añadir](#)
[Editar](#)
[Eliminar](#)
[Filtrar](#)

Criterios de búsqueda:

Orden Id: Lugar de Envío:

Estado:

[Buscar](#) [Limpiar](#)

| ID Pedido | Lugar de envío | Estado | Ult. Actua | Descuento |
|-----------|----------------|--------|------------|-----------|
| 5678 | sadasd | S | 26/12/2010 | 3 |
| 6354 | dsdfsdf | S | 23/02/2011 | 23 |
| 6412 | sad | S | 03/08/0021 | 1 |
| 6666 | kjhkjh | S | 06/12/2010 | 1 |
| 8888 | jhfgjh | S | 28/12/2010 | 6 |
| 8999 | jejejejeje | S | 21/12/2010 | 3 |
| 9172 | ssss | S | | 0 |
| 9541 | mas | S | 01/12/2010 | 3 |
| 9648 | eeeeeeeeee | S | 17/01/2011 | 66 |
| 9777 | Mi casa | S | | 2 |

[Primera Pagina](#)
[Anterior](#)
Página 2 de 3
[Siguiente](#)
[Ultima Pagina](#)

Mostrando 11 - 20 de 21

EJIE 2010 ©

- Patrón **Diálogos**: Permite lanzar un subproceso o un mensaje de confirmación dentro de un proceso principal sin salirse de este. Es el sustituto mejorado del antiguo pop-up.

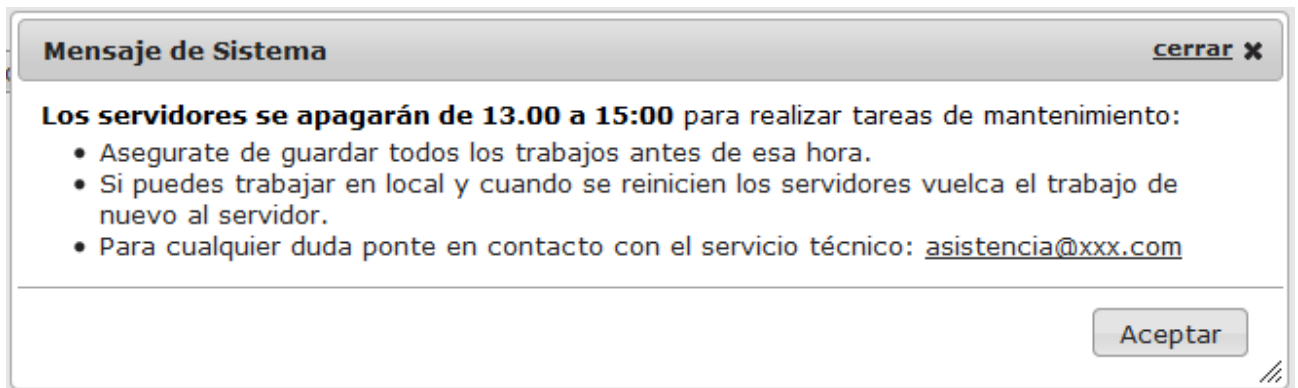
El uso de las ventanas modales debe ser únicamente en ocasiones muy concretas; tales como:

- Subprocesos dentro de un proceso principal.

La estructura de una ventana modal debe consistir en una capa semitransparente que deje ver ligeramente el proceso principal que se está llevando a cabo para dejar claro al usuario que sigue trabajando en ese proceso. Sobre la capa semitransparente se debe añadir la capa con el contenido del subproceso. Debe constar, además del contenido en sí mismo, un aspa de cierre, un enlace para cancelar la acción y un botón destacado para la ejecución de la acción.

La funcionalidad implementada en los diálogos permite que el desarrollador decida si el diálogo a mostrar debe ser realmente modal o no. Del mismo modo se permite configurar el tamaño de las ventanas, si se pueden redimensionar, arrastrar...

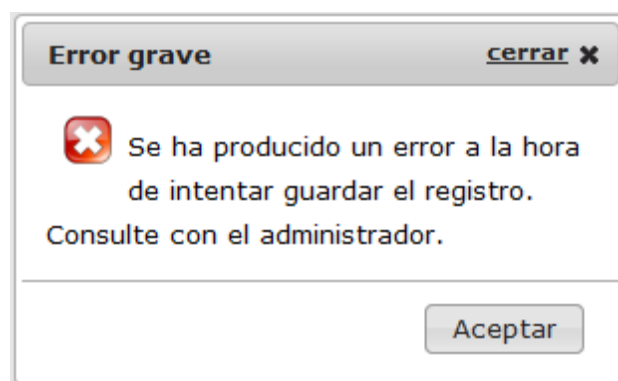
A continuación se muestra una maqueta típica del patrón:



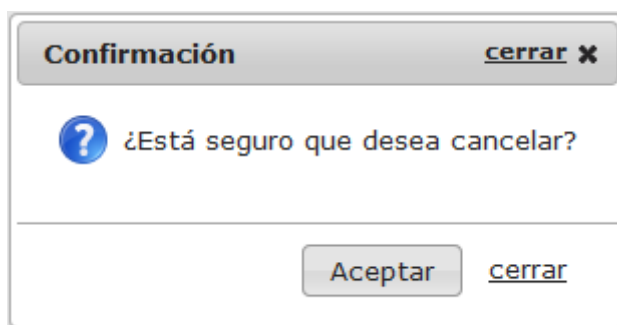
- Patrón **Mensajes**: El patrón Mensajes tiene el objetivo de mostrar al usuario de forma homogénea, clara y llamativa los posibles mensajes que pueden desencadenar las acciones que realiza en la aplicación. Estos mensajes pueden ser de error, confirmación, aviso o alerta como se muestran en las imágenes inferiores. Se recomienda el uso del patrón:
 - Cuando el usuario tenga una necesidad de información ya sea por cambios en el sistema, ejecución de procesos, realización de tareas o confirmación de acciones siempre que sea imperativo que el usuario tenga constancia, si no es tan crítico se usarán los mensajes integrados en la pantalla (patrón Feedback).

A continuación se muestra un ejemplo de cada tipología del patrón:

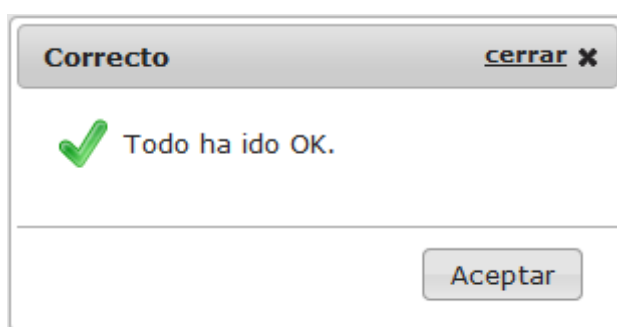
Mensajes de error:



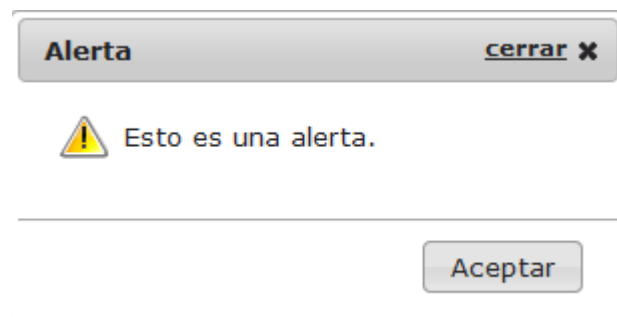
Mensajes de confirmación:



Mensajes de aviso:



Mensajes de alerta:



- Patrón **Feedback**: sirve para informar al usuario de cómo interactuar con los elementos de la aplicación o del resultado de cualquier acción que realice o cualquier problema que tenga y como solucionarlo. Se recomienda el uso del patrón:

- Cuando el usuario tenga una necesidad de información ya sea por cambios en el sistema, ejecución de procesos, realización de tareas o confirmación de acciones.

En concreto, los principales tipos de feedback que tenemos que considerar son los siguientes:

- Mensajes de sistema o informativos.
- Feedback de formularios y edición.

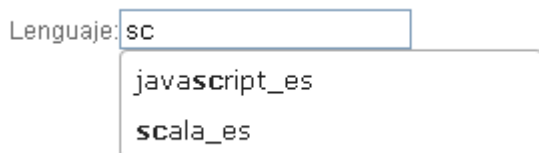
- Estado de un proceso.
- Mensajes previos / pre visualización.

Se presentan a continuación un ejemplo del patrón:



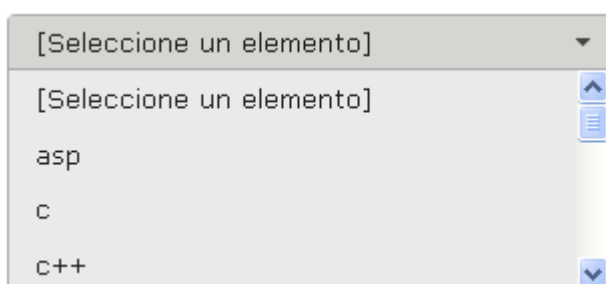
- Patrón **Autocomplete**: sirve para rellenar de forma automática un elemento de la aplicación sobre el que se está realizando una búsqueda, mostrando las opciones que cumplen las condiciones según se escribe en el elemento. Se recomienda el uso de este patrón, cuando se desea mejorar la búsqueda ofreciendo sugerencias a los usuarios

A continuación se muestra un ejemplo de este patrón:



- Patrón **Combo**: permite al usuario recuperar un elemento de una lista de elementos o de varias listas dependientes de forma sencilla y ocupando poco espacio en la interfaz. Se recomienda el uso de este patrón cuando la entrada de datos dependa de una selección de información previa. El ejemplo más común es la selección de provincia y municipio

A continuación se muestra un ejemplo de este patrón:



Este patrón puede configurarse para agrupar las opciones desplegadas en grupos, o mostrar imágenes asociadas al texto descriptivo de la opción, y se pueden cargar desde local o con una petición al servidor.

Athletic ▼

Futbol

Alaves

Athletic

Real Sociedad

Baloncesto

Caja Laboral

BBB

Lagun Aro

Formula 1

Fernando Alonso

- Patrón **Grid**: presenta a los usuarios los datos de forma que les sean fácilmente escaneables visualmente y que puedan encontrar fácilmente la información que buscan. Incluye una serie de funcionalidades como la paginación y el control de número de elemento que se quieren visualizar, así como la ordenación por columnas. Se recomienda el uso del patrón cuando tengamos que presentar a los usuarios filas de datos y queramos que les resulte fácil encontrar la información que buscan

A continuación se muestra una maqueta típica del patrón:

| ID Pedido | Lugar de envio | Estado | Ult. Actua | Descuento |
|-----------|----------------|--------|------------|-----------|
| 9777 | Madrid | S | | 2 |
| 8999 | Barcelona | S | 21/12/2010 | 3 |
| 9541 | Madrid | S | 01/12/2010 | 3 |
| 4444 | Madrid | S | 29/12/2010 | 3 |
| 32459 | Madrid | S | 29/12/2010 | 2 |
| 9648 | Barcelona | S | 17/01/2011 | 66 |
| 1234 | Madrid | N | 28/12/2010 | 12 |
| 9876 | Madrid | N | 28/12/2010 | 2 |
| 2345 | Madrid | N | 28/12/2010 | 4 |
| 5678 | Madrid | S | 26/12/2010 | 3 |

[Primera Pagina](#)
[Anterior](#)

 Página de 3

[Siguiente](#)
[Ultima Pagina](#)

 10 ▼

Mostrando 1 - 10 de 23

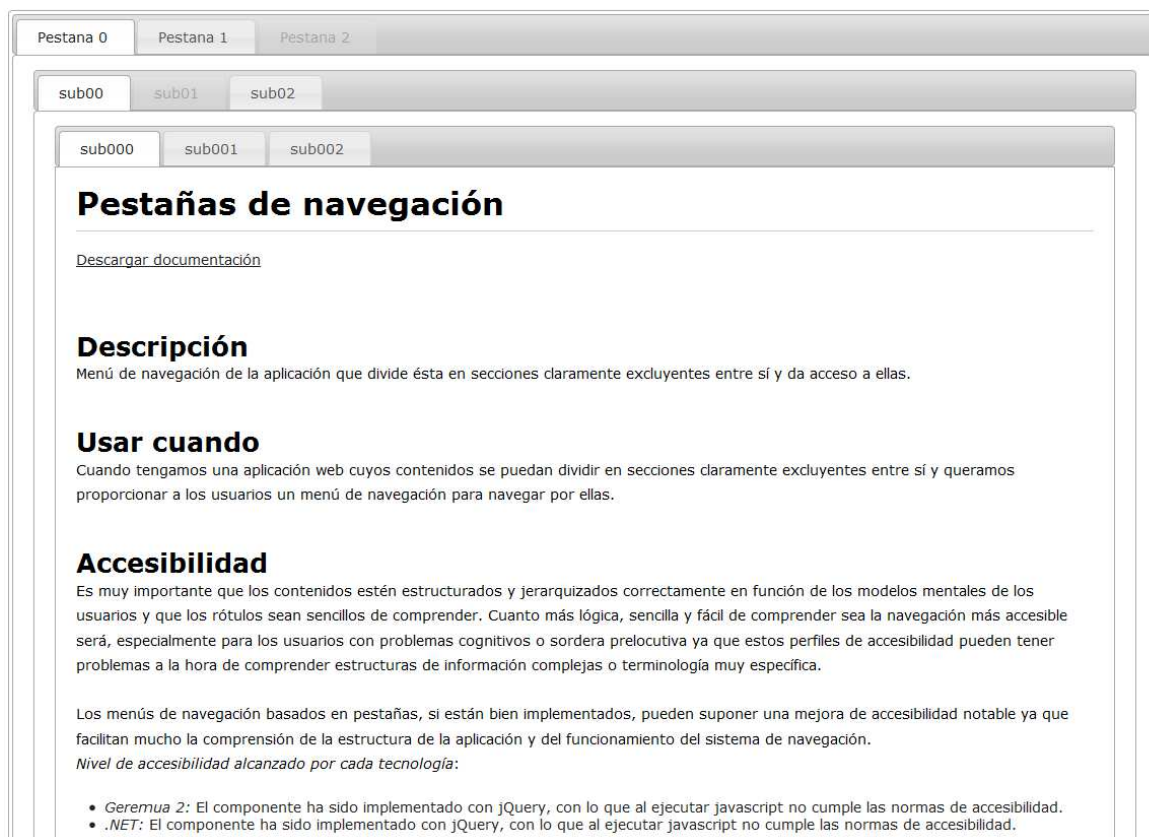
- Patrón **Tooltip**: sistema de ayuda directa al usuario en el elemento exacto que se encuentra. El objetivo del sistema de ayuda debe ser, por un lado ayudar al impaciente y/o al usuario ocasional tan extensamente como sea posible y, por otro, ayudar a los usuarios expertos. Se recomienda el uso de este patrón cuando se desea añadir una ayuda/descripción extra sobre algún componente de tal manera que se muestre al interaccionar con dicho componente. Por defecto UDA aplica este patrón a todos los elementos susceptible de ayuda haciendo uso de las propiedades del propio elemento.

Se presentan a continuación un ejemplo de este patrón:

Nombre: Introduzca su nombre.

- Patrón **Tabs**: Permiten dar acceso de forma compacta a grupos de contenidos mutuamente excluyentes pudiendo ser integradas en zonas muy reducidas de la interfaz. Se recomienda el uso cuando se pretende separar grupos de contenidos mutuamente excluyentes. El patrón permite el anidamiento de n niveles de profundidad y la posibilidad cargar el contenido de las pestañas vía petición Ajax además de la habilitación y des habilitación de las mismas.

A continuación se muestra un ejemplo del patrón con n pestañas anidadas:



Pestañas de navegación

[Descargar documentación](#)

Descripción
Menú de navegación de la aplicación que divide ésta en secciones claramente excluyentes entre sí y da acceso a ellas.

Usar cuando
Cuando tengamos una aplicación web cuyos contenidos se puedan dividir en secciones claramente excluyentes entre sí y queramos proporcionar a los usuarios un menú de navegación para navegar por ellas.

Accesibilidad
Es muy importante que los contenidos estén estructurados y jerarquizados correctamente en función de los modelos mentales de los usuarios y que los rótulos sean sencillos de comprender. Cuanto más lógica, sencilla y fácil de comprender sea la navegación más accesible será, especialmente para los usuarios con problemas cognitivos o sordera prelocutiva ya que estos perfiles de accesibilidad pueden tener problemas a la hora de comprender estructuras de información complejas o terminología muy específica.

Los menús de navegación basados en pestañas, si están bien implementados, pueden suponer una mejora de accesibilidad notable ya que facilitan mucho la comprensión de la estructura de la aplicación y del funcionamiento del sistema de navegación.

Nivel de accesibilidad alcanzado por cada tecnología:

- Geremua 2: El componente ha sido implementado con jQuery, con lo que al ejecutar javascript no cumple las normas de accesibilidad.
- .NET: El componente ha sido implementado con jQuery, con lo que al ejecutar javascript no cumple las normas de accesibilidad.

3.2 Control

La subcapa de Control de UDA se basa en el paradigma REST. Es decir, todos los controladores que UDA genera son servicios Web RESTful que la capa visual ha de consumir.

REST (REpresentational State Transfer) es un estilo de arquitectura de software para sistemas hipermedia distribuidos como la World Wide Web. Básicamente consiste en una serie de servidores y de clientes, donde los clientes inician peticiones a los servidores, estos las procesan y devuelven la respuesta correspondiente. Tanto en las peticiones como en las respuestas se transmite una representación de los recursos hipermedia.

Podríamos decir que REST es un mecanismo para el intercambio y manipulación de recursos a través de Internet. Es semejante a los Web Services, pero no usa un protocolo concreto para el intercambio de la información (los Web Services usan SOAP), de manera que podemos intercambiar información en formato HTML, XML, JSON, etcétera. En UDA, se utiliza el formato JSON.

Inicialmente REST se describe en el contexto de HTTP, por lo que se aprovecha de todas sus características: URIs, tipos de contenido (content type), sesiones, cookies, seguridad, cache, etcétera.

Las operaciones típicas que se pueden hacer con REST son:

- GET, para recuperar un recurso. Es idempotente, es decir si la ejecutamos la misma petición más de una vez siempre devuelve el mismo recurso.
- POST, para añadir recursos. No es idempotente, es decir si la ejecutamos dos veces estaremos añadiendo dos recursos.
- PUT, para modificar un recurso. Es idempotente, si la ejecutamos más de una vez la modificación es siempre la misma (por ejemplo cambiar el nombre de una persona en una agenda).
- DELETE, para borrar un recurso. Es idempotente, si lo ejecutamos más de una vez el resultado es siempre el mismo: el recurso deja de estar en el sistema (la primera vez se borra realmente, las siguientes veces simplemente se ignora la petición, pero no da error).

Esta subcapa de Control en UDA tiene cuatro principales cometidos, que son:

- Gestionar la vista a devolver a cada petición de tipo navegación.
- Gestionar el mapeo o traducción de información de las peticiones de datos al modelo de datos.
- Interactuar con la capa de servicios de negocio para trasladar el modelo de datos construido a la(s) operación(es) correspondiente(s).
- Gestionar las excepciones que se produzcan en esta capa o en inferiores, informando al sistema de Logging y al usuario.

Al final de este capítulo, se comentan los métodos que genera UDA por defecto, a nivel de Controller.

3.2.1. Navegación

Al iniciar una interacción entre cliente y servidor, el primer paso consiste en obtener la vista deseada. Para ello, el cliente a de enviar una petición al Controller correspondiente, haciendo referencia a la URL de la que cuelga la vista, como por ejemplo:

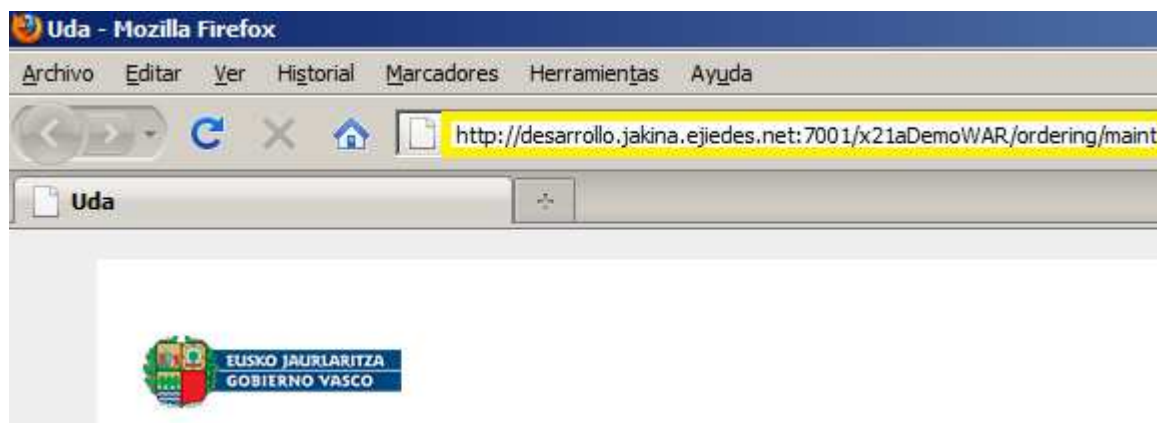


Ilustración 32. Petición de navegación.

En la parte del servidor, esta petición se mapea directamente al siguiente método del Controller “OrderingController”, que a su vez esta escuchando las peticiones que llegan al contexto “/ordering”.

```
@RequestMapping(value = "maint", method = RequestMethod.GET)
public String getCreateForm(Model model) {
    return "ordering";
}
```

Como se puede observar, este método devuelve el texto “ordering”, que será utilizado por los componentes responsables de proporcionar la vista (Apache Tiles) para proporcionar la vista deseada:

```
<definition extends="template" name="ordering">
    <put-attribute name="content" value="/WEB-INF/views/ordering/ordering.jsp"/>
    <put-attribute name="includes" value="/WEB-INF/views/ordering/ordering-
includes.jsp"/>
</definition>
```

Tal y como se puede comprobar en el extracto del fichero tiles.xml, las encargadas, en última instancia, de generar la vista, serán las JSPs que se definen en el mapeo “ordering”.

3.2.2. Peticiones de datos

Una vez que el cliente ya dispone de la vista, al interactuar con la misma, se generan las peticiones de modificación de datos. Estas peticiones Ajax RESTful se mapean con métodos concretos del Controller, aunque después estos métodos pueden, internamente, invocara más métodos del propio Controller o de la capa subyacente (servicios de negocio).

El mapeo entre peticiones de datos y métodos se realiza discriminando inicialmente la URL, después por método HTTP y por último, por el contenido de dichas peticiones.

Por ejemplo, una petición “GET” con contexto “/ordering/orderid” se mapea directamente con el siguiente método:

```
@RequestMapping(value =("/{orderid})", method = RequestMethod.GET)
public @ResponseBody Ordering getById(@PathVariable Long orderid) {...}
```

En este caso, el Controller mapea directamente la última parte de la URL con la variable “orderid”, aunque también es posible recibir datos en forma de entidad:

```
@RequestMapping(value = "/count", method = RequestMethod.GET)
public @ResponseBody Long getAllCount(@RequestParam(value = "ordering", required
= false) Ordering filterOrdering, HttpServletRequest request) {...}
```

En este caso, todos los campos de la petición se mapearan automáticamente a un objeto de tipo "Ordering".
Por último, también es posible decibir los datos de una petición unitariamente, como en el siguiente ejemplo:

```
@RequestMapping(method = RequestMethod.GET)
public @ResponseBody Object getAll(
    @RequestParam(value = "orderid", required = false) Long orderid,
    @RequestParam(value = "lastupdate", required = false) Timestamp lastupdate,
    @RequestParam(value = "status", required = false) String status,
    @RequestParam(value = "shipmentinfo", required = false) String shipmentinfo,
    @RequestParam(value = "discount", required = false) Long
    discount, HttpServletRequest request) {...}
```

En cualquier caso, una vez que se mapea a Objetos Java la información enviada por el cliente, el Controller se encarga de llamar a otros métodos del propio Controller y/o métodos del servicio de negocio del que dispone.

```
@Autowired
private OrderingService orderingService;
```

En cuanto al retorno de los datos (o generación del objeto HttpServletResponse) Spring MVC se encarga automáticamente de hacerlo, en base al Objeto que se anote con @ResponseBody.

```
public @ResponseBody Ordering ...
```

En estos casos, Spring MVC se encarga de serializar estos Objetos al formato indicado por la petición, que en el caso de UDA en `application/json`, por lo que Spring invocará a Jackson para que serialize dicha información y posteriormente la incluirá en el Objeto HttpServletResponse.

3.2.3. Interacción con capa de servicios de negocio

La llamada a servicios de negocio transaccionales (o no) resulta tan sencilla como invocar mediante Java al servicio correspondiente, pasando como parámetro los datos enviados por el cliente:

```
@RequestMapping(value =("/{orderid}", method = RequestMethod.GET)
public @ResponseBody Ordering getById(@PathVariable Long orderid) {
    try {
        Ordering ordering = new Ordering();
        ordering.setOrderid(orderid);
        ordering = this.orderingService.find(ordering);
        if (ordering == null) {
            throw new Exception(orderid.toString());
        }
        return ordering;
    } catch (Exception e) {
        throw new ResourceNotFoundException(orderid.toString());
    }
}
```


3.2.4. Gestionar Excepciones (desactualizado)

Toda excepción que se produzca en la subcapa de control o en subcapas subyacentes, tendrá que ser capturada por el Controller correspondiente. En caso de no ser capturada, se escribirá directamente en los ficheros de Logging y provocará la devolución de la página de error al cliente.

```
<mvc:view-controller path="/error" view-name="error" />
```

Esta página la proporciona por defecto UDA, pero reside en los Wars y es totalmente customizable por las aplicaciones.

Si las excepciones son capturadas, opción que se recomienda insistentemente, conviene transformar dicha excepción en otra que Spring MVC sepa transferir a la vista en modo JSON.

Para ello, se recomienda crear una excepción por cada código de error HTTP que se quiera informar al cliente. Por ejemplo, si se intenta obtener un recurso determinado, por ejemplo, "/ordering/1", es decir, un elemento de la tabla Ordering cuya clave primaria sea "1" y dicha tupla no existe, se recomienda informar a la vista con un HttpStatus.NOT_FOUND junto con el texto correspondiente. Por ejemplo:

```
@RequestMapping(value =("/{orderid}", method = RequestMethod.GET)
public @ResponseBody Ordering getById(@PathVariable Long orderid) {
    try {
        Ordering ordering = new Ordering();
        ordering.setOrderid(orderid);
        ordering = this.orderingService.find(ordering);
        if (ordering == null) {
            throw new Exception(orderid.toString());
        }
        return ordering;
    } catch (Exception e) {
        throw new ResourceNotFoundException(orderid.toString());
    }
}
```

En este método, si el servicio de negocio no ha encontrado datos, se lanza una excepción genérica, con el código de la tupla que no ha sido encontrada.

Esta excepción genérica, y otras que puedan ser producidas por los servicios de negocio y la capa de acceso a datos, son capturadas por el "catch", que genera una excepción de tipo "ResourceNotFoundException" y se alimenta con el código de la tupla no encontrada.

Por último, esta excepción es capturada por el método "handleException", que ha de existir en todo Controller.

```
@ExceptionHandler
public @ResponseBody String handle(ControlException e) {
    logger.log(Level.WARN, e.getMessage());
    return e.getMessage();
}
```

Este método captura dichas excepciones, informa al sistema de Logging y la devuelve a Spring MVC, que se encargará de generar el paquete HTTP con el código de estado correspondiente (404 en este caso) y como mensaje, introducirá el código de la tupla no encontrada.

Se pueden crear tantas excepciones de este tipo como se desee, siempre y cuando, se siga con el siguiente patrón:

```
@ResponseStatus(value=HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends ControlException {
```

```
private static final long serialVersionUID = 1L;  
  
public ResourceNotFoundException(String message) {  
    super(message);  
}  
  
}
```

Donde se puede jugar con el código de estado (que en el ejemplo corresponde a 404, NOT FOUND).

3.2.5. Métodos generados por UDA

A continuación se comentan los métodos que UDA genera automáticamente para los Controllers:

3.2.5.1. *getcreateForm*

Este método sirve exclusivamente para pedir al Controller una página.

```
@RequestMapping(value = "maint", method = RequestMethod.GET)
public String getcreateForm(Model model){...}
```

Este tipo de petición de navegación es de tipo GET y se devuelve el nombre correspondiente de la definición de Tiles a aplicar.

3.2.5.2. *getById*

Este método sirve para obtener una entidad (que representa a una tupla de base de datos) a partir de su clave primaria.

```
@RequestMapping(value =("/{orderid}", method = RequestMethod.GET)
public @ResponseBody Ordering getById(@PathVariable Long orderid) {...}
```

En el caso de tratarse de una entidad con clave primaria compuesta, UDA se encarga de generar el método recibiendo tantos parámetros como sean necesarios. Por ejemplo, para una entidad con clave primaria compuesta por "itemid" y "orderid", se obtiene lo siguiente.

```
value =("/{itemid}/{orderingorderid}"
```

NOTA: Se concatena en campo "ordering" a la propiedad "orderid" si se trata de una Primary Foreign Key para contra la tabla "Ordering".

3.2.5.3. *getAll*

Este método sirve para obtener una entidad o conjunto de entidades que cumplan con las reglas de filtrado que se indican. Estas reglas de filtrado se corresponden con los propios datos de la entidad que envía en cliente (este concepto de Query By Example se explica más adelante).

```
@RequestMapping(method = RequestMethod.GET)
public @ResponseBody Object getAll(
    @RequestParam(value = "itemid", required = false) Long itemid,
    @RequestParam(value = "orderingorderid", required = false) Long orderingorderid,
    @RequestParam(value = "vendorPartVendorpartnumber", required = false) BigDecimal
    vendorPartVendorpartnumber,
    @RequestParam(value = "quantity", required = false) Long quantity,
    HttpServletRequest request) {...}
```

Además, se pueden enviar más parámetros referentes a la paginación, que se reciben en el interior de este método.

```
pagination.setPage(Long.valueOf(request.getParameter("page")));
pagination.setRows(Long.valueOf(request.getParameter("rows")));
pagination.setSort(request.getParameter("sidx"));
pagination.setAscDsc(request.getParameter("sord"));
```

Todos estos parámetros se insertarán en los Objetos correspondientes ("Ordering" y "Pagination" en este caso) y se pasan como parámetro al servicio.

```
List<Ordering> orderings = this.orderingService.findAll(ordering, pagination);
```

Este método llama internamente al método `getAllCount` para obtener los datos de paginación necesarios (número de tuplas que corresponden al filtrado a realizar).

3.2.5.4. `getAllCount`

Este método se utiliza para obtener el número de entidades o tuplas que corresponden con el filtrado que se desea realizar. Se puede invocar directamente desde la vista, o a través del método "getAll".

```
@RequestMapping(value = "/count", method = RequestMethod.GET)
public @ResponseBody Long getAllCount(
    @RequestParam(value = "ordering", required = false) Ordering filterOrdering,
    HttpServletRequest request) {...}
```

Mapa directamente los campos de la request con el Objeto Ordering e invoca al método correspondiente de la capa de servicios de negocio, de manera que obtiene el número de tuplas correspondientes al filtrado.

```
return orderingService.findAllCount(filterOrdering != null ? filterOrdering : new
Ordering());
```

3.2.5.5. `edit`

Este método es invocado a la hora de realizar modificaciones sobre tuplas o entidades ya existentes.

```
@RequestMapping(method = RequestMethod.PUT)
public @ResponseBody Ordering edit(@RequestBody Ordering ordering,
    HttpServletResponse response) {...}
```

Simplemente, mapea los campos enviados por la vista a una entidad la pasa como parámetro al método correspondiente de la capa de servicios de negocio.

```
Ordering orderingAux = this.orderingService.update(ordering);
```

3.2.5.6. `add`

Este método añade nuevas tuplas a la tabla de base de datos correspondiente.

```
@RequestMapping(method = RequestMethod.POST)
public @ResponseBody Ordering add(@RequestBody Ordering ordering) {...}
```

Simplemente, mapea los campos enviados por la vista a una entidad la pasa como parámetro al método correspondiente de la capa de servicios de negocio.

```
Ordering orderingAux = this.orderingService.add(ordering);
```

3.2.5.7. `remove`

Este método es invocado a la hora de eliminar tuplas o entidades ya existentes.

```
@RequestMapping(value =("/{orderid})", method = RequestMethod.DELETE)
```

```
public void remove(@PathVariable Long orderid, HttpServletResponse response) {...}
```

Recibe la(s) clave(s) primaria(s) de la tupla a eliminar, la(s) mapea a la entidad y pasa la entidad al servicio de negocio como parámetro.

```
this.orderingService.remove(ordering);
```

3.2.5.8. *removeAll*

Recibe una serie de claves primarias correspondientes a tuplas existentes en base de datos desde la vista y las mapea en sus correspondientes entidades.

```
@RequestMapping(value = "/deleteAll", method = RequestMethod.POST)
public void removeMultiple(@RequestBody ArrayList<ArrayList<String>> orderingIds,
HttpServletResponse response) {...}
```

Después las elimina.

```
this.orderingService.removeMultiple(orderingList);
```

3.2.5.9. *handle*

Gestiona las excepciones que la vista recibirá.

```
@ExceptionHandler
public @ResponseBody
String handle(ControlException e) {...}
```

3.2.5.10. *bindxxxxyy*

Método que se utiliza para añadir tuplas en las tablas intermedias de las relaciones ManyToMany. Es decir, cuando dos tablas están relacionadas en modo ManyToMany, UDA da la posibilidad de vincular tuplas de ambas tablas relacionadas en su tabla intermedia (ver el capítulo de acceso a datos para más detalles).

Para ello, hay que invocar al método bindxxxxyy, como ejemplo:

```
@RequestMapping(value = "/bindvendorPayment", method = RequestMethod.POST)
public void bindvendorPayment(@RequestParam(value = "vendorid", required = false)
Long vendorid, @RequestParam(value = "paymentPaymentid", required = false)
BigDecimal paymentPaymentid) {...}
```

Para ello, se recogen las claves primarias de las tuplas a relacionar, se insertan en las entidades correspondientes, se anidan estas entidades de manera oportuna y se le pasa la entidad principal al servicio de negocio.

```
vendor.getPayments().add(payment);
this.vendorService.addvendorPayment(vendor);
```

3.2.5.11. *unBindxxxxyy*

Este método es la antítesis del anterior, ya que se encarga de desvincular tuplas en una relación de tipo ManyToMany, eliminando exclusivamente la tupla de la tabla intermedia.

```
@RequestMapping(value = "/unbindvendorPayment", method = RequestMethod.POST)
public void unBindvendorPayment(@RequestParam(value = "vendorid", required =
false) Long vendorid, @RequestParam(value = "paymentPaymentid", required = false)
BigDecimal paymentPaymentid) {...}
```

Para ello, se recogen las claves primarias de las tuplas a desvincular, se insertan en las entidades correspondientes, se anidan estas entidades de manera oportuna y se le pasa la entidad principal al servicio de negocio.

```
vendor.getPayments().add(payment);
this.vendorService.removevendorPayment(vendor);
```

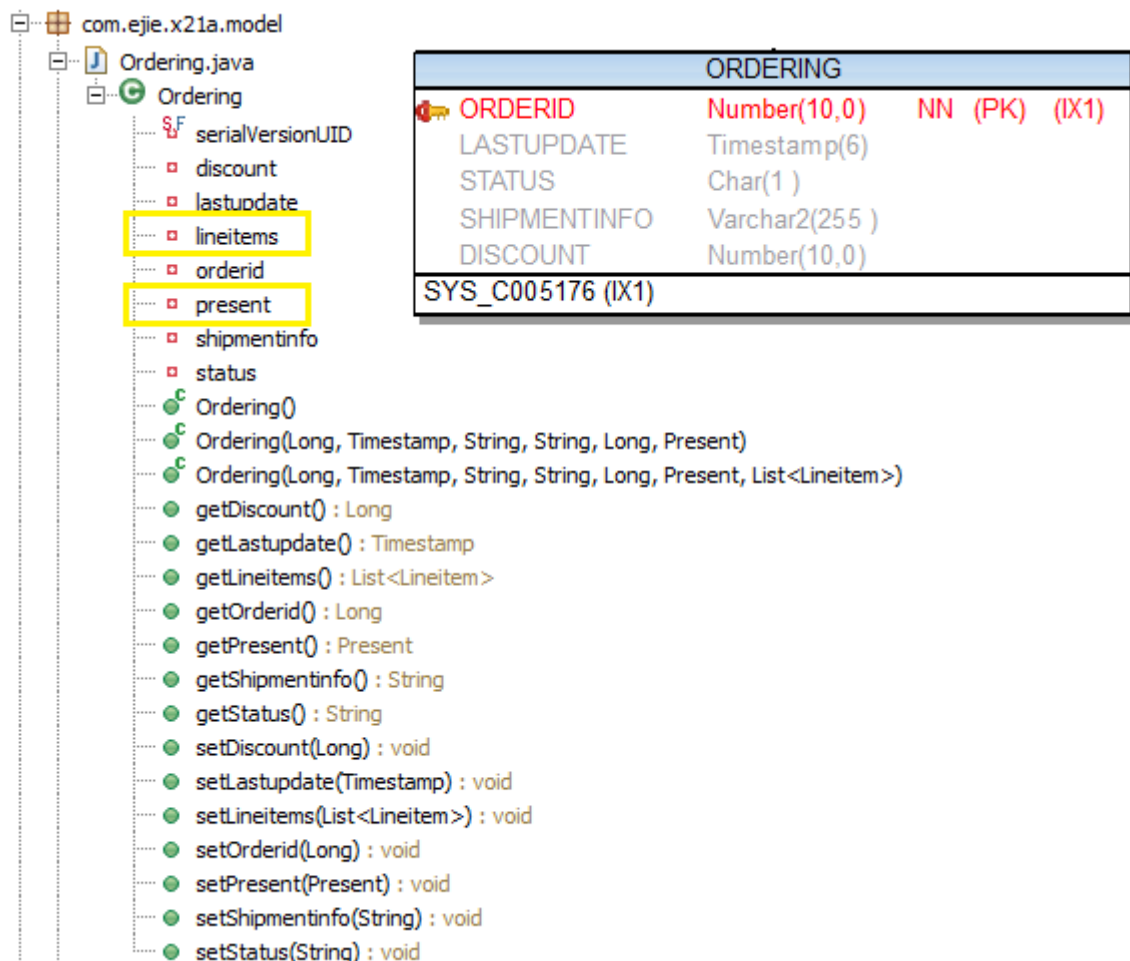
4 Modelo de datos

Uno de los elementos fundamentales del diseño de una aplicación es la definición de los objetos que conforman el modelo de datos de la aplicación. En un modelo de capas cada una de ellas tiene acceso únicamente a las funciones expuestas por la capa siguiente con la excepción del modelo. El modelo es una capa vertical utilizada por todas las capas horizontales, es por ello que el modelo no utiliza las funciones de ninguna de las capas horizontales. Cuando se dice que el modelo es utilizado por todas las capas horizontales se hace referencia a que habitualmente forma parte de los parámetros de entrada y salida que definen sus interfaces.

Para clarificar más este último punto, en inglés las capas horizontales son llamadas layers y las verticales tiers. El modelo pertenece a tiers.

A continuación se hace un repaso de la manera en la que UDA genera el modelo de datos de manera automática, partiendo de un modelo relacional de base de datos.

En UDA, cada tabla existente en el modelo relacional de base de datos, será representada por una clase con su mismo nombre y sus mismas propiedades (o columnas). Como ejemplo, se muestra una tabla del esquema que se distribuye con UDA y la clase Java que la representa:



| ORDERING | | |
|-------------------|----------------|---------------|
| ORDERID | Number(10,0) | NN (PK) (IX1) |
| LASTUPDATE | Timestamp(6) | |
| STATUS | Char(1) | |
| SHIPMENTINFO | Varchar2(255) | |
| DISCOUNT | Number(10,0) | |
| SYS_C005176 (IX1) | | |

Ilustración 33. Representación de la tabla como modelo en Java.

Como se puede observar en la imagen anterior, el Objeto “Ordering” representa la tabla “ORDERING”. Todas sus propiedades coinciden (UDA también se encarga de mapear los tipos de datos, por ejemplo, de

VARCHAR2 a Sting) excepto las enmarcadas en cuadros amarillos, que representan las relaciones de la tabla “ORDERING” con otras tablas:

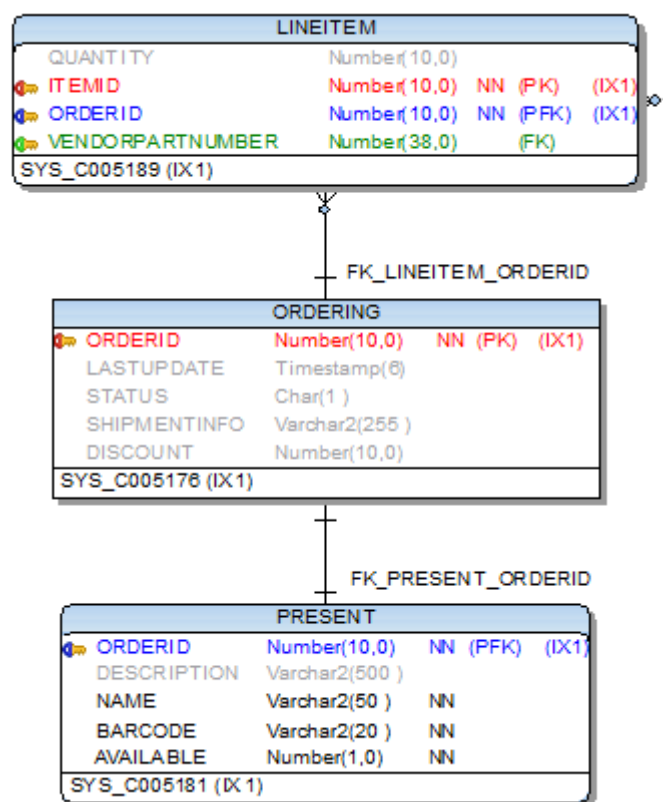


Ilustración 34. Relación de otras tablas con ORDERING

Si se retoman las propiedades del Objeto “Ordering” remarcadas en amarillo, se puede comprobar que coinciden con las propiedades de las tablas colindantes que conforman las Foreign Keys de las tablas “LINEITEM” y “PRESENT” hacia “ORDERING”.

Haciendo mayor hincapié en las propiedades que representan a las tablas colindantes, se observa que en el caso de “Present”, se trata de un Objeto simple que se almacena como propiedad del Objeto “Ordering”, ya que la relación entre ambas tablas es de 1:1. Sin embargo, la propiedad “lineitems” que representa la relación con la tabla “LINEITEM” es una colección de Objetos de tipo “Lineitem”, debido a que la relación entre esa tabla y “ORDERING” es de 1:N.

En cierta manera estos objetos representan un diccionario de datos de la aplicación y permiten hablar un lenguaje común a lo largo de la misma y por consiguiente fomentan la reutilización del código.

Adicionalmente, en la arquitectura, el modelo de datos albergará las anotaciones que definirán las validaciones que se aplicarán a los datos que contiene el modelo.

En las aplicaciones JPA, las anotaciones (o decoradores) que representan el modelo relacional de datos, se aplicarán en el modelo de datos, tal y como se explica más adelante.

Por lo tanto, hay que tener muy presente que **el diseño del modelo de datos condicionará irremediablemente la complejidad de las aplicaciones**, haciendo que estas puedan ser realmente simples e intuitivas, o por el contrario, desmesuradamente complejas y enrevesadas. Como conclusión, se aconseja **diseñar concienzudamente el modelo Entidad-Relación de Base de Datos**, ya que UDA realiza la ingeniería inversa de dicho esquema, creando automáticamente todas las entidades correspondientes al modelo de datos, entre otras cosas.

Actualmente, UDA soporta dos tipos de motores de persistencia. Por un lado está el motor JDBC y por otro lado se ofrece el motor JPA. La elección de uno u otro modelo condiciona fuertemente al modelo de datos. A continuación se hace mención de los principales aspectos que afectarán al modelo en uno u otro caso.

4.1 Representación del esquema relacional en Java

UDA se encarga de crear automáticamente la capa vertical llamada Modelo de datos, reflejando fielmente en Java el esquema relacional que subyace en base de datos.

En este capítulo se reflejan diferentes ejemplos obtenidos tras generar código con UDA partiendo del siguiente diagrama Entidad-Relación (ver imagen):



UDA_Schema.jpg

Al generar el modelo, UDA crea automáticamente tantos Objetos en el paquete “com.ejje.app.model” como tablas se hayan seleccionado en el asistente (excepto en JPA, donde se crea una clase adicional por cada Primary Key compuesta). Además, UDA permite excluir columnas de dichas tablas para que no se reflejen en el modelo y no sean gestionadas por la aplicación.

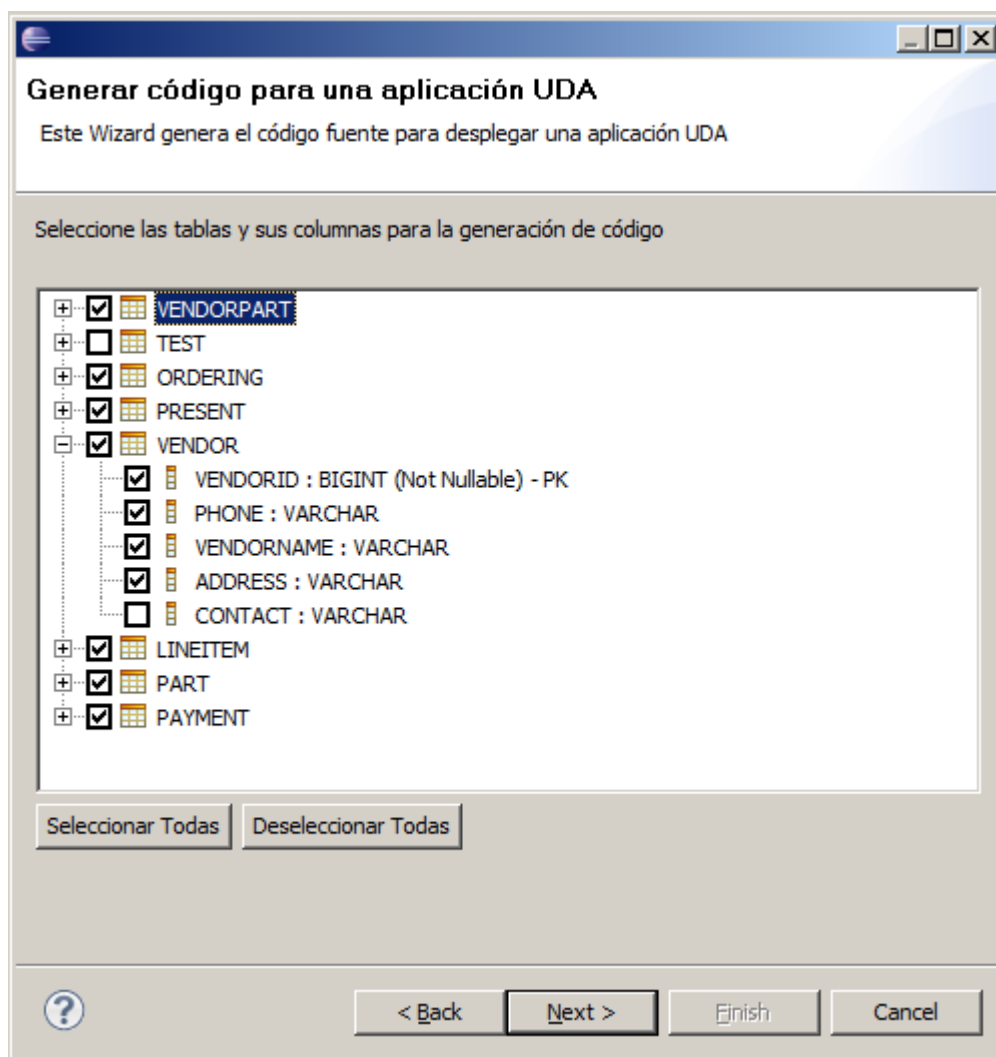


Ilustración 35. Selección de tablas y columnas.

Tras realizar la selección de tablas y columnas indicada en la imagen anterior, UDA genera los siguientes objetos en el paquete model de la aplicación:

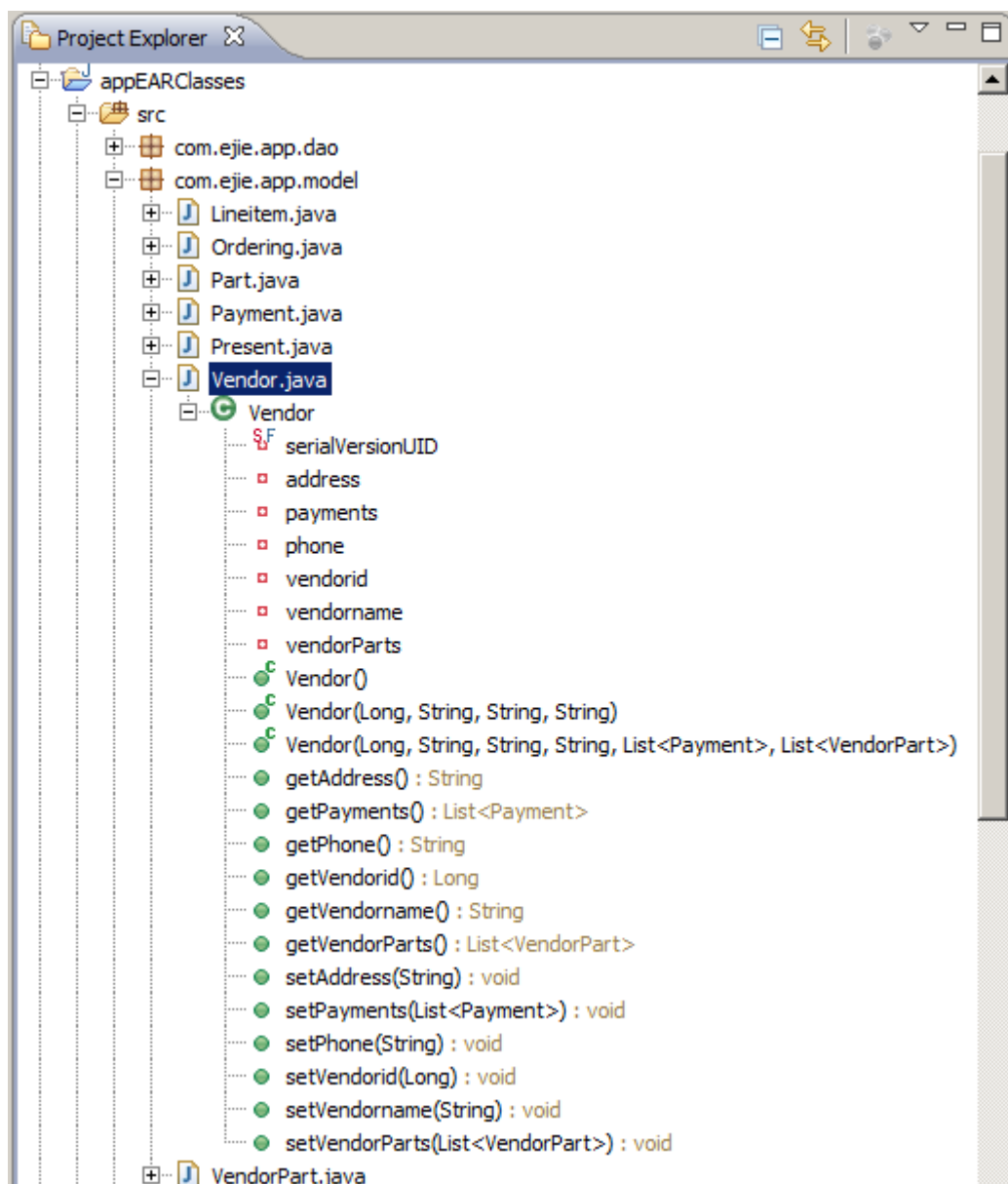


Ilustración 36. Modelo generado tras la selección.

Como se puede observar, UDA ha ignorado la tabla “TEST” y la columna “CONTACT” de la tabla “VENDOR”, ya que ninguna de las dos fue seleccionada en el asistente.

Para el resto de casos, existe una clase por tabla y una propiedad en cada clase por cada columna de su respectiva tabla. Además, UDA se encarga de realizar el mapeo de datos y la normalización de los nombres.

Dependiendo del modelo relacional de base de datos, UDA establece ciertas conexiones o dependencias entre las clases del modelo.

4.1.1. Relación OneToOne (1:1)

A continuación se ilustra el caso en el que el modelo representa a una relación 1:1 en base de datos Oracle (en otros sistemas gestores de bases de datos, esta relación se plasma de diferente manera).

NOTA: En Oracle, para conseguir una relación 1:1, la tabla padre y la hija tiene que tener la misma Primary Key, incluyendo nombre de la columna y tipo de dato.

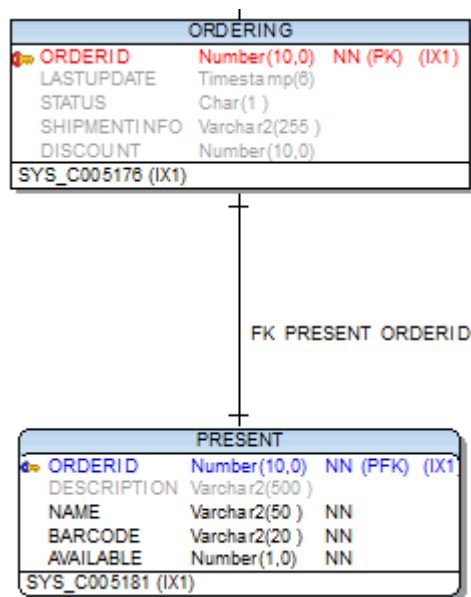


Ilustración 37. Relación 1:1 entre ORDERING y PRESENT.

En esta situación, se entiende que un pedido solo puede tener un regalo, y un regalo pertenece exclusivamente a un pedido determinado.

Esto en UDA (y en cualquier modelo orientado a Objetos) se representa de la siguiente manera:

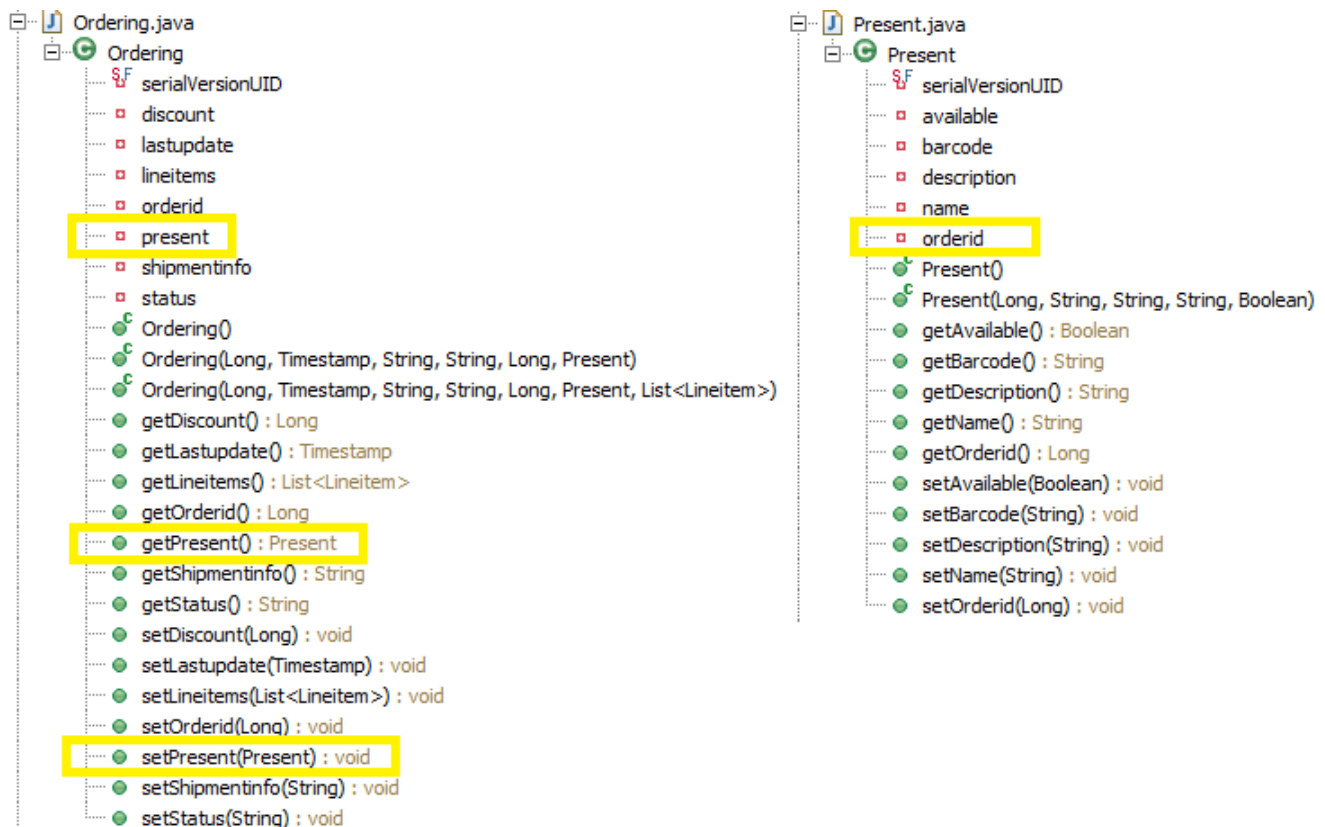


Ilustración 38. Modelo de Ordering y Present.

Por un lado, se observa que la clase Ordering contiene una propiedad llamada “present” que es de tipo “Present”, con sus correspondientes métodos getter y setter. Aquí es donde subyace la relación, ya que un Objeto de tipo Ordering contiene un único Objeto de tipo Present.

Por otro lado, la clase Present contiene una propiedad llamada “orderid”, del mismo tipo que la propiedad del mismo nombre de la clase Ordering, en este caso, java.lang.Long.

4.1.2. Relación ManyToOne (M:1)

En los casos en los que varias filas de una tabla corresponden a una única fila de otra tabla se produce el siguiente escenario.

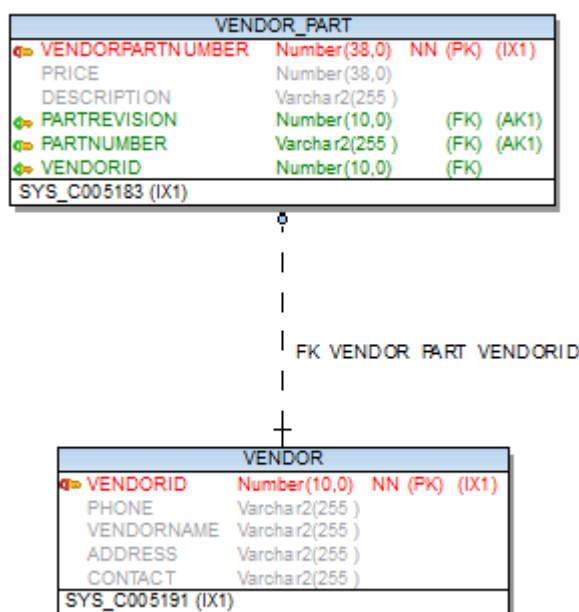


Ilustración 39. M:1 entre VENDOR_PART y VENDOR.

En este caso, varias piezas de un proveedor, albergadas en la tabla “VENDOR_PART” que provienen de un único proveedor.

Esto en Java se representa de la siguiente manera.

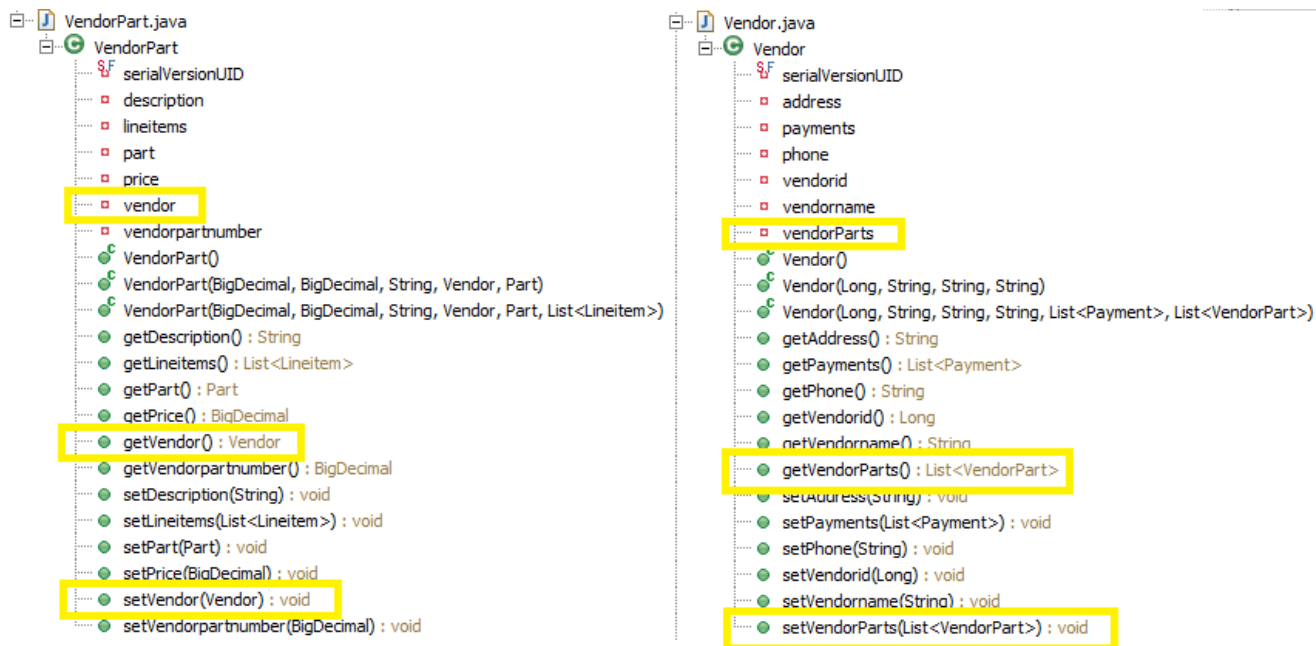


Ilustración 40. Modelo M:1 entre VendorPart y Vendor.

En la imagen, se observa que la clase “VendorPart” mantiene una propiedad “vendor” de tipo “Vendor”. Esto representaría la parte *ToOne, mientras que la clase “Vendor” tiene como propiedad “vendorParts” que es una colección de objetos VendorPart. Esto correspondería a la parte ManyTo*.

4.1.3. Relación OneToMany (1:N)

En este apartado se ilustra el caso en el que el modelo representa a una relación 1:N.

Además, el caso que se va a utilizar como ejemplo es un tanto especial, ya que existen dos tablas relacionadas, donde una de ellas (la hija) tiene una Primary Key compuesta por dos campos. Uno de estos campos, se corresponde con la Primary Key de la tabla padre, haciéndola ser una Primary Foreign Key (PFK).

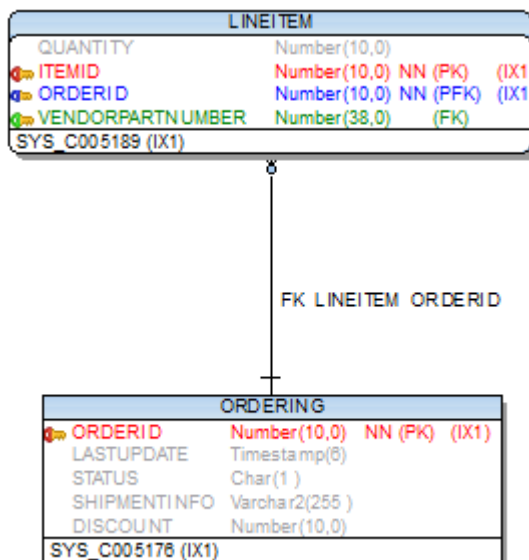


Ilustración 41. Relación 1:N con PFK.

Esta relación refleja que un pedido “ORDERING” puede contener N líneas. Cada una de estas líneas “LINEITEM” tiene un código propio (“ITEMID”), una cantidad (“QUANTITY”) y pertenece a un pedido (“ORDERID”) y a una pieza de un proveedor (“VENDORPARTNUMBER”), a la que se hace referencia a través de una Foreign Key.

A su vez, los campos “ITEMID” (PK) y “ORDERID” (PFK) conforman la Primary Key compuesta de la tabla.

Todo esto, en Java se resuelve de la siguiente manera:

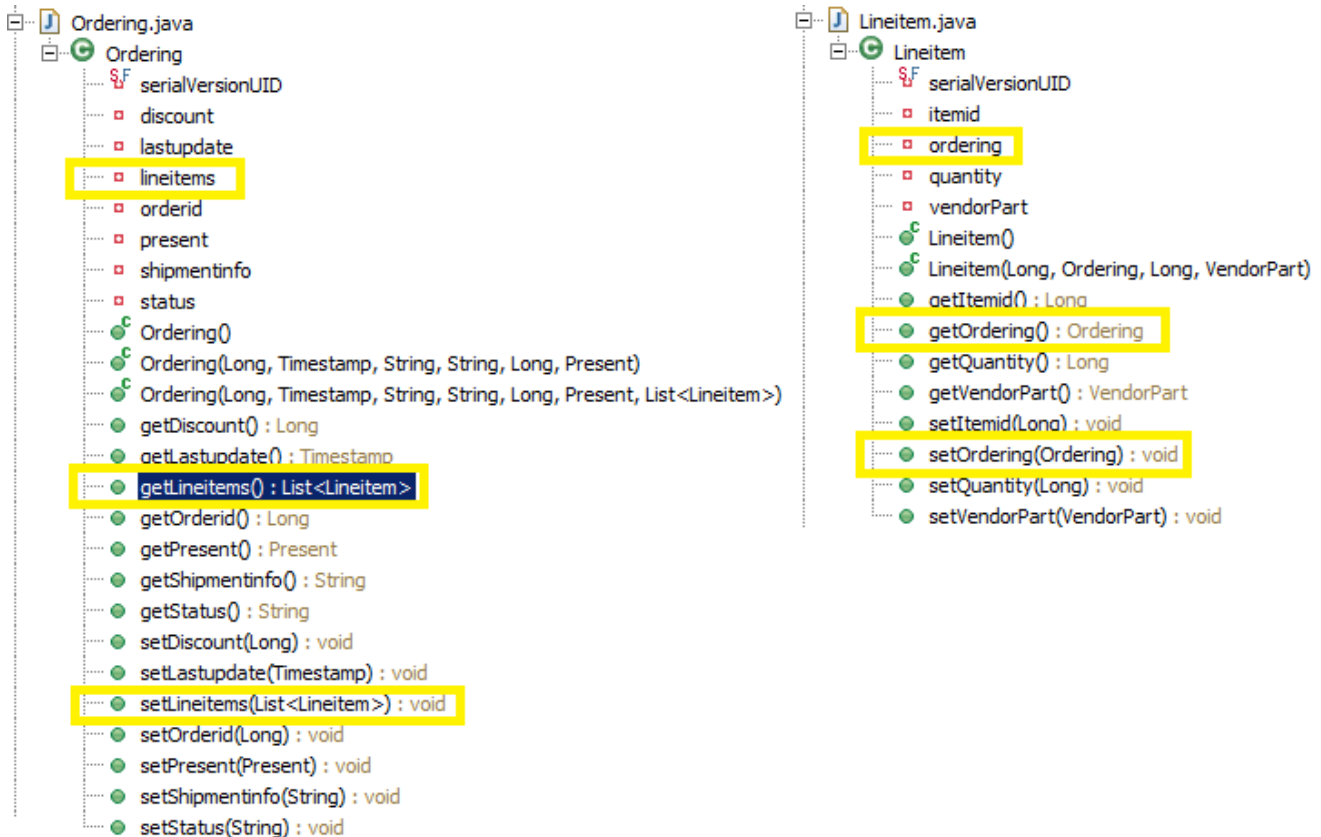


Ilustración 42. Modelo de Lineitem y Ordering.

Por un lado, en cuanto a Ordering, se aprecia que contiene una propiedad llamada “lineitems”, que representa a una colección de objetos de tipo “Lineitem”. Esto sería la parte OneTo*.

Por otro lado, en la parte *ToMany, se observa que en Lineitem, la clase tiene una propiedad llamada “ordering” del tipo “Ordering”, con sus correspondientes métodos getter y setter. Sin embargo, no se aprecia que las columnas “ORDERID” y “VENDORPARTNUMBER” se encuentren representadas. Esto se debe a que la representación de dichas columnas subyace en los propios Objetos relacionados, como son la propiedad “ordering” y la propiedad “vendorPart”.

Así, se consigue que la un objeto “Ordering” pueda contener N objetos Lineitem, pero un “Lineitem” solo pueda contener un Ordering.

4.1.4. Relación MayToMany (M:N)

En los casos en los que varias filas se relacionan con varias filas, el escenario es un tanto más complejo, ya que aparece un nuevo concepto: tabla intermedia.

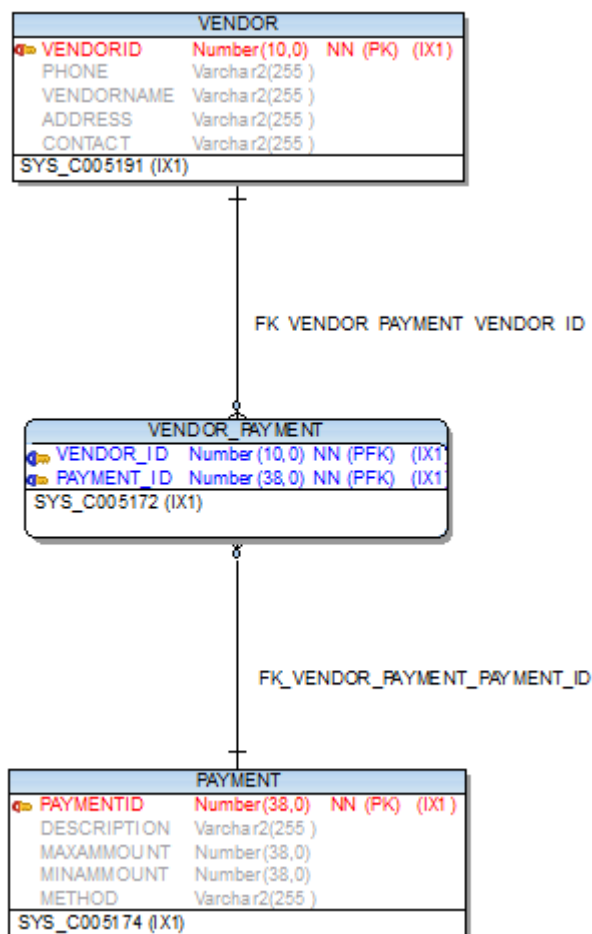


Ilustración 43. ManyToMany entre VENDOR y PAYMENT.

En este caso, se observa que aparece una nueva tabla llamada “VENDOR_PAYMENT” en la cual se definen las relaciones entre las filas de “PAYMENT” y de “VENDOR” a través de crear registros conjugando sus Primary Keys.

NOTA: En UDA, así como en Hibernate Tools, es estrictamente necesario que **la tabla intermedia este compuesta exclusivamente de las Primary Keys** de las tablas relacionadas. Si se añade algún campo extra a la tabla intermedia, dejará de considerarse tabla intermedia y se le dará el tratamiento que se le da a cualquier otra tabla. **Las tablas intermedias no son una entidad y por lo tanto no se crea una clase Java que las represente.** Si a una tabla intermedia se le añade un campo extra (con lo cual dejará de considerarse intermedia) se la tratará como entidad y se creará una clase Java que la represente.

La representación de la relación entre estas tablas en Java, queda de la siguiente guisa:

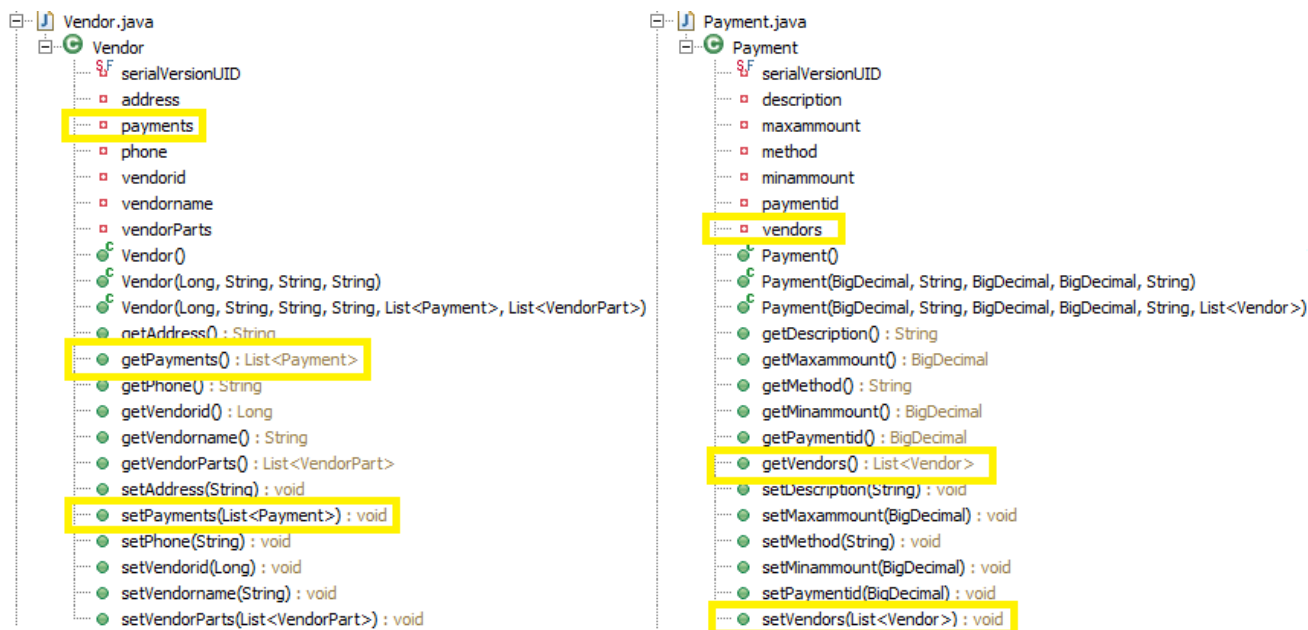


Ilustración 44. Modelo de M:N entre Vendor y Payment.

Tal y como se puede apreciar, la tabla intermedia “VENDOR_PAYMENT” no se refleja explícitamente en el modelo Java. Lo que sucede, es que la relación entre ambas entidades se representa implícitamente, ya que la clase “Vendor” tiene la propiedad `payments`, que representa a una colección de objetos “Payment” y la clase “Payment” tiene una propiedad `vendors` que representa una colección de objetos “Vendor”.

4.2 — Reglas de validación (desactualizado)

Todas las reglas de validación, absolutamente todas, se declaran una única vez para toda la aplicación.

Estas reglas se declaran sobre el modelo.

La técnica utilizada para ello consiste en aplicar las anotaciones definidas por la JSR-303 y proporcionadas por Hibernate Validator, en el paquete *org.hibernate.validator*. *

Todos los mensajes de validación, han de estar internacionalizados. Para ello, el mensaje declarado en la anotación de validación ha de estar definido en los ficheros *messages* del War o del EARClasses.

Por ejemplo, para añadir una validación referente al tamaño de una propiedad numérica de la clase “Payment”, habría que añadir la siguiente anotación sobre una propiedad:

```
@Max(value = 999, message = "max")
private BigDecimal maxamount;
```

Así, cuando se valide toda la clase “Payment” o sólo la propiedad “maxamount”, si el contenido de la propiedad supera el valor “999” se lanzará el mensaje que contiene la clave “max”.

Dependiendo de la Locale del usuario, la clave `max` se leerá del fichero *messages.properties* (por defecto), *messages_en.properties* (inglés), *messages_es.properties* (castellano), *messages_eu.properties* (euskara) o *messages_fr.properties* (francés).

Como ejemplo de este mecanismo clave/mensaje, se muestra un extracto del fichero *messages.properties*:

```
max=Permitted maximum number exceeded
```

De esta manera, se añaden todas las reglas de validación con sus respectivos mensajes al modelo.

Además de las anotaciones que proporciona Hibernate Validator, es posible crear nuevas anotaciones de validación utilizando el API que proporciona el propio Hibernate (consultar la documentación de la herramienta para más información).

Más, adelante se incide en cómo hacer uso manual de dichas reglas de validación y se explica como funciona el sistema de validación automática para la vista.

4.3 Serialización

El modelo forma parte del grupo de parámetros que definen las firmas de los métodos de los que están compuestos los Interfaces de las diferentes capas horizontales, es decir, los diferentes componentes se comunican pasándose objetos pertenecientes al modelo. Por eso, resulta fundamental que el modelo ofrezca facilidades para serializarlo de diferentes maneras.

4.3.1. Formato String

Además, en muchos casos, interesa registrar los datos que se intercambian entre llamadas a componentes de UDA. Para ello, todas las clases pertenecientes al modelo, tienen un método *toString()*, con el cual el objeto traduce toda la información que lleva embebida a un String para que el sistema de Logging de UDA refleje dicha información en Logs de manera sencilla. Por ejemplo, la clase *Ordering* se convertirá en String de la siguiente manera:

```
@Override
public String toString() {
    StringBuilder result = new StringBuilder();
    result.append(this.getClass().getName()).append(" Object { ");
    result.append(" [orderid: ").append(this.orderid).append(" ]");
    result.append(", [lastupdate: ").append(this.lastupdate).append(" ]");
    result.append(", [status: ").append(this.status).append(" ]");
    result.append(", [shipmentinfo: ").append(this.shipmentinfo)
        .append(" ]");
    result.append(", [discount: ").append(this.discount).append(" ]");
    result.append("}");
    return result.toString();
}
```

4.3.2. Serialización binaria

De la misma manera, hay que tener en cuenta que también para las llamadas remotas RMI entre EJBs, la información se transfiere embebida en el modelo. Por ello, el modelo ha de estar preparado para la (de)serialización, de manera que este proceso sea lo más óptimo posible. Esta máxima, se traduce a que cuando se use JPA, no resulta práctico serializar el modelo decorado con las anotaciones propias de JPA, ya que entre otras cosas, los servidores de aplicaciones JEE5 (como Oracle WebLogic 11g) pueden producir situaciones no deseadas al transferir entidades JPA implementadas con JPA 2.0, que es superior a la versión de JPA que el servidor de aplicaciones soporta, JPA 1.0.

Por ello, solo en los casos en los que se utilice JPA 2.0, hay que encapsular el modelo bajo un Data Transfer Object (DTO) que será el que finalmente se serialice y se envíe por RMI. En estos casos, UDA se encargará de generar el subpaquete *com.ejje.app.model.dto* automáticamente. Así pues, los proyectos JPA quedarán de esta guisa:

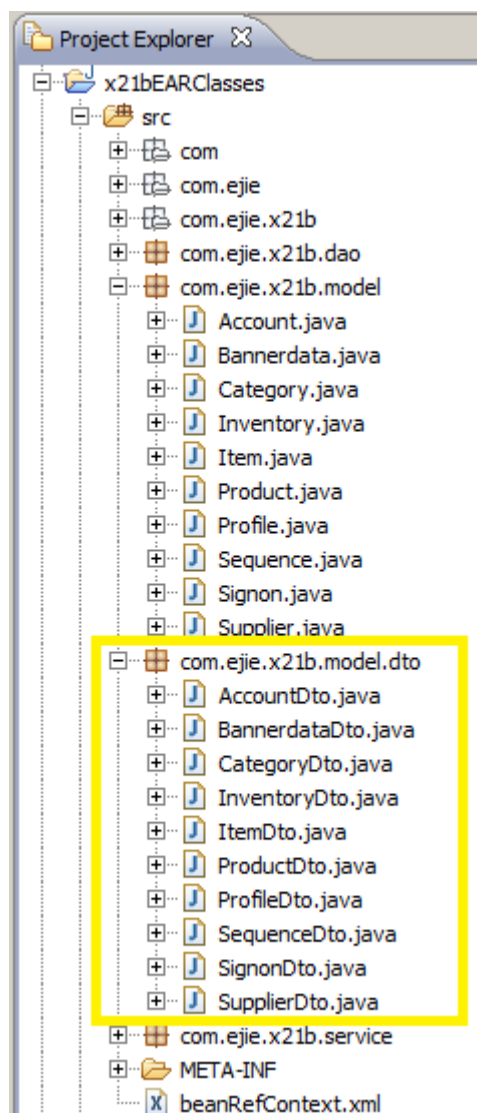


Ilustración 45. Proyecto JPA con subpaquete DTO.

Como se puede observar, las clases del subpaquete “dto”, son las mismas clases que están en el paquete “model”, con el término “Dto” concatenado. En cuanto a contenido, el cambio fundamental es que se extraen las propiedades de las clases del paquete model y se transportan a las clases del subpaquete “dto”, con sus respectivos getters y setters. De esta manera, se consigue que todos los metadatos JPA se queden en el paquete “model” y que todos los datos pasen al subpaquete “dto”. Además, las clases del paquete “model” extenderán a las del paquete “dto”, recuperando sus propiedades.

Como es de suponer, los interfaces de los EJBs que se encargan de las comunicaciones remotas RMI estarán definidas para transferir DTOs, por lo que tendrán que desencapsular los DTOs para enviar los datos y volver a encapsularlos al recibirlos.

```
@Override
@TransactionalAttribute(TransactionAttributeType.REQUIRED)
public ProductDto add(ProductDto productData, TransactionMetadata
transactionMetadata) {
    return productService.add(new Product(productData));
}
```

Esto se explicará con más detalle en la sección de Remoting.

A continuación se ilustra un ejemplo con la clase `Category` y su respectivo Dto. La primera contiene getters y setters decorados con JPA, mientras que la segunda solo contiene propiedades. La primera extiende a la segunda.

```
@Entity
@Table(name = "CATEGORY")

public class Category extends CategoryDto implements java.io.Serializable {
```

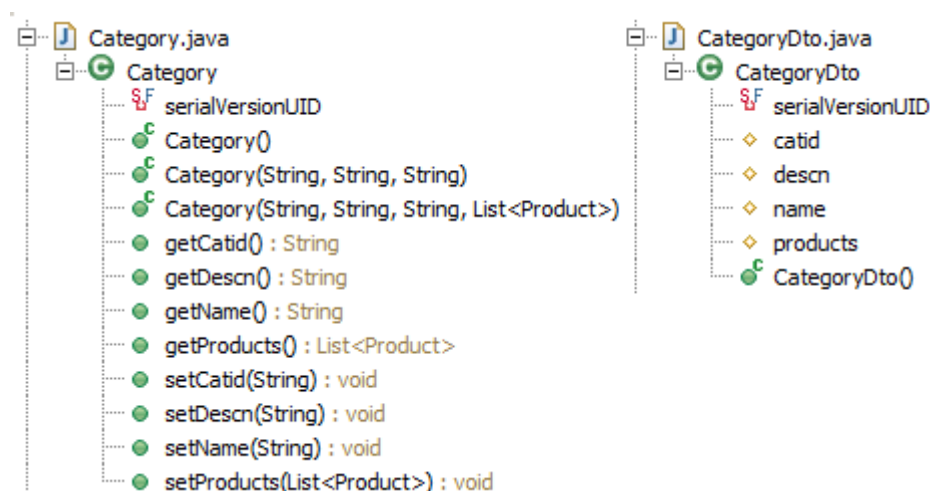


Ilustración 46. Clase `Category` con getters y setters. Clase `CategoryDto` con propiedades.

4.3.3. Formato JSON

Por último, la capa de Control (Spring MVC) también trabaja constantemente con el modelo para intercambiar información con la vista, en formato JSON.

La conversión a JSON la realiza automáticamente Spring MVC apoyándose en la librería Jackson.

Lo único a tener en cuenta son las fechas y las colecciones.

Para las fechas en formato Timestamp, UDA aplicará automáticamente las siguientes anotaciones a los getters y setters del modelo:

```
@JsonSerialize(using = JsonSerializer.class)
public Timestamp getLastupdate() {
    return this.lastupdate;
}

@JsonDeserialize(using = JsonDateDeserializier.class)
public void setLastupdate(Timestamp lastupdate) {
    this.lastupdate = lastupdate;
}
```

En cuanto a las colecciones, hay que indicar a Jackson en qué punto ha de dejar de serializar el modelo, ya que como es conocido, unas clases se componen de otras, e incluso de colecciones de otras clases, pudiendo llegar a serializar un número ingente de clases relacionadas. Para cortar esta cadena de serializaciones, se utiliza esta anotación principalmente sobre colecciones:

```
@JsonIgnore
public List<Product> getProducts() {
    return this.products;
}
```

}

4.4 Particularidades para JPA 2.0

En proyectos JPA 2.0 la capa vertical de modelo de datos alberga algunos componentes que en proyectos JDBC no existen. En concreto, se trata de Metamodel y Data Transfer Objects (DTOs en adelante).

4.4.1. Metamodel

El Metamodel es una abstracción simplificada del modelo de datos y sirve para realizar “type safe queries” con el API de Criteria que proporciona JPA 2.0. El uso del Metamodel se cubre en el capítulo de la capa de acceso a datos.

La generación del Metamodel sucede de manera automática. Una vez que el asistente de generación de código de UDA genera el modelo de datos, automáticamente lanza una tarea interna de Eclipse que genera el Metamodel en el mismo paquete en el que se encuentra el modelo.

Si se introducen cambios en el modelo, basta con tener activado el “Build Automatically” o lanzar un “Build” manual, para volver a autogenerar el Metamodel. Esto se hace de la siguiente manera:

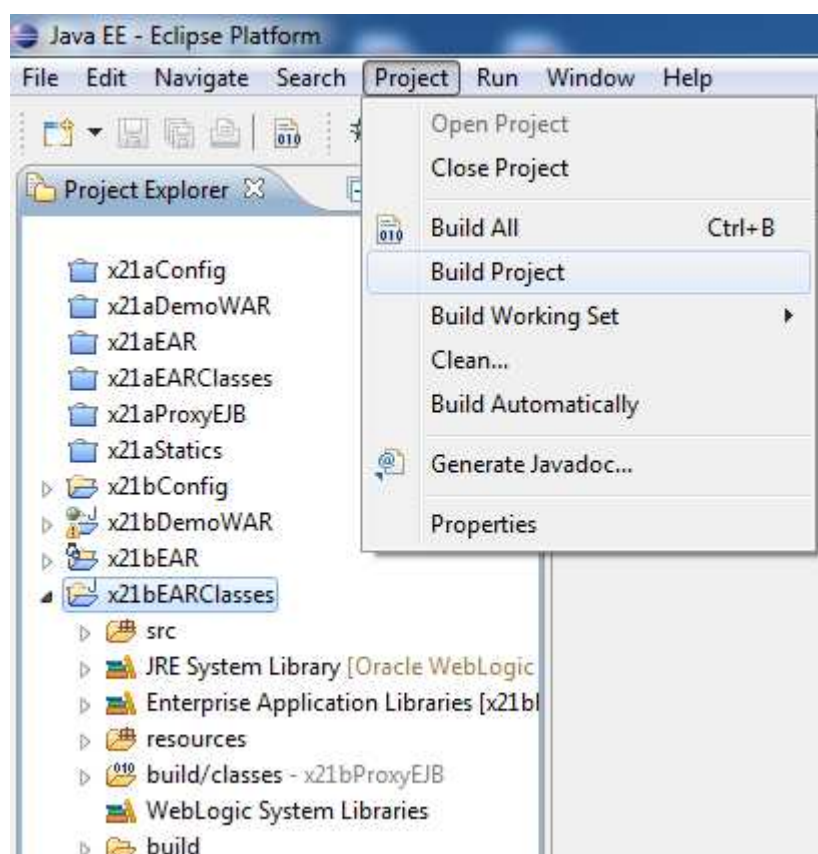


Ilustración 47. Generación de Metamodel.

Tras ejecutar el “Build”, el paquete model de la aplicación quedará de la siguiente manera:

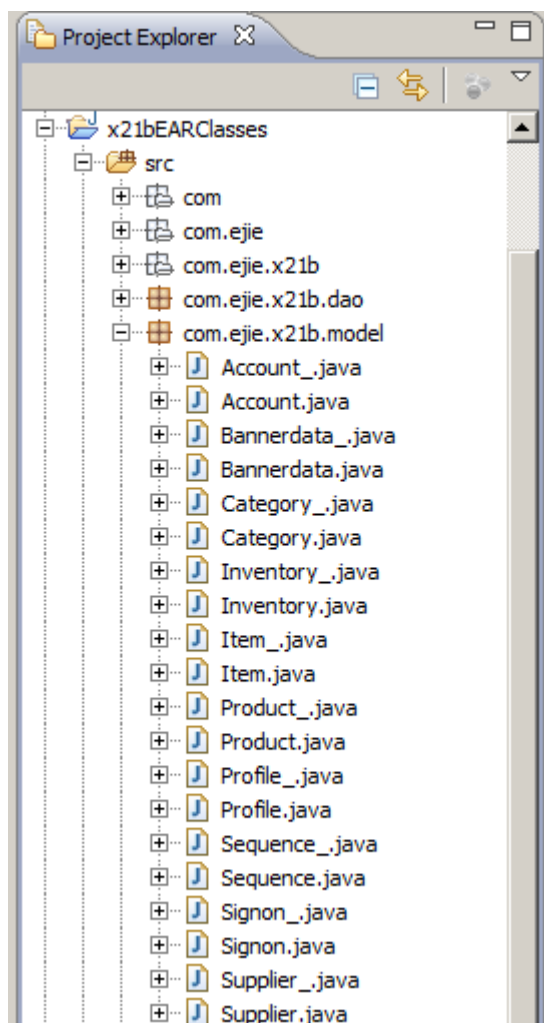


Ilustración 48. Model + Metamodel.

Por cada clase o entidad existente en el paquete model, se creará una clase con el mismo nombre concatenado con un guión bajo.

4.4.2. Data Transfer Objects

En las aplicaciones con JPA 2.0 el modelo suele contener multitud de anotaciones y demás metadatos que hacen que las clases que pertenecen a este paquete no sean meros contenedores de datos. El modelo tiende a ser una capa vertical más y más compleja, donde cada vez es más difícil abstraer la parte fundamental en la que se almacenan los datos de negocio.

La parte fundamental de toda entidad, en la que viajan sus datos es el Data Transfer Object, muy útil de cara a optimizar la (de)serialización de los objetos. Por ello, las propiedades de las clases del modelo se han de exportar a DTOs donde exclusivamente existan las propiedades fundamentales de las clases, junto con sus métodos getter y setter. Estos DTOs estarán en un subpaquete llamado "dto".

Afortunadamente, UDA se encarga de generar los DTOs automáticamente, cuando se le indica que cree la capa de modelo de datos en aplicaciones JPA 2.0.

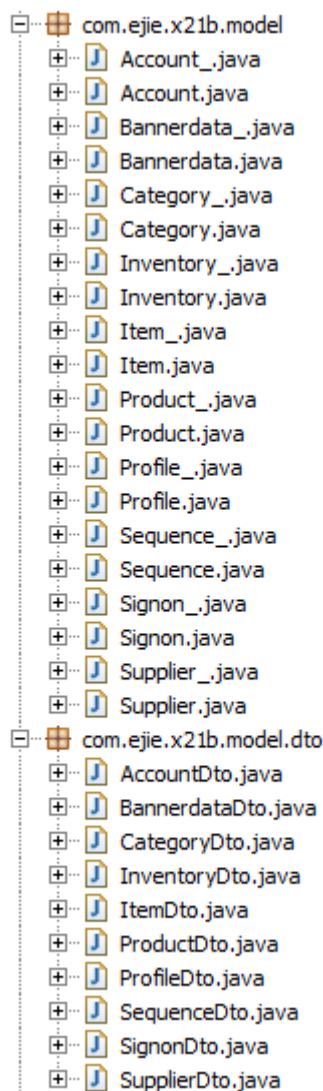


Ilustración 49. Modelo de datos + Metamodel + DTOs

De esta manera, el modelo original se desprende de sus propiedades, que se trasladan a los DTOs, y se queda únicamente con getters y setters decorados con anotaciones.

¿Y cómo es posible que el modelo original se quede sin propiedades? En realidad, no es eso lo que sucede, ya que las clases del modelo heredan de las clases DTO, recuperando así las propiedades del modelo. Además, toda clase del modelo original tendrá un constructor donde recibirá su respectivo DTO como parámetro.

En resumidas cuentas, en proyectos JPA 2.0, UDA se encargará de generar un subpaquete con DTOs que contengan exclusivamente propiedades, getters, setter y anotaciones referentes a la validación de datos. Estas últimas han de estar ligadas obligatoriamente a las propiedades.

Eso sí, todas las clases, menos las Metamodel, deben de tener la propiedad serialVersionUID.

A continuación, unos ejemplos:

- Una clase de modelo ,por ejemplo, Product:

```
@Entity
@Table(name = "PRODUCT")
public class Product extends ProductDto implements java.io.Serializable {
```

```
private static final long serialVersionUID = 2078001324267954911L;

public Product(ProductDto productDto) {
    this.productid = productDto.getProductid();
    this.category = productDto.getCategory();
    this.name = productDto.getName();
    this.descn = productDto.getDescn();
}

public Product(String productid, Category category, String name,
    String descn) {
    this.productid = productid;
    this.category = category;
    this.name = name;
    this.descn = descn;
}

public Product(String productid, Category category, String name,
    String descn, List<Item> items) {
    this.productid = productid;
    this.category = category;
    this.name = name;
    this.descn = descn;
    this.items = items;
}

@Id
@Column(name = "PRODUCTID", nullable = false, length = 10)
public String getProductid() {
    return this.productid;
}

...
...
```

- Una clase del Metamodel, por ejemplo, Product_:

```
@javax.persistence.metamodel.StaticMetamodel
(value=com.ejje.x21b.model.Product.class)
@javax.annotation.Generated
(value="org.apache.openjpa.persistence.meta.AnnotationProcessor6",date="Thu Jan
27 12:55:01 CET 2011")
public class Product_ {
    public static volatile SingularAttribute<Product,String> productid;
    ...
    ...
}
```

- Una clase DTO, por ejemplo, ProductDto:

```
public class ProductDto implements java.io.Serializable {

    private static final long serialVersionUID = -200329466974003094L;
```



```
protected String productid;  
  
public String getProductid() {  
    return productid;  
}  
  
public void setProductid(String productid) {  
    this.productid = productid;  
}  
  
...  
  
...
```

4.4.3. Tips & tricks

A decir verdad, puede que en aplicaciones grandes el paquete “model” quede algo difícil de manejar por el programador, principalmente, porque comparando con aplicaciones JBDC, el número de clases de modelo se triplica (modelo + Metamodel+ DTO).

Para mejorar el manejo de dicho paquete, se recomienda crear el Metamodel en una carpeta oculta que se encuentre al nivel de src. Para ello, sólo hay que editar las propiedades del proyecto appEARClasses, de la siguiente manera:

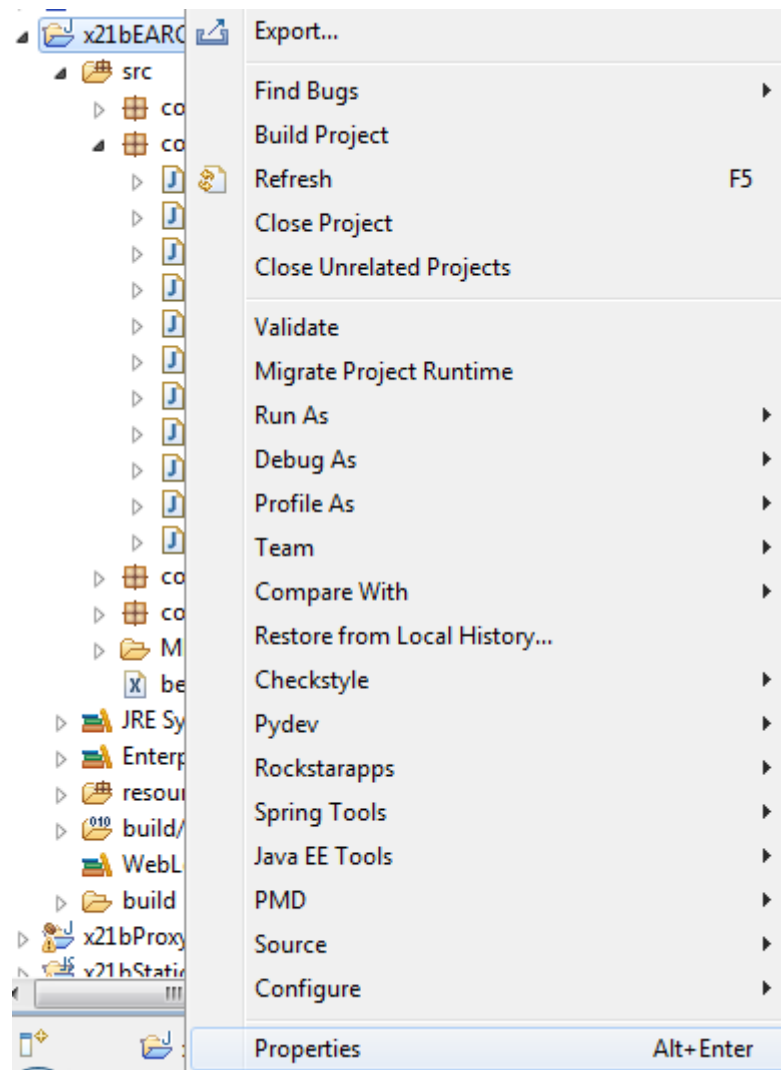


Ilustración 50. Acceder a propiedades del appEARClasses.

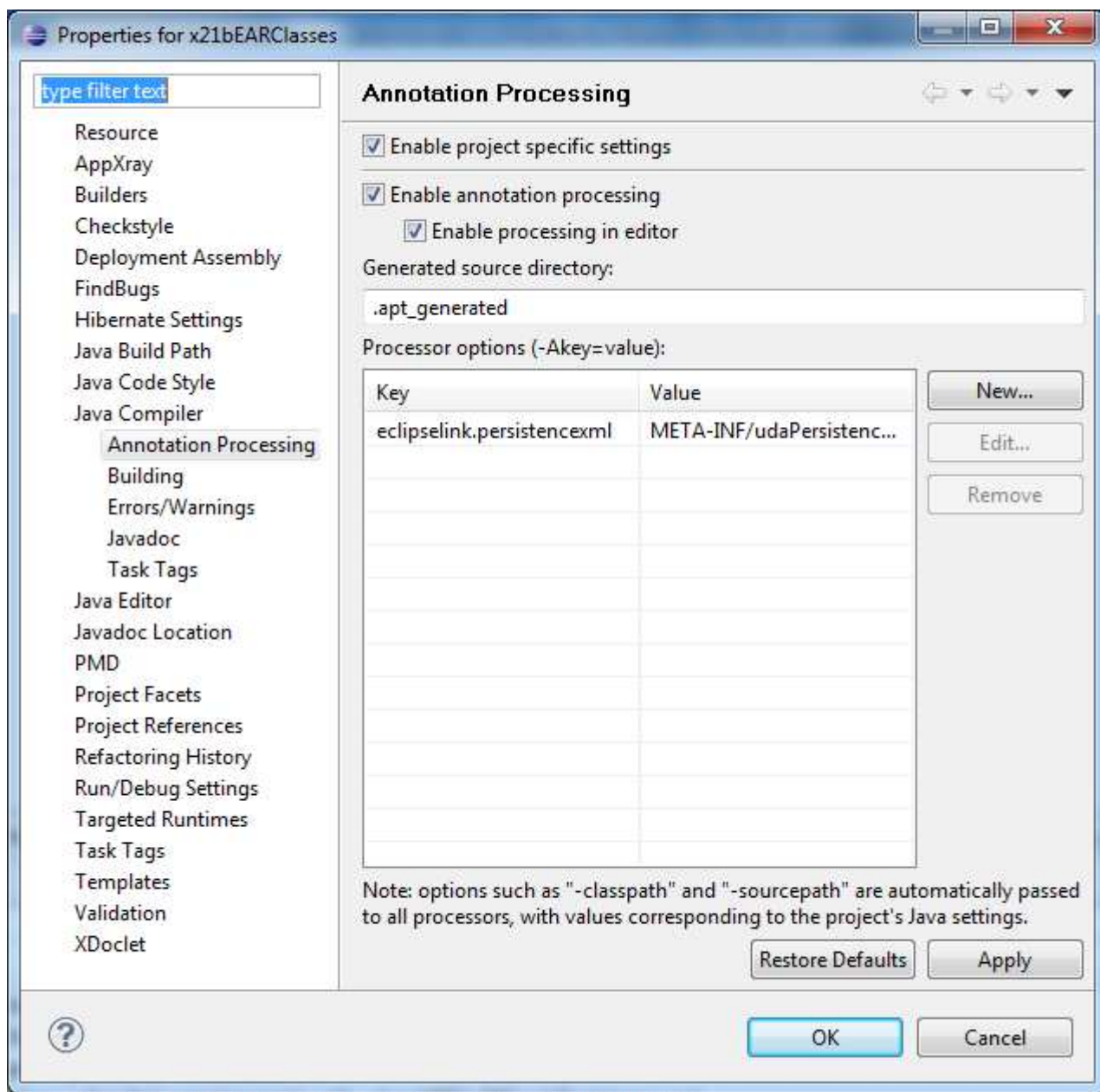


Ilustración 51. Cambiar la ruta src por .apt_generated

Así, el paquete “model” de las aplicaciones JPA 2.0 quedará mucho más limpio:

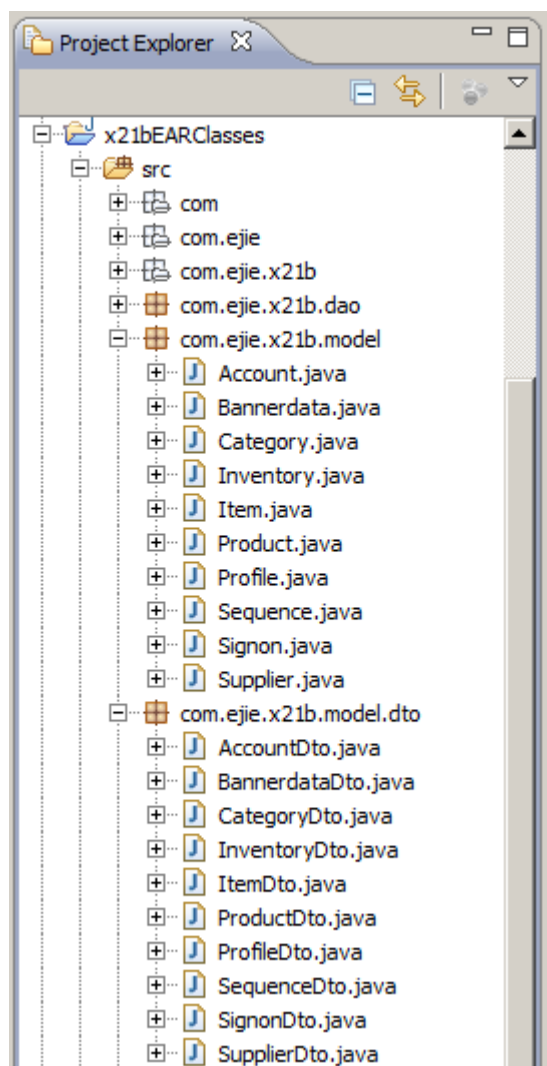


Ilustración 52. Model con el Metamodel en .apt_generated.

En cualquier caso, el Metamodel habrá que subirlo a SubVersion en el paquete model que se encuentra bajo src. Si se utiliza este truco para el desarrollo, habrá que volcar el código a src y refactorizar la aplicación antes de subirla a SubVersion.

IMPORTANTE: Este truco esta fuera de la normativa de desarrollo de Ejje, por lo que cualquier problema derivado de su uso, es problema exclusivo del proveedor que lo utilice.

5 Servicios de negocio

Esta capa es la encargada de exponer la funcionalidad de la aplicación que fundamentalmente será utilizada por los interfaces de usuario desarrollados en la capa de presentación, aunque también puede ser consumida remotamente por otras aplicaciones.

Exponiendo estos métodos la capa de servicios esconde las tecnologías utilizadas en el resto de la aplicación permitiendo la especialización de programadores de presentación y de negocio. Con excepción de los cambios en las interfaces de los servicios los cambios en el código de las clases que conforman esta capa y la de acceso a datos no afectarán al código desarrollado para la capa de presentación.

Las funciones de la capa de servicios (también denominada business logic) han de ser “coarse grained”. Por aplicar este concepto anglosajón a nuestras aplicaciones decir que las funciones de esta capa en su mayoría resuelven en una sola llamada un caso de uso de la aplicación: addProduct, removeProduct,... Recogen de una sola vez todos los datos necesarios y de igual forma devuelven toda la información en una llamada.

Por lo explicado anteriormente los métodos de las clases que conforman la capa de servicios son **el punto ideal para marcar la transaccionalidad y la seguridad (a nivel de método)**, aspectos que tienen dedicado su propio capítulo.

5.1 Diseño Técnico de la capa de Servicios de negocio

Abordar el diseño y la implementación de la capa de servicios de negocio es realmente sencillo con UDA, ya que para este menester se utiliza Spring Framework, tal y como se detalla a continuación.

5.1.1. Principios

En el diseño de la capa de servicios de negocio es necesario contemplar los siguientes principios:

- La lógica de negocio debe estar totalmente desacoplada de la de presentación. Una operación de presentación no debe definir ningún flujo de procesos de negocio.
- Los interfaces presentados por los componentes que implementan la lógica de negocio deben estar perfectamente definidos y tener entidad funcional para facilitar su reutilización
- Estos interfaces deben estar orientados hacia la función de negocio y no presentar una granularidad muy detallada, ya que acoplaría la capa de presentación con la de negocio y conduciría a implementar la lógica de negocio o el flujo del proceso de forma externa al mismo. Además, pensando en que pueden ser accedidos de forma remota, un interface muy granular podría producir problemas de rendimiento y sobrecarga de la red.
- Los datos de negocio nunca deben exponerse directamente a ninguno de los canales de acceso.

5.1.2. Patrón Fachada

El patrón de diseño de **Facade** permite encapsular la complejidad de las interacciones entre los datos/objetos de negocio que participan en un proceso. Parece evidente que todos los métodos de esta capa no pueden residir en una misma clase (fachada) por cuestiones de legibilidad. Por ello habitualmente se dividen en varias clases. No existen reglas a priori sobre cómo agrupar dicha funcionalidad. Es algo que se debe decidir en cada aplicación. Generalmente dicha separación responde a agrupación de casos de uso relacionados. Otra forma de verlo es estableciendo una correspondencia con grupos de opciones en los menús de la aplicación. UDA genera servicios que gestionan entidades relacionadas, por ello, las fachadas de los servicios están orientadas a entidades, proporcionando los métodos necesarios para añadir, borrar, modificar y buscar entidades relacionadas. Es decir, UDA proporciona los métodos CRUD (Create, Read, Update, Delete) para cualquier entidad y sus relaciones.

```
package com.ejje.x21b.service;

import com.ejje.x38.dto.Pagination;
import java.util.ArrayList;
import java.util.List;
import com.ejje.x21b.model.Vendor;

public interface VendorService {

    Vendor add(Vendor vendor);

    Vendor update(Vendor vendor);

    Vendor find(Vendor vendor);

    List<Vendor> findAll(Vendor vendor, Pagination pagination);

    Long findAllCount(Vendor vendor);

    List<Vendor> findAll(Vendor vendor, Pagination pagination, Boolean
startsWith);

    void remove(Vendor vendor);

    void removeMultiple(ArrayList<Vendor> vendorList);

    void addVendorpayment(Vendor vendor);

    void removeVendorpayment(Vendor vendor);
}
```

Las aplicaciones deberán construir sus propios Servicios con métodos donde se implemente la lógica de negocio y estos a su vez podrán invocar (o no) a los métodos creados por UDA, que a su vez son modificables o extensibles.

La elección correcta es aquella que permita a los usuarios de los servicios de negocio encontrar el método que buscan fácilmente.

Generalmente los métodos significativos de las clases que implementan la capa de servicios (o business logic layer) harán cierta orquestación con los elementos de la capa de acceso a datos. Para entendernos

implementarán algoritmos que involucren una o varias invocaciones a métodos expuestos en los DAOs de la aplicación.

Estos métodos siempre serán ‘coarse grained’, o lo que es lo mismo, transaccionales y no contendrán estado, es decir, serán stateless.

Lo que sucede detrás de las llamadas a la capa de servicios es **transparente** para el cliente que desconoce el algoritmo interno y las tecnologías que hay detrás. Además estos métodos funcionan de forma **transaccional**, si se produce un error automáticamente se hará un **rollback** de forma que el estado de los datos sea siempre consistente. Es **stateless** porque no guarda ningún estado, simplemente recoge los parámetros, ejecuta el algoritmo aplicando reglas y accediendo a la capa de datos y devuelve un resultado sin dejar en memoria ningún dato.

5.2 Implementación de la capa de servicios

Como se indicaba en el punto anterior la implementación de la capa de servicios es extremadamente sencilla gracias a Spring. Consiste en crear dos simples POJOs (interface e implementación) por cada servicio en los que se indica mediante anotaciones el comportamiento transaccional deseado. Spring aportará la instanciación, las referencias a los componentes de la capa de acceso a datos y el comportamiento transaccional mediante las mencionadas anotaciones.

A continuación se muestra un extracto de las clases que implementan el servicio que implementa los dos ejemplos planteados en el punto anterior. No es importante entender la lógica que hay detrás de la implementación, simplemente hay que observar 3 aspectos.

1. Se trata de simples POJOs no hay herencia ni implementación de otra interface que la del propio servicio.
2. No hay lookups, ni factorías, para acceder a los componentes de la capa de acceso a datos. En su lugar Spring se encarga de instanciar los objetos e inyectar sus referencias allá donde haga falta. Estas inyecciones se declaran con la anotación `@Autowired` o en el fichero XML de la capa de servicio (service-config.xml). Además, hace falta el setter de la propiedad.

```
@Autowired
private VendorDao vendorDao;

<bean id="vendorPartService"
class="com.ejje.x21a.service.VendorPartServiceImpl">
    <property name="vendorPartDao" ref="vendorPartDao" />
</bean>

public void setVendorDao(VendorDao vendorDao) {
    this.vendorDao = vendorDao;
}
```

3. Las anotaciones `@Transactional` sobre los métodos del servicio indican los valores aplicados en cada método. Los métodos de lectura (find*) suelen ser `readOnly`, mientras que al resto se le aplican los valores por defecto.

Por defecto `@Transactional` toma los valores `propagation="required"` `readOnly="false"` `isolation="DEFAULT"`, que son los valores asociados en la mayoría de los casos para operaciones de modificación. Para funciones tipo finders el valor `readOnly="true"` es el recomendado para optimizar el rendimiento.

5.3 Componentes y frameworks de UDA en la capa de Servicios

UDA es en muchos casos es normativa y asistentes en lugar de componentes a medida.

Las normas de uso de UDA asociadas a la capa de servicios son:

- Se utiliza Spring para instanciar los componentes de la capa de Servicios declarados en el fichero **service-config.xml** o bien anotados con `@Service(value="nombreService")`.
- Se inyectan los componentes de la capa de acceso a datos y se hace mediante properties (setters) como se muestra en el ejemplo anterior.
- Se utilizan las capacidades **transaccionales** de Spring y se hace **mediante anotaciones**.

Transparentemente para el usuario **las llamadas a la capa de servicio son logueadas** mediante aspectos AOP de UDA, en esta ocasión sí, hechos a medida y utilizando las capacidades de AOP de Spring. De la misma forma las **excepciones no capturadas son logueadas por UDA**.

5.4 Acceso a los servicios.

Si para hacer referencia a los componentes de la capa de acceso a datos utilizamos Dependency Injection exactamente lo mismo se hace para acceder a los servicios en este caso desde la capa de presentación o más concretamente desde la capa de control. El fichero que recoge esta configuración no es otro que *mvc-config.xml* que se muestra a continuación en el que podemos observar como los beans de control hacen referencia a ciertos servicios, que están declarados en el fichero *service-config.xml*.

```
<bean id="lineitemController"
class="com.ejje.app.control.LineitemController">
    <property name="lineitemService" ref="lineitemService" />
</bean>
```

De forma paralela, existe la posibilidad de inyectar dependencias con anotaciones, prescindiendo de XML.

Para ello, se utiliza la anotación `@Autowired` en la capa de control.

```
@Autowired
private LineitemService lineitemService;
```

5.5 Invocación de EJBs sin estado.

Actualmente Spring ofrece un tag específico para acceder a EJBs. Dicho tag se encuentra en el namespace `jee` y su identificador es `remote-slsb`. Su uso es realmente sencillo, como se muestra a continuación.

```
<jee:remote-slsb id="productServiceStubRemote" business-
interface="com.ejje.x21a.remoting.ProductServiceStubRemote" jndi-
name="ejb.stateless.ProductServiceStub#com.ejje.x21a.remoting.ProductServiceStubRemote"
lookup-home-on-startup="true" refresh-home-on-connect-failure="true" environment-
ref="appConfiguration" />
```

Esta etiqueta es generada por el asistente de creación de un nuevo cliente EJB de UDA. La configuración asociada al bean se representa con una utilidad de Spring, que se alimenta del fichero de configuración de la

aplicación “xxx.properties” que está en el proyecto xxxConfig generado por UDA, donde xxx representa al código de aplicación.

Esta técnica permite crear un representante de la interface @Remote del EJB en el ApplicationContext de Spring. En este caso, el Bean productServiceStubRemote se inyectaría en un servicio de la siguiente manera:

```
<bean id="orderingService" class="com.ejje.x21a.service.OrderingServiceImpl">  
  <property name="productServiceStubRemote" ref="productServiceStubRemote" />  
</bean>
```

En el ejemplo anterior, se inyecta un EJB local (Stub) a través de su interface remota. Esta inyección también podría darse a través de anotaciones:

```
@Autowired  
private ProductServiceStubRemote productServiceStubRemote;
```

6 Acceso a Datos

En este apartado se detalla todo lo referente a la capa de acceso a datos. En UDA, a día de hoy, existen dos posibles ramas en lo que a persistencia se refiere, que se distinguen por la tecnología utilizada a la hora de trabajar contra los sistemas gestores de bases de datos. Las tecnologías disponibles son Spring JDBC y JPA 2.0. Para ambos casos, UDA se encarga de generar todo el código necesario que gestiona automáticamente la información que contiene el esquema relacional de base de datos.

También, se indica cómo obtener las conexiones a bases de datos desde las aplicaciones y se explican detalladamente las estrategias de lectura/escritura masiva de datos utilizadas en cada tecnología de persistencia.

6.1 Estrategias y convenciones

Aunque UDA soporte dos tipos de motores de persistencia (por el momento), a priori muy diferentes, se siguen ciertas pautas o estrategias de manera que el comportamiento de la capa de acceso a datos resulta similar y homogéneo (en la medida de lo posible) en todos los casos.

6.1.1. Query by example (QBE)

El mecanismo de búsqueda de consulta según ejemplo (QBE) busca objetos en un registro (base de datos) mediante un objeto existente.

Para usar QBE, la aplicación crea una instancia de un objeto principal (una entidad que representa a una tabla de la base de datos) y establece las propiedades en el objeto. Este objeto se convierte en el filtro a aplicar en la consulta. El método de búsqueda obtiene del dominio especificado (tabla) objetos con propiedades idénticas a las establecidas en el filtro de consulta. Dicho método devuelve todos los objetos que coinciden con el objeto suministrado.

La consulta creada por QBE produce un AND lógico de todas las propiedades establecidas en el objeto principal. Cuando los atributos tienen varios valores, la consulta resultante es un AND lógico de todos los valores establecidos para el atributo.

QBE sólo admite las propiedades no referenciales. Si el objeto principal que se utiliza en la consulta contiene valores de propiedades que son otros objetos principales (o colecciones de estos), estos últimos no formarán parte de los parámetros de filtrado.

Por ejemplo, considere un objeto “Vendor” con las propiedades “phone” y “address” establecidas. Como estas propiedades no son referenciales, es decir, que no son otros objetos principales y no están vinculadas a otros objetos principales, serán aplicadas como filtro. Si el objeto “Vendor” agrega un miembro como una colección de objetos “Payment”, dicho miembro no será tenido en cuenta por el mecanismo de filtrado.

6.2 Object Relational Mapping (ORM)

ORM es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y el utilizado en una base de datos relacional, utilizando un motor de persistencia.

En la práctica esto crea una base de datos orientada a objetos virtual, sobre la base de datos relacional. Esto posibilita el uso de las características propias de la orientación a objetos (básicamente herencia y polimorfismo). Hay paquetes comerciales y de uso libre disponibles que desarrollan el mapeo relacional de objetos (por ejemplo, JPA), aunque también se pueden crear herramientas ORM propias basadas en JDBC. UDA proporciona ambas alternativas, generando automáticamente el código en todos los casos.

El principal problema surge porque hoy en día, prácticamente todas las aplicaciones están diseñadas para usar la Orientación a Objetos (POO), mientras que las bases de datos más extendidas son del tipo relacional.

Las bases de datos relacionales solo permiten guardar tipos de **datos primitivos** (enteros, cadenas de texto, etc...) por lo que no se puede guardar de forma directa los objetos de la aplicación en las tablas, sino que estos se deben de convertir antes en registros, que por lo general afectan a varias tablas. En el momento de volver a recuperar los datos, hay que hacer el proceso contrario, se deben **convertir** los registros en objetos.

Es entonces cuando ORM cobra importancia, ya que se encarga de forma **automática** de convertir los objetos en registros y viceversa, simulando así tener una base de datos orientada a objetos.

Gracias a todo ello, el desarrollador se concentrará más en codificar la lógica de negocio, que en programar operaciones contra la base de datos. Además el ORM reduce la cantidad de código, ya que permite centralizar los procesos de búsqueda de datos en la base de datos, liberando al desarrollador de escribir consultas ad-hoc innecesarias en el código. Sin mencionar que, también gestionará el pool de conexiones a la base de datos.

El uso de un ORM aporta las siguientes ventajas:

- Rapidez de desarrollo: La mayoría de las herramientas ORM disponibles, permiten la creación del modelo de datos y la capa de acceso a datos a partir del esquema de la base de datos, es decir, el desarrollador crea la base de datos y la herramienta **automáticamente** lee el esquema de tablas y relaciones, creando una infraestructura de persistencia ajustada.

Esto evita mucho tiempo de desarrollo repetitivo y evita posibles errores humanos.

Por otro lado, una vez que se tiene la infraestructura de persistencia creada, el desarrollador avanza mucho más rápido gracias a la metodología totalmente orientada a objetos, olvidándose así de las consultas a la base de datos.

- Abstracción del motor de base de datos: El sistema ORM puede generar de forma automática las consultas a la base de datos para convertir los registros en objetos (y viceversa) y éstas deben poder adaptarse a los **distintos proveedores** (MySQL, Oracle, PostgreSQL, etc...).

Esto permitirá a los desarrolladores tener una preocupación menos a la hora de comenzar un proyecto, ya que tienen la seguridad de que el impacto que supone un cambio drástico en el proveedor de base de datos, será suavizado por el ORM.

- API propio para la consulta a la base de datos: Los ORMs pueden proporcionar utilidades para abstraer al desarrollador del uso de SQL a la hora de extraer de diferente manera los datos (ordenaciones, filtros, agrupaciones,...). UDA delega en el API de JPA para esto, aunque en proyectos de tipo JDBC no proporciona ningún API adicional al respecto.

Desventajas de los ORMs:

- Curva de aprendizaje: Las herramientas ORM suelen ser muy amplias, por lo que llegar a explotar su máximo rendimiento costará algo de tiempo. Para proyectos complejos, se recomienda empezar por comprender el funcionamiento del ORM, para no sufrir las consecuencias de un mal diseño derivado del desconocimiento de la herramienta.
- Menor rendimiento: Está claro que tener una capa entre el código y el sistema gestor de base de datos, penaliza mínimamente el rendimiento de la aplicación. Pongamos el ejemplo para una simple query para lo que el sistema tendrá que: convertir la consulta al SQL del proveedor usado; enviar la consulta; leer registros, limpiarlos y convertirlos a objetos.
- Sistemas complejos: Normalmente la utilidad de ORM desciende con la mayor complejidad del sistema relacional. Es decir, si se tiene una base de datos compleja, el manejo del ORM también se hará más complejo. UDA salva en gran medida este escoyo, ya que genera código ajustado a cualquier esquema relacional. Adicionalmente, se dispone de su rama JDBC por si se desea disponer de código a más bajo nivel.

6.3 Estrategia de carga de datos

En los mapeos ORM (Object Relational Mapping), las relaciones entre objetos se pueden especificar como Eager o Lazy.

Cuando una relación se establece como Eager (póngase como ejemplo la relación de 1:N entre la entidad “Vendor” y la entidad “VendorPart”, que en realidad no es Eager, tal y como se explica más adelante), al recuperar un objeto del tipo “Vendor” esta arrastrará además una colección completa de objetos “VendorPart” asociados.

Como la cantidad de objetos “VendorPart” puede ser muy grande, obliga a mantener una gran cantidad de objetos en memoria, cosa que resulta costosa. En casos en los que al cliente solicitante sólo le interesen los atributos propios de “Vendor” y no le interese consultar la colección de “VendorPart”, debería ser innecesario pagar este costo.

Por eso, es común que en las relaciones *ToMany (1:N o M:N) el motor de persistencia implemente el patrón LazyLoad. Este patrón permite retrasar la carga de la colección dependiente hasta el momento en que sea imperativamente requerida. Si la relación entre “Vendor” y “VendorPart” se establece como Lazy (cosa que sucede en realidad), el cliente no paga el costo de la carga masiva.

Sin embargo, el patrón Lazy tiene otros costes que repercuten en el cliente. Cuando se necesita acceder a un objeto de la colección dependiente, el ORM lanza una nueva consulta a base de datos y establece los datos obtenidos en el objeto al que se pretendía acceder.

En JPA, la carga de los objetos de la colección dependiente sucede de manera totalmente transparente gracias a la infraestructura AOP propia de JPA.

```
Vendor vendor = this.vendorDao.find(vendorFilter);  
VendorPart vendorPart = vendor.getVendorParts().get(0);
```

En el ejemplo anterior, se dan dos consultas a base de datos, una de ellas explícita (llamando al DAO) y otra implícita realizada por JPA al acceder a un objeto de una colección Lazy.

En JDBC, la propia lógica de negocio se tiene que encargar de invocar a la capa de acceso a datos para cargar el objeto correspondiente de manera explícita, usando el objeto “Pagination” (que se explica en el siguiente capítulo) para finar dicha búsqueda y no sobrecargar la memoria con N objetos.

```
Vendor vendor = this.vendorDao.find(vendorFilter);  
VendorPart vendorPartFilter = new VendorPart();  
Pagination pagination = new Pagination();  
pagination.setRows(new Long(1));  
vendorPartFilter.setVendor(vendor);  
VendorPart vendorPart=vendorPartDao.findAll(vendorPartFilter, pagination).get(0);
```

Como conclusión, UDA establece toda relación *ToOne (M:1 y 1:1) como Eager y toda relación *ToMany (1:N y M:N) como Lazy.

A modo de consideración general, hay que tener en cuenta que los métodos de lectura que genera UDA (finders) obtienen todas las columnas de las tablas. Si se quieren obtener solo determinadas columnas, habrá que modificar los finders o crear nuevos que solo pregunten por ciertas columnas.

6.4 Paginación

La paginación en el entorno UDA se obtiene utilizando el objeto “Pagination”, el cual es un parámetro que se proporciona a los métodos de tipo finder que implementan el filtrado.

```
public List<Vendor> findAll(Vendor vendor, Pagination pagination) {...}
```

El objeto “Pagination” es una clase java que posibilita la ordenación y filtro de número de registros sobre una query. Dicho objeto contiene las siguientes propiedades que posibilitan esa funcionalidad:

```
public class Pagination implements java.io.Serializable{  
  
    private Long rows;  
    private Long page;  
    private String ascDsc;  
    private String sort;  
}
```

- “rows”: Número de registros a recuperar.
- “page”: Número de página a recuperar.
- “ascDsc”: Indicador de orden Asc (ascendente) o Dsc (descendente).
- “sort”: Nombre del campo referente a la propiedad del modelo de datos por la cual se va a realizar la ordenación. En caso de ser más de uno, deben ir separados por comas.

A nivel de DAO, para aplicar la paginación, es necesario que el objeto “Pagination” no sea nulo. En caso contrario, la consulta devolverá todos los resultados obtenidos, pudiendo provocar así una gran carga en memoria y en red.

El objeto “Pagination” está compuesto por dos subgrupos, siendo el primero el encargado de la recuperación del número de registros que afectan a la query, mientras que el segundo afecta al orden de las tuplas resultantes. Ambos módulos son independientes, pudiendo ordenar el filtrado de datos únicamente por uno de ellos.

- Ordenación de registros: Las propiedades “ascDsc” y “sort” del objeto son las responsables de esta ordenación. No puede existir una sin la otra. En caso de que uno de los dos campos este vacío no se aplicará ninguna ordenación a la query.
- Filtrado de número de registros y posición de los mismos: Las propiedades “rows” y “page” del objeto son las responsables de filtrar el número y posición de los objetos. Para poder reducir el número de registros a recuperar, es obligatorio que el campo rows esté alimentado, de lo contrario se recuperarán tantos registros como existan. Sin embargo, la propiedad page, indica el número de página en la que se encuentra el usuario. Esta segunda es opcional. En caso de que no esté alimentado dicho parámetro, siempre se devolverán las primeras N tuplas.

6.5 Estrategia de generación de código. Tablas, sinónimos y entidades.

La estrategia de generación de código en UDA tiene como base modelos normalizados de base de datos. En base a esta característica del modelo relacional, UDA obedece a la estrategia de generación de código que se describe a continuación:

- La generación de código no detecta las vistas. Es decir, las vistas no implican generación de código en ningún caso. En vez de utilizar vistas, se deberá disponer de un modelo relacional correctamente normalizado.
- El modelo relacional de base de datos se traduce a entidades Java relacionadas. Es decir, se genera una especie de ORM orientado a objeto, donde el desarrollador no tiene porque ser consciente de las tablas subyacentes.
- Las relaciones OneToOne son detectadas por la generación de código, generando así su relación correspondiente en cada entidad.
- La tabla intermedia que existe en las relaciones ManyToMany no será representada por entidad alguna. Para gestionar dichas tablas se deberá utilizar cualquier entidad padre de la relación. Ambas

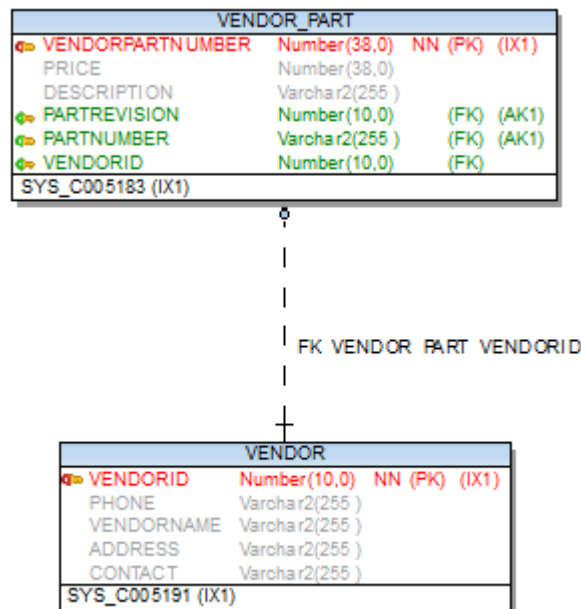
entidades padre tendrán los métodos necesarios para el mantenimiento y consulta de la tabla intermedia.

- Al generar el código para una tabla en concreto, en caso de que esta disponga de sinónimos, el nombre de la entidad cogerá el nombre de este, en vez del nombre de la tabla. En el siguiente capítulo se explica con más profundidad este aspecto.
- En caso de estar trabajando con un proyecto JPA, al generar código en el mismo, si la entidad dispone de una clave primaria compuesta, se generará una entidad <<nombreEntidad>>Id, que representará la composición de dicha clave.

6.5.1. Modelo relacional

Partiendo del esquema relacional de base de datos, UDA genera un modelo de datos normalizado. Se ha optado por la generación orientada a objetos (entidades), en vez de la generación orientada a tabla. De tal manera, que el esquema de base de datos y sus relaciones quedarán plasmadas en el modelo de datos, resultando así más legible y comprensible para el desarrollador.

A continuación se observa la composición de la relación de las tablas “Vendor_part” y “Vendor”:



En un modelo orientado a tabla, el modelo se compondría de la siguiente manera (únicamente quedarían plasmadas las columnas de la tabla y sus tipos):

```

public class VendorPart implements java.io.Serializable {
    private static final long serialVersionUID = 1L;

    private BigDecimal vendorpartnumber;
    private BigDecimal vendorId;
    private String partNumber;
    private BigDecimal partRevision;
    private BigDecimal price;
  
```

```
private String description;
```

Mientras que la generación de código proporcionada por UDA vincula directamente las entidades:

```
public class VendorPart implements java.io.Serializable {

    private static final long serialVersionUID = 1L;

    private BigDecimal vendorpartnumber;
    private Vendor vendor;
    private Part part;
    private BigDecimal price;
    private String description;

    private List<Lineitemprivateido> lineitems = new
    ArrayList<Lineitemprivateido>();
```

6.5.2. Sinónimos

A la hora de generar código, al recuperar de nombres de tabla en base de datos, se sigue la siguiente estrategia:

1. En caso de que la tabla disponga de sinónimos privados (siempre que el sinónimo creado sea del esquema con el que se ha realizado el login), la entidad generada basará su nombre en dicho sinónimo. La creación de un sinónimo privado se realiza de la siguiente manera:

```
CREATE SYNONYM << Esquema propietario sinónimo privado >>.<<nombreSinonimo>>
FOR << EsquemaPropietarioTabla >>.<<nombreTabla>>
```

2. En caso de no disponer de sinónimos privados, se procederá a verificar la existencia de sinónimos públicos para dicha tabla. En caso de existir alguno, la entidad generada basará su nombre en dicho sinónimo. La creación de un sinónimo público se realiza de la siguiente manera:

```
CREATE PUBLIC SYNONYM <<nombreSinonimoPublico>> FOR
<<EsquemaPropietarioTabla>>.<<nombreTabla >>
```

3. En caso de no disponer de ningún tipo de sinónimo, el nombre de la tabla servirá como base para la entidad.

6.6 Demarcación transaccional

En UDA, la demarcación transaccional de los métodos se establece a nivel de servicios de negocio. Sin embargo, la capa de acceso a datos es la responsable en última instancia de propagar dichas transacciones hasta el driver de base de datos, o modificar su comportamiento.

Toda transacción que llegue hasta la capa de acceso a datos será propagada tal cual, excepto en los casos en los que se invoque a métodos de sólo lectura (o finders), donde la transacción se propagará como Read-Only, con motivo de mejorar el rendimiento y no arrastrar sobrecarga operacional innecesaria al driver de base de datos.

```
@Transactional (readOnly = true)
public List<Vendor> findAll(Vendor vendor, Pagination pagination) {...}
```

6.7 Java Database Connectivity (JDBC)

El generador de código es suficientemente inteligente para detectar automáticamente la naturaleza del proyecto. En base a eso se genera un conjunto de métodos por entidad, los cuales pueden variar en función de las características del modelo de base de datos.

6.7.1 Comportamientos específicos Spring JDBC

La estrategia de acceso a datos en proyectos JDBC, en base al modelo de base de datos, es la siguiente:

6.7.1.1. Clave primaria compuesta

En caso de que la clave primaria de la entidad a tratar sea compuesta, se generará una propiedad por columna existente en la tabla, siempre respetando un modelo orientado a objetos. En el caso de la tabla "LineItem", en la cual se ve que la clave primaria está compuesta por los campos "Itemid" y "Orderid" (esta última a su vez es la relación con la entidad "Ordering"):

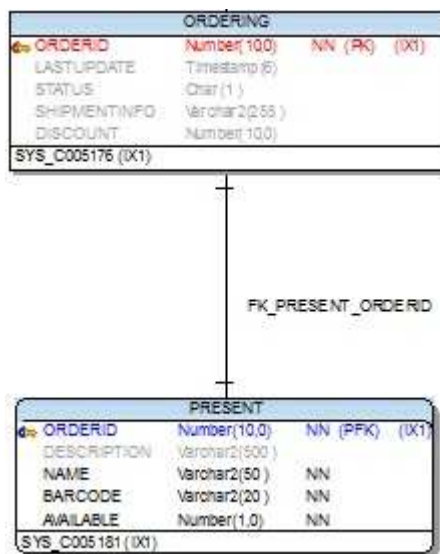
| LINEITEM | | |
|------------------|--------------|---------------|
| QUANTITY | Number(10,0) | |
| ITEMID | Number(10,0) | NN (FK) (X1) |
| ORDERID | Number(10,0) | NN (PFK) (X1) |
| VENDORPARTNUMBER | Number(38,0) | (FK) |
| SYS_C005189 (X1) | | |

En JDBC, la representación de la clave primaria queda de la siguiente manera:

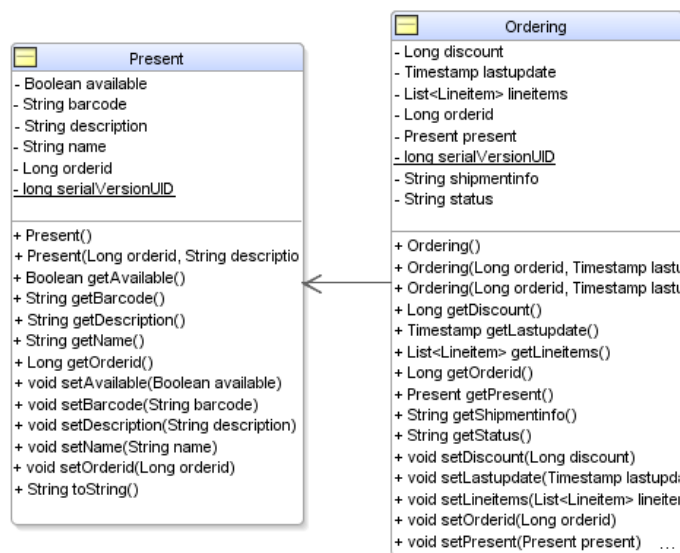
```
public class Lineitem implements java.io.Serializable {
    private static final long serialVersionUID = 1L;
    //Clave compuesta
    private Long itemid;
    private Ordering ordering;
```

6.7.1.2. Relación OneToOne

Las relaciones OneToOne en JDBC, se reflejan exclusivamente en la parte propietaria de la relación. Ambas entidades se relacionan manteniendo la misma Primary Key en las tuplas afectadas. A continuación se muestran las entidades "Ordering" y "Present", donde existe una relación OneToOne entre ambas:



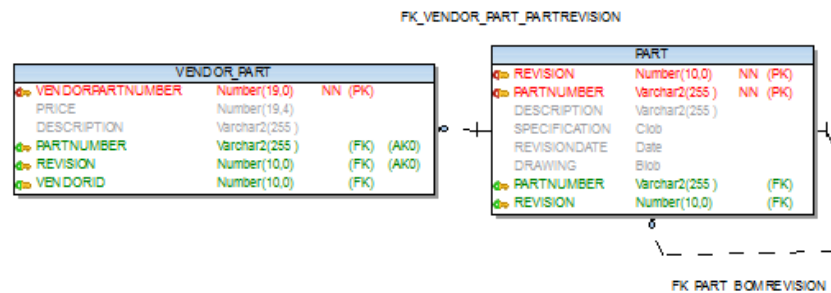
Las entidades generadas por UDA, tienen el siguiente aspecto:



En el caso de la entidad “Ordering”, los DAOs generados tendrán un comportamiento Eager, respecto a la entidad “Present”. Es decir, al recuperar la entidad “Ordering”, recuperaremos también la tupla de “Present” afectada por la query, debido a que únicamente supone una tupla. En el caso de la entidad “Present”, sin embargo, no se recuperará ninguna tupla al no existir relación en el modelo.

6.7.1.3. Relación ManyToOne

Las relaciones ManyToOne en JDBC, como puede ser el caso entre “Vendor_Part” y “Part”, siguen un comportamiento Eager. El modelo de datos en este caso es:



En la imagen se observa que la tabla “Vendor_Part” dispone de una relación ManyToOne con la tabla “Part”, la cual a su vez dispone de una relación ManyToOne contra ella misma. El modelo de datos generado por UDA es el siguiente:

```
public class VendorPart implements java.io.Serializable {
...
    private Part part;
...
}
```

```
public class Part implements java.io.Serializable {
    private static final long serialVersionUID = 1L;
    //Clave compuesta
...
    private Part part;
...
    private List<VendorPart> vendorParts = new ArrayList<VendorPart>();
}
```

Al realizar una consulta sobre la tabla “Vendor_Part”, se obtienen dos niveles de profundidad. Es decir, se obtiene la tabla “Vendor_Part”, y por cada tupla obtenida, se obtiene a su vez la tupla en la tabla “Part”. En función del modelaje de base de datos, podría llegar a ser infinita la recuperación de registros. De tal manera, que las consultas se han limitado a dos niveles de profundidad.

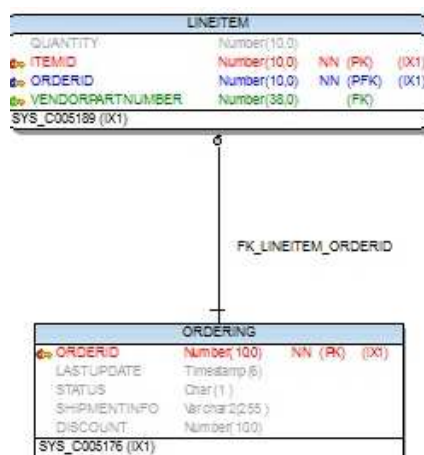
Sin embargo, también se rellena la clave primaria del tercer nivel, evitando así una consulta innecesaria en caso de volver a consultar datos del tercer nivel.

Sin embargo, si se realiza la consulta sobre la tabla “Part”, debido a que las tuplas obtenidas pueden estar relacionadas con varias tuplas de la tabla “Vendor_Part” (o visto de otra manera, “Part” contiene una colección de entidades “VendorPart”), no se recuperarán los datos de “Vendor_Part”, implementando así la estrategia Lazy Fetching.

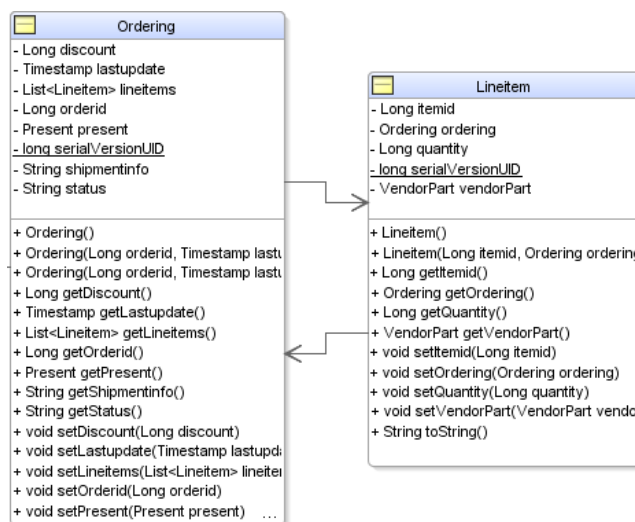
Esto significa que para obtener las entidades “VendorPart” relacionadas con una entidad “Part” en concreto, hay que hacer uso del DAO correspondiente a la parte fuerte de la relación (en este caso sería VendorPartDaoImpl).

6.7.1.4. Relación OneToMany

Las relaciones OneToMany en JDBC, como puede ser el caso entre “Ordering” y “LineItem”, siguen un comportamiento Lazy. Es decir, obtener la entidad padre, no implica la recuperación de los registros del hijo. En el siguiente modelo de datos, “LineItem” es la parte fuerte de la relación, mientras que “Ordering” es la parte débil.



El modelo de datos generado por UDA tiene el siguiente aspecto:



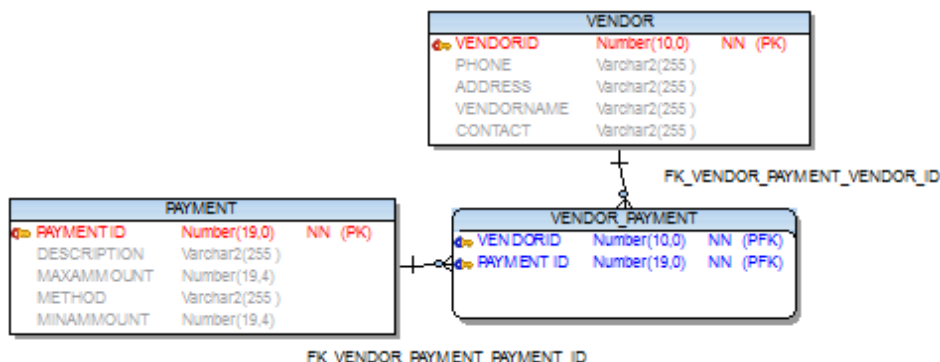
Al obtener registros de la tabla “Ordering”, ningún registro de la tabla “Lineitem” será recuperado.

Para poder obtener los registros “Ordering” y “Present” conjuntamente, se debería crear en la capa de servicios de negocio un método de este tipo:

```
public Ordering findOrderingLineitem(Ordering ordering, Pagination pagination)
{
    Ordering orden = orderingDao.find(ordering);
    Lineitem lit = new Lineitem();
    lit.setOrdering(new Ordering());
    lit.getOrdering().setOrderid(ordering.getOrderid());
    List<Lineitem> listalit = this.lineitemDao.findAll(
        lit, pagination);
    orden.setLineitems(listalit);
    return orden;
}
```

6.7.1.5. Relación ManyToMany

Para la detección de una relación ManyToMany, la tabla relacional intermedia (la M:N), se deberá componer únicamente de las claves primarias de sus entidades padres. Todos estos campos deberán ser clave primaria a su vez en la tabla relación. Estas tablas no generan entidad, y la manera de gestionarlas es siempre desde sus entidades padre. En el siguiente caso, las tablas “Vendor” y “Payment”, poseen una relación ManyToMany tal y como se muestra a continuación:



Al no tener entidad intermedia, la declaración en el modelo queda de la siguiente manera:

```
public class Vendor implements java.io.Serializable {
    ...
    private List<Payment> payments = new ArrayList<Payment>();
    ...
}
```

```
public class Payment implements java.io.Serializable {
    ...
    private List<Vendor> vendors = new ArrayList<Vendor>();
    ...
}
```

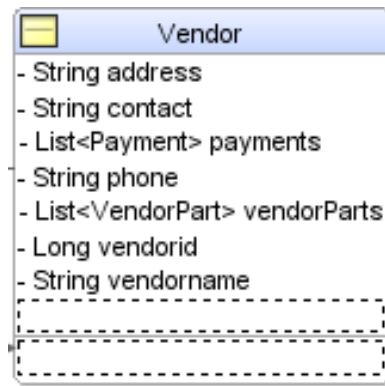
Este tipo de relación tiene un comportamiento Lazy. No existe parte fuerte en la relación. Para la gestión de registros relacionados, UDA genera métodos específicos en sus entidades padre.

```
public Vendor addvendorPayment(Vendor vendor) {...}
public void removevendorPayment(Vendor vendor) {...}
public Vendor findvendorPayment(Vendor vendor, Payment payment,
    Pagination pagination) {...}
public Long findvendorPaymentCount(Vendor vendor, Payment payment) {...}
```

6.7.2 Métodos generados por UDA

UDA, mediante su utilidad de generación de código, genera, a nivel de DAO, una serie de métodos útiles para el usuario. Por cada entidad, se generan los métodos CRUD necesarios su mantenimiento.

A continuación se explican los métodos generados para la gestión de la entidad “Vendor”.



6.7.2.1 add

El método “add”, inserta registros en la tabla correspondiente a la entidad. Este método recibe como parámetro un objeto entidad (“Vendor” en este caso). En este método no entra ninguna relación en juego, es decir, cada entidad solo inserta en su tabla, nunca en las tablas relacionadas.

```
Vendor add(Vendor vendor);
```

6.7.2.2 update

El método “update”, actualiza registros en la tabla correspondiente a la entidad. El método recibe como parámetro un objeto entidad (“Vendor” en este caso), en el cual estarán reflejados los nuevos valores que adquirirá el registro correspondiente. Debido a que la actualización se realiza mediante la Primary Key de la tabla, en la entidad que se proporciona como parámetro, dicha clave a de existir previamente en la tabla.

```
Vendor update(Vendor vendor);
```

En caso de desear modificar valores de la clave primaria, habría que realizarlo mediante el método “remove” primero, y “add” posteriormente, debido a que las Primary Key no son modificables.

6.7.2.3 find

El método “find”, busca un registro en la tabla correspondiente a la entidad. El método recibe como parámetro un objeto entidad (“Vendor” en este caso), en el cual deberán estar alimentados todos los campos que componen la Primary Key. La búsqueda se realiza por Primary Key.

```
Vendor find(Vendor vendor);
```

6.7.2.4 remove

El método “remove”, elimina una única tupla de la tabla correspondiente a la entidad. El método recibe como parámetro un objeto entidad (“Vendor” en este caso), en el cual deberán estar alimentados todos los campos que componen la Primary Key. El borrado de registros se realiza por Primary Key.

```
void remove(Vendor vendor);
```

6.7.2.5 findAll

El método “findAll”, recupera una lista de registros de la tabla correspondiente a la entidad. El método recibe como parámetro un objeto entidad (“Vendor” en este caso) y un objeto “Pagination”. Este último puede ser nulo, de tal manera que se recuperarían todas las tuplas afectadas por la query. El objeto entidad introducido como parámetro, tendrá alimentados aquellos campos por los que se desea filtrar, con los valores correspondientes (es decir, se usa la estrategia Query By Example).

```
List<Vendor> findAll(Vendor vendor, Pagination pagination);
```

6.7.2.6 findAllCount

El método “findAllCount”, recupera el número total de registros de una query. El método recibe como parámetro un objeto entidad (“Vendor” en este caso), en el cual estarán rellenos aquellos campos por los que se desea filtrar (QBE). En base a ese filtro, se recuperará el número de registros afectados por la query.

```
Long findAllCount(Vendor vendor);
```

6.7.2.7 findAllLike

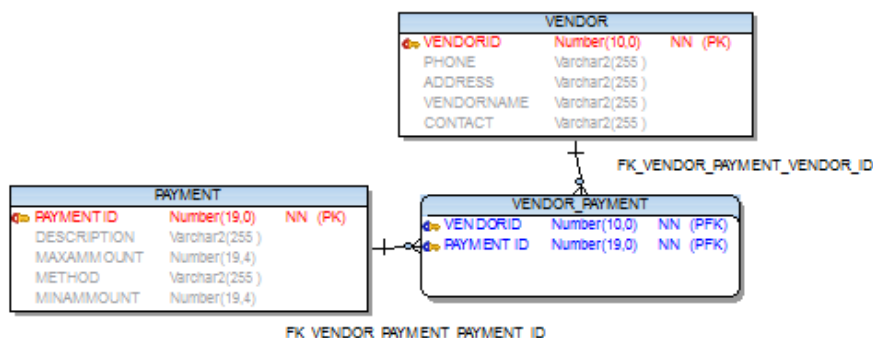
El método “findAllLike”, recupera una lista de registros de la tabla correspondiente a la entidad. El método recibe como parámetro un objeto entidad (“Vendor” en este caso), un objeto “Pagination” y un parámetro booleano denominado “startsWith”. El último puede ser nulo, de tal manera que se recuperarían todas las tuplas afectadas por la query.

El objeto entidad introducido como parámetro, tendrá alimentados aquellos campos por los que se desea filtrar, con los valores correspondientes, siendo estos parte de la consulta a obtener. Es decir, el filtro aplicado utilizará ‘Like’, tanto en números como en caracteres, en vez de “=” que utiliza el findAll.

El parámetro “startsWith”, tendrá los valores “True” o “False” en función del filtrado que se desea realizar. Cuando el objetivo de la select sea que la cadena introducida, por ejemplo, empiece por la cadena pasada como parámetro, el valor de este será “True”. Si lo que se desea es que contenga dicha cadena, empiece o no por ella, este parámetro deberá llevar el valor “False”.

```
List<Vendor> findAllLike(Vendor vendor, Pagination pagination, Boolean  
startsWith);
```

Debido a que la entidad “Vendor” se encuentra relacionada por ManyToMany con la entidad “Payment”, se generan los métodos que se explican a continuación, exclusivos para este tipo de relaciones. La tabla intermedia “Vendor_Payment”, al no disponer de entidad propia en el modelo de datos, podrá ser gestionada por ambos padres.



6.7.2.8 addxxxxyyy

El método “addxxxxyyy”, sirve para insertar registros en la tabla intermedia de una relación ManyToMany. En este caso, se insertarán registros en la tabla “Vendor_Payment”.

```
Vendor addVendorPayment(Vendor vendor);
```

Este método devuelve un objeto entidad del tipo correspondiente al DAO que gestiona la entidad (en este caso, al tratarse del VendorDaoImpl, la entidad que se devuelve será un “Vendor”). El modelo de datos que representa a “Vendor” es el siguiente:

```
public class Vendor implements java.io.Serializable {
    ...
    private List<Payment> payments = new ArrayList<Payment>();
    ...
}
```

Al invocar al método, el objeto “Vendor” que se envía como parámetro, deberá tener una colección de objetos “Payment”, concretamente en la propiedad “payments”, con los “Payment” a vincular con ese “Vendor”.

El método se recorrerá la lista de “payments” y asociará al “Vendor” indicado los registros de la tabla intermedia.

En ambas entidades, únicamente resulta necesario rellenar los campos pertenecientes a la clave primaria. Para poder realizar la inserción, deben existir todos los registros a los que las entidades están representando.

6.7.2.9 removexxyyyy

El método “removexxyyyy”, sirve para eliminar registros en la tabla intermedia de una relación ManyToMany. En este caso, se eliminarán registros en la tabla “Vendor_Payment”.

```
void removeVendorPayment(Vendor vendor);
```

Este método devuelve un objeto entidad del tipo correspondiente al DAO que gestiona la entidad (en este caso, al tratarse del VendorDaoImpl, la entidad que se devuelve será un “Vendor”). El modelo de datos que representa a “Vendor” es el siguiente:

```
public class Vendor implements java.io.Serializable {
    ...
    private List<Payment> payments = new ArrayList<Payment>();
    ...
}
```

Al invocar al método, el objeto “Vendor” que se envía como parámetro, deberá tener una colección de objetos “Payment”, concretamente en la propiedad “payments”, con los “Payment” a desvincular de ese “Vendor”.

El método se recorrerá la lista de “payments” y eliminará la asociación del “Vendor” indicado en los registros de la tabla intermedia.

En ambas entidades, únicamente resulta necesario rellenar los campos pertenecientes a la clave primaria.

En ningún caso se eliminarán los registros padres, simplemente la asociación de ambos.

Para poder realizar el borrado de la tabla M:N, los registros deberán existir previamente.

6.7.2.10 *findxxxxyy*

El método “findxxxxyy”, recupera una colección con las entidades relacionadas y las inserta en la entidad principal. Es decir, el método devuelve un objeto entidad, el cual lleva asociado una colección de entidades relacionadas por ManyToMany.

En el caso de “Vendor”, este llevará asociada una colección de entidades de tipo “Payment”, en la cual aparecerá el detalle de la tabla “Payments” para el registro seleccionado. Este método no recupera ningún registro de la entidad actual, “Vendor” en este caso.

```
Vendor findvendorPayment(Vendor vendor, Payment payment, Pagination pagination);
```

El método recibe como parámetro un objeto entidad (“Vendor” en este caso), un objeto de la entidad relacionada por ManyToMany, además de un objeto “Pagination”. Este último puede ser nulo, de tal manera que se recuperarían todas las tuplas afectadas.

El objeto de entidad principal enviado como parámetro (“Vendor” en este caso), deberá tener alimentados obligatoriamente los campos que componen la Primary Key, con los valores correspondientes, acotando así los resultados a recuperar al mínimo. La entidad relacionado por ManyToMany (“Payment” en este caso), deberá tener alimentados aquellos parámetros por los que se desea filtrar.

6.7.2.11 *findxxxxyyCount*

El método “findxxxxyyCount”, recupera el número de entidades relacionadas, aplicando determinado filtro. Es decir, el método devuelve un objeto de tipo numérico, el cual indica el número de registros de la ManyToMany, aplicando el filtrado correspondiente. En el caso de “Vendor”, se contabilizará el número de registros “Payment” asociados.

```
Long findvendorPaymentCount(Vendor vendor, Payment payment);
```

El método recibe como parámetro un objeto de entidad principal (“Vendor” en este caso) y un objeto entidad relacionada (“Payment” en este caso). El objeto entidad principal enviado como parámetro, deberá llevar obligatoriamente alimentados los campos que componen a la Primary Key, acotando así los resultados a recuperar al mínimo. La entidad relacionado por ManyToMany (“Payment” en este caso), deberá tener alimentados aquellos parámetros por los que se desea filtrar.

6.8 Java Persistence API (JPA) 2.0

El generador de código es suficientemente inteligente para detectar automáticamente la naturaleza del proyecto, JPA en este caso. En base a eso se genera un conjunto de métodos y anotaciones por entidad, los cuales pueden variar en función de las características del modelo de base de datos.

6.8.1 Comportamientos específicos JPA 2.0

La estrategia de acceso a datos en proyectos JPA, en base al modelo de base de datos, es la siguiente:

6.8.1.2 Primary Key compuesta

En caso de que la Primary Key de la entidad a tratar sea compuesta, se generará una nueva entidad cuya denominación será “NombreEntidadId” y tendrá su correspondiente DTO libre de anotaciones JPA, con denominación “NombreEntidadIdDto”. Un ejemplo claro es la tabla “LineItem”

| LINEITEM | | |
|------------------|--------------|----------|
| ITEMID | Number(10,0) | NN (PK) |
| QUANTITY | Number(10,0) | |
| ORDERID | Number(10,0) | NN (PFK) |
| VENDORPARTNUMBER | Number(19,0) | (FK) |

La entidad generada a través de la Primary Key tendrá como propiedad toda aquella columna que compone dicha clave compuesta.

```
public class LineitemIdDto implements java.io.Serializable {

    private static final long serialVersionUID = 1L;

    protected Long itemid;

    protected Long orderid;
```

Se puede observar que no se plasman las relaciones en esta clase generada.

Esta clase de tipo “NombreEntidadIdDto”, se inyecta como propiedad en el DTO principal, quedando la clase “LineitemDto”, en este caso, de la siguiente manera:

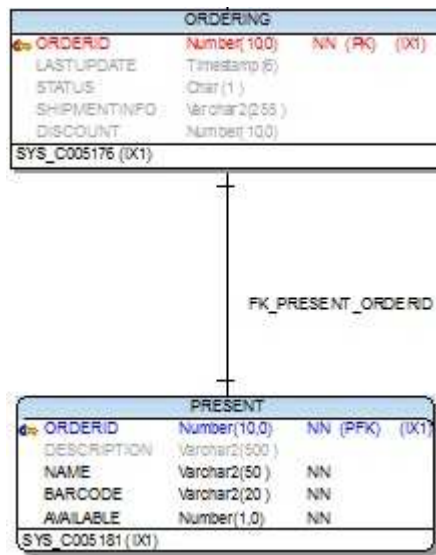
```
public class LineitemDto implements java.io.Serializable {
    private static final long serialVersionUID = 1L;
    protected LineitemId id;
    protected VendorPart vendorPart;
    protected Ordering ordering;
    protected Long quantity;
```

Se puede observar, que las relaciones, en este caso, quedan plasmadas fuera de la clave primaria, como ocurre con “Ordering”.

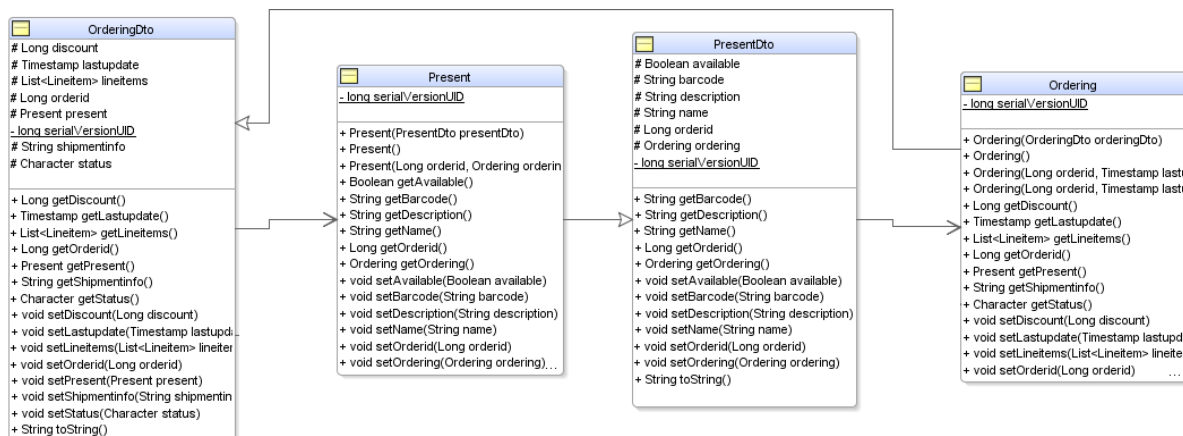
La propiedad “orderid” de la clase “LineitemIdDto”, es una propiedad básica de tipo numérico, mostrando la relación con “Ordering” en la propiedad “ordering” como una propiedad más de “LineitemDto”.

6.8.1.3 Relación OneToOne

Las relaciones OneToOne en JPA, se reflejan en las entidades de ambas partes de la relación como propiedad. A continuación se muestran las entidades “Ordering” y “Present”, donde existe una relación OneToOne entre ambas:



Las entidades generadas por UDA, tienen el siguiente aspecto:

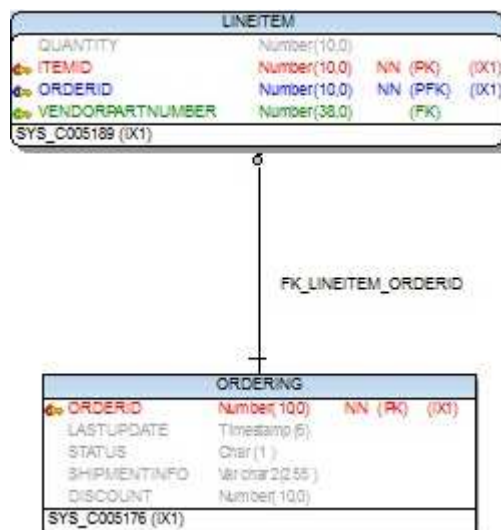


Las relaciones OneToOne en JPA tienen un comportamiento Eager, es decir, al consultar cualquiera de los dos, se recuperará a su vez la entidad relacionada, que se vincula a la propiedad correspondiente de la entidad consultada.

NOTA: Aunque una relación OneToOne sea marcada como Lazy en la anotación correspondiente (`@OneToOne(fetch = FetchType.LAZY)`), no surgirá efecto, ya que para ello habría que activar el Weaving para las entidades en juego.

6.8.1.4 Relación ManyToOne

Las relaciones ManyToOne en JPA, como puede ser el caso de "LineItem" con respecto a la entidad "Ordering", obedecen al comportamiento Eager. El modelo entidad relación para este caso es el siguiente:



En la imagen se observa que la tabla “LineItem” dispone de una relación ManyToOne con la tabla “Ordering”. El modelo de datos generado por UDA es el siguiente:

```
public class LineitemDto implements java.io.Serializable {
    . . .
    protected Ordering ordering;
    . . .
}
```

```
public class OrderingDto implements java.io.Serializable {
    . . .
    protected List<Lineitem> lineitems = new ArrayList<Lineitem>(0);
    . . .
}
```

Al realizar una consulta sobre la tabla “LineItem”, se obtiene también la tupla correspondiente de la entidad “Ordering”. Es decir, se obtiene la tabla “LineItem”, y por cada tupla obtenida, se obtiene a su vez la tupla en la tabla “Ordering”.

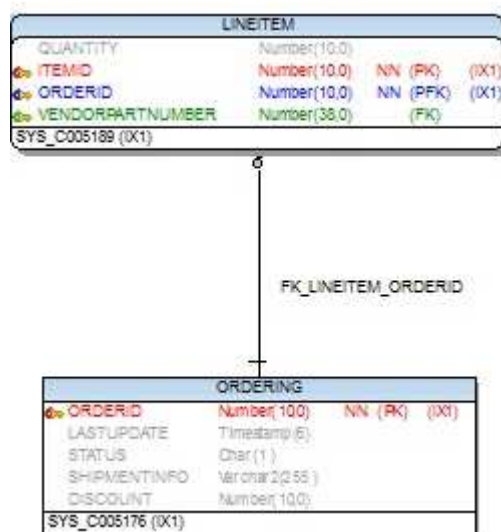
En este caso, las relaciones ManyToOne son relaciones con comportamiento Eager. La parte fuerte de la relación es “LineItem”, debido a que, al consultar esta entidad, se recuperan también la tupla de “Ordering”.

En caso de consultar la tabla “Ordering” (la parte debil), esta mantiene una relación OneToMany con la entidad “LineItem”. Este caso se explica más adelante.

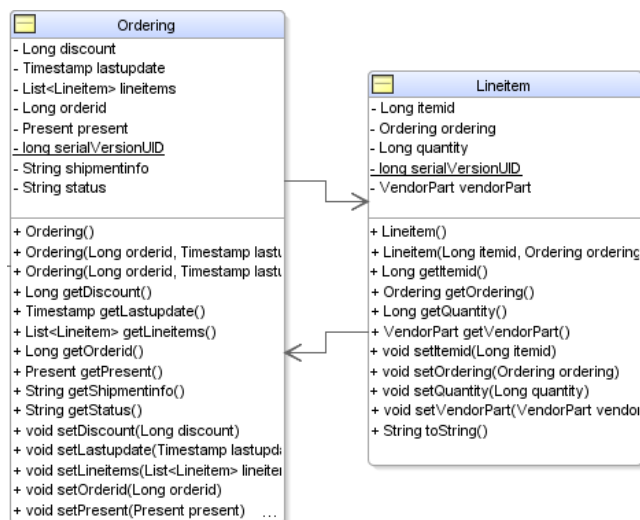
NOTA: Aunque una relación ManyToOne sea marcada como Lazy en la anotación correspondiente (`@ManyToOne(fetch = FetchType.LAZY)`), no surgirá efecto, ya que para ello habría que activar el Weaving para las entidades en juego.

6.8.1.5 Relación OneToMany

Las relaciones OneToMany en JPA, como puede ser el caso que se da entre “Ordering” y “LineItem”, obedecen a un comportamiento Lazy. Es decir, obtener la entidad padre, no implica la recuperación de los registros del hijo.



El modelo de datos generado por UDA tiene el siguiente aspecto:



El código generado para ambas entidades es el siguiente:

```
public class LineitemDto implements java.io.Serializable {
    . . .

    protected Ordering ordering;
    . . .
}
```

```
public class OrderingDto implements java.io.Serializable {
    . . .

    protected List<Lineitem> lineitems = new ArrayList<Lineitem>(0);
    . . .
}
```

En el siguiente modelo de datos, “LinItem” es la parte fuerte de la relación, debido a que obtener una tupla de ésta entidad, implica obtener la información de “Ordering” a su vez. Al contrario no sucede lo mismo.

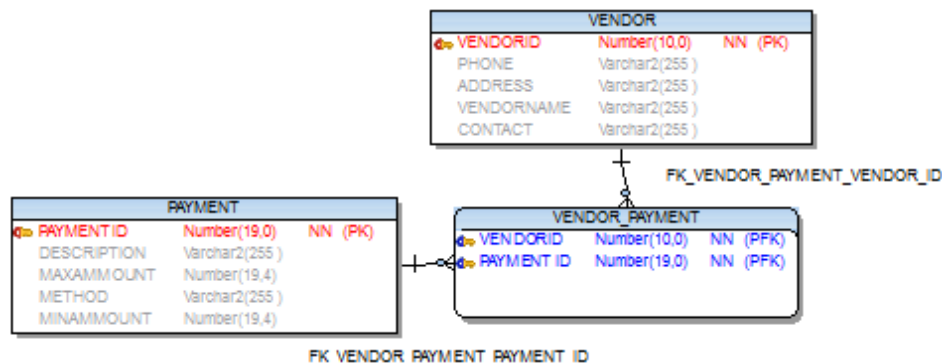
En JPA, la parte débil de la relación, “Ordering” en este caso, tiene un comportamiento Lazy Fetching. Es decir, no se obtendrán los datos de “LinItem” (parte fuerte), hasta el momento que se acceda a la propiedad correspondiente. En este caso, únicamente se acudiría a la base de datos a por los datos de “LinItem”, cuando se acceda a la entidad “Ordering” de la siguiente manera:

```
ordering.getLineItems();
```

JPA, al acceder a una propiedad anotada con Lazy Fetching (como en este acceso a “LinItem” desde “Ordering”), se encargará de forzar en background el acceso a base de datos, siendo transparente para el desarrollador.

6.8.1.6 Relación ManyToMany

Para la detección de una relación ManyToMany, la tabla relacional intermedia (la M:N), se deberá componer únicamente de las claves primarias de sus entidades padres. Todos estos campos deberán ser clave primaria a su vez en la tabla relación. Estas tablas no generan entidad, y la manera de gestionarlas es siempre desde sus entidades padre. En el siguiente caso, las tablas “Vendor” y “Payment”, poseen una relación ManyToMany tal y como se muestra a continuación:



Al no tener entidad, la declaración en el modelo queda de la siguiente manera:

```
public class VendorDto implements java.io.Serializable {
    . . .
    protected List<Paymentido> payments = new ArrayList<Paymentido>(0);
    . . .
}
```

```
public class PaymentDto implements java.io.Serializable {
    . . .
    protected List<Vendor> vendors = new ArrayList<Vendor>(0);
    . . .
}
```

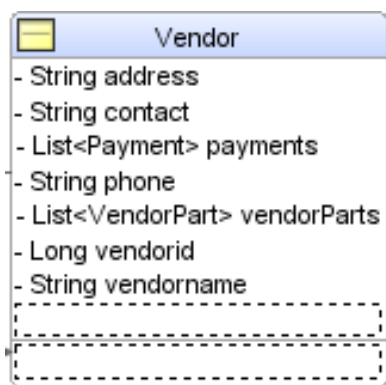
Este tipo de relación tiene un comportamiento Lazy. Para el mantenimiento de la tabla intermedia M:N, UDA proporciona los siguientes métodos, a parte de los métodos CRUD de cada entidad padre, en la capa de servicios de ambas entidades padre:

```
public Vendor addvendorPayment(Vendor vendor) {...}
public void removevendorPayment(Vendor vendor) {...}
public Vendor findvendorPayment(Vendor vendor, Payment payment,
                                Pagination pagination) {...}
public Long findvendorPaymentCount(Vendor vendor, Payment payment) {...}
private TypedQuery<Payment> queryPaginationVendorPayment(Pagination pagination,
                                                           CriteriaQuery<Payment> query, Root<Payment> paymentAux,
                                                           CriteriaBuilder cb) {...}
```

6.8.2 Métodos generados por UDA

UDA, mediante su utilidad de generación de código, genera, a nivel de DAO, una serie de métodos útiles para el usuario. Por cada entidad, se generan los métodos CRUD necesarios para el mantenimiento de cada entidad. Los métodos CRUD los proporciona un DAO genérico, denominado `GenericDaoImpl`, el cual es extendido por el DAO de cada una de las entidades. Este DAO ofrece las funcionalidades básicas, como son: add, remove, update y find.

A parte de dichos métodos, se proporcionan también una serie de métodos útiles, tales como aquellos orientados al filtrado, la paginación y la gestión de relaciones ManyToMany. Estos estarán generados en el DAO correspondiente a cada entidad. A continuación se ilustran los métodos generados para la gestión de la entidad "Vendor".



6.8.2.1 add

El método "add", inserta registros en la tabla correspondiente a la entidad. Este método lo proporciona el DAO genérico. Recibe como parámetro un objeto de la entidad a insertar ("Vendor" en este caso).

En este método no entra ninguna relación en juego, es decir, cada entidad solo inserta en su tabla, nunca en las tablas relacionadas.

```
Vendor add(Vendor vendor);
```

6.8.2.2 update

El método “update”, actualiza registros en la tabla correspondiente a la entidad. Este método lo proporciona el DAO genérico.

El método recibe como parámetro un objeto entidad (“Vendor” en este caso), en el cual estarán reflejados los nuevos valores que adquirirá el registro correspondiente. Debido a que la actualización se realiza mediante la Primary Key de la tabla, en la entidad que se proporciona como parámetro, dicha clave a de existir previamente en la tabla.

```
Vendor update(Vendor vendor);
```

En caso de desear modificar valores de la Primary Key, habría que realizarlo mediante el método “remove” primero, y *add* posteriormente, debido a que las claves primarias no son modificables

6.8.2.3 find

El método “find”, busca un registro en la tabla correspondiente a la entidad. Este método lo proporciona el DAO genérico.

Recibe como parámetro la clave primaria de la entidad a consultar (“vendorid” en este caso), siendo “NombreEntidadId” el parámetro en caso de que la entidad tenga clave primaria compuesta. Es por ello, que la consulta se realiza por clave primaria.

```
Vendor find(Long vendorid);
```

6.8.2.4 remove

El método “remove”, elimina una única tupla de la tabla correspondiente a la entidad. Este método lo proporciona el DAO genérico.

Recibe como parámetro la clave primaria de la entidad a consultar (“vendorid” en este caso), siendo “NombreEntidadId” el parámetro, en caso de que la entidad tenga clave primaria compuesta.

El borrado de registros mediante la clave primaria.

```
void remove(Long vendorid);
```

6.8.2.5 findAll

El método “findAll”, recupera una lista de registros de la tabla correspondiente a la entidad.

Recibe como parámetro un objeto entidad (“Vendor” en este caso) y un objeto “Pagination”. Este último puede ser nulo, de provocando así la recuperación de todas las tuplas afectadas por la query. El objeto entidad introducido como parámetro, tendrá alimentados aquellos campos por los que se desea filtrar, con los valores correspondientes (es decir, se usa la estrategia Query By Example).

```
List<Vendor> findAll(Vendor vendor, Pagination pagination);
```

6.8.2.6 findAllCount

El método “findAllCount”, recupera el número total de registros de una query.

El método recibe como parámetro un objeto entidad (“Vendor” en este caso), en el cual estarán rellenos aquellos campos por los que se desea filtrar. En base a ese filtro, se recuperará el número de registros afectados por la query.

```
Long findAllCount(Vendor vendor);
```

6.8.2.7 findAllLike

El método “findAllLike”, recupera una lista de registros de la tabla correspondiente a la entidad. El método recibe como parámetro un objeto entidad (“Vendor” en este caso), un objeto “Pagination” y un parámetro booleano denominado “startsWith”. El último puede ser nulo, de tal manera que se recuperarían todas las tuplas afectadas por la query.

El objeto entidad introducido como parámetro, tendrá alimentados aquellos campos por los que se desea filtrar, con los valores correspondientes, siendo estos parte de la consulta a obtener. Es decir, el filtro aplicado utilizará ‘Like’, únicamente en campos de tipo carácter, en vez de “=” que utiliza el findAll.

El parámetro “startsWith”, tendrá los valores “True” o “False” en función del filtrado que se desea realizar. Cuando el objetivo de la select sea que la cadena introducida, por ejemplo, empiece por la cadena pasada como parámetro, el valor de este será “True”. Si lo que se desea es que contenga dicha cadena, empiece o no por ella, este parámetro deberá llevar el valor “False”.

```
List<Vendor> findAllLike(Vendor vendor, Pagination pagination, Boolean startsWith);
```

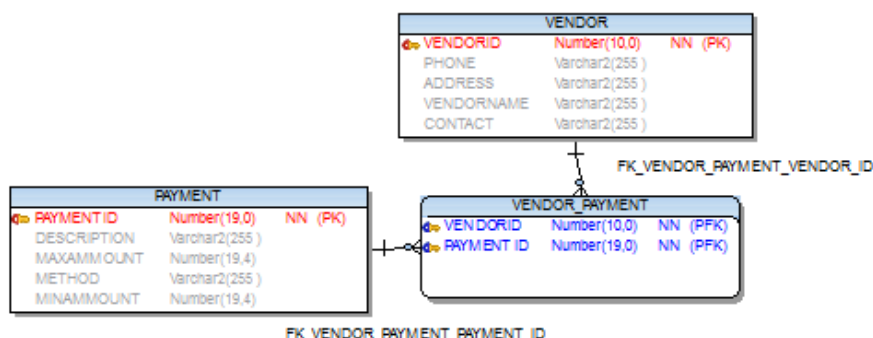
6.8.2.8 queryPagination

El método ‘queryPagination’, se crea únicamente en la implementación del DAO. Es un método auxiliar, utilizado por las funciones ‘findAll’ y ‘findAllLike’, el cual se encarga de crear la query para la consulta. En dicho método, a partir de una query ya proporcionada como parámetro, se crea otra query la cual aplica la ordenación y la paginación proporcionada al método.

El método ‘queryPagination’ recibe como parámetros un objeto ‘Pagination’ (pudiendo ser nulo), un objeto ‘CriteriaQuery’ (generado en los métodos ‘findAll’ y ‘findAllLike’), un parámetro Root<EntidadARecuperar> (generado en los métodos ‘findAll’ y ‘findAllLike’) y un objeto CriteriaBuilder (generado en los métodos ‘findAll’ y ‘findAllLike’).

Partiendo de estos parámetros, el método genera una ‘TypedQuery’ del tipo de entidad a recuperar, en la cual se aplica la paginación y la ordenación de la query proporcionada como parámetro. Los métodos ‘findAll’ y ‘findAllLike’, serán los encargados de ejecutar la sentencia que devuelve este método.

Debido a que la entidad “Vendor” se encuentra relacionada por ManyToMany con la entidad “Payment”, se generan los métodos que se explican a continuación, exclusivos para este tipo de relaciones.



6.8.2.9 *findxxxxyyy*

El método “findxxxxyyy”, recupera una colección con las entidades relacionadas y las inserta en la entidad principal. Es decir, el método devuelve un objeto entidad, el cual lleva asociado una colección de entidades relacionadas por ManyToMany.

En el caso de “Vendor”, este llevará asociada una colección de entidades de tipo “Payment”, en la cual aparecerá el detalle de la tabla “Payment” para el registro seleccionado. Este método no recupera ningún registro de la entidad actual, “Vendor” en este caso.

```
Vendor findvendorPayment(Vendor vendor, Payment payment, Pagination pagination);
```

El método recibe como parámetro un objeto entidad (“Vendor” en este caso), un objeto de la entidad relacionada por ManyToMany, además de un objeto “Pagination”. Este último puede ser nulo, de tal manera que se recuperarían todas las tuplas afectadas.

El objeto de entidad principal enviado como parámetro (“Vendor” en este caso), deberá tener alimentados obligatoriamente los campos que componen la Primary Key, con los valores correspondientes, acotando así los resultados a recuperar al mínimo. La entidad relacionado por ManyToMany (“Payment” en este caso), deberá tener alimentados aquellos parámetros por los que se desea filtrar.

6.8.2.10 *findxxxxyyyCount*

El método “findxxxxyyyCount”, recupera el número de entidades relacionadas, aplicando determinado filtro. Es decir, el método devuelve un objeto de tipo numérico, el cual indica el número de registros de la ManyToMany, aplicando el filtrado correspondiente. En el caso de “Vendor”, se contabilizará el número de registros “Payment” asociados.

```
Long findvendorPaymentCount(Vendor vendor, Payment payment);
```

El método recibe como parámetro un objeto de entidad principal (“Vendor” en este caso) y un objeto entidad relacionada (“Payment” en este caso). El objeto entidad principal enviado como parámetro, deberá llevar obligatoriamente alimentados los campos que componen a la Primary Key, acotando así los resultados a recuperar al mínimo. La entidad relacionado por ManyToMany (“Payment” en este caso), deberá tener alimentados aquellos parámetros por los que se desea filtrar.

6.8.2.11 *queryPaginationxxxxyyy*

El método ‘queryPaginationxxxxyyy’, se crea únicamente en la implementación del DAO. Es un método auxiliar, utilizado por la función ‘findxxxxyyy’, el cual se encarga de crear la query para la consulta que se desea ejecutar. En dicho método, a partir de una query ya proporcionada como parámetro, se crea otra query la cual aplica la ordenación y la paginación proporcionada al método.

El método ‘queryPaginationxxxxyyy’ recibe como parámetros un objeto ‘Pagination’ (pudiendo ser nulo), un objeto ‘CriteriaQuery’ (generado en el método ‘findAllxxxxyyy’), un parámetro Root<EntidadARecuperar> (generado en el método ‘findAllxxxxyyy’) y un objeto CriteriaBuilder (generado en el método ‘findAllxxxxyyy’).

Partiendo de estos parámetros, el método genera una ‘TypedQuery’ del tipo de entidad a recuperar, en la cual se aplica la paginación y la ordenación de la query proporcionada como parámetro. El método ‘findAllxxxxyyy’ será el encargado de ejecutar la sentencia que devuelve este método.

6.9 Orígenes de datos

La conexión a bases de datos en las aplicaciones UDA se obtendrá a través de un Pool de Conexiones, que es una colección de conexiones abiertas a una base de datos de manera que puedan ser reutilizadas bajo demanda.

Al mantenerse abierto un grupo de conexiones, éstas son atribuidas a los diferentes hilos de ejecución únicamente el tiempo de una transacción con la base de datos. Al finalizar su utilización, la conexión se pone a disposición de otro hilo de ejecución que necesite de ese recurso, en lugar de cerrarla o de asignarla permanentemente a un único hilo de ejecución.

Los servidores de aplicaciones incluyen mecanismos para crear Pools de Conexiones, utilizando Drivers transaccionales o no, y publicarlos en el árbol JNDI del propio servidor, creando así un Data Source.

De esta manera, las aplicaciones podrán acceder fácilmente a dichos Pools de Conexiones, realizando una búsqueda JNDI en el propio servidor en el que se encuentran desplegadas. Para realizar esta búsqueda, se utilizarán los tags que Spring proporciona a tal efecto, quedando así una solución declarativa y basada en estándares.

El primer paso para crear un Pool de conexiones, en el caso del servidor Oracle WebLogic 11g, consiste en acceder a la consola y dar los siguientes pasos, usando los parámetros adecuados para cada escenario.

1. Acceder al menú de Data Sources de WebLogic:

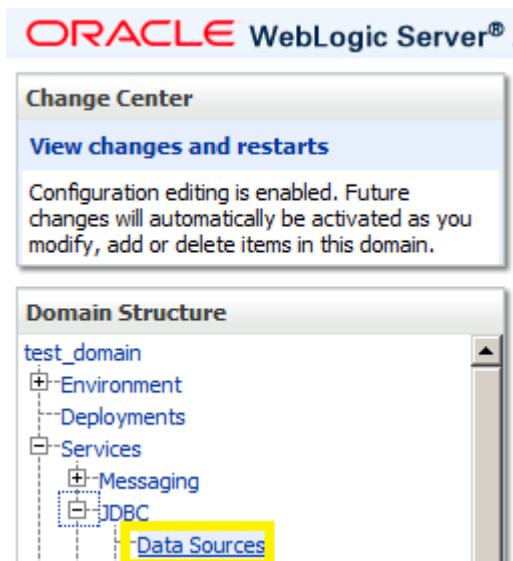


Ilustración 53. Acceder al menú de Data Sources.

2. Nuevo Data Source:

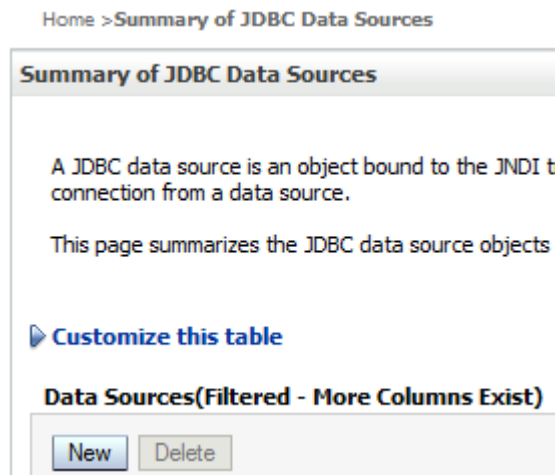


Ilustración 54. Crear nuevo Data Source.

3. Configurar el Data Source:

Create a New JDBC Data Source

Back | Next | Finish | Cancel

JDBC Data Source Properties

The following properties will be used to identify your new JDBC data source.

* Indicates required fields

What would you like to name your new JDBC data source?

*** Name:**

What JNDI name would you like to assign to your new JDBC Data Source?

JNDI Name:

What database type would you like to select?

Database Type: ▼

What database driver would you like to use to create database connections? Note: * indicates that the driver is explicitly supported by J2EE

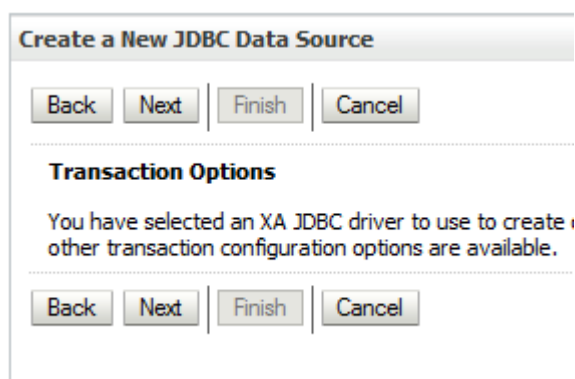
Database Driver: ▼

Back | Next | Finish | Cancel

Ilustración 55. Configuración del Data Source.

NOTA: En este ejemplo, se usa la nomenclatura de Ejje/EJGV para dar un nombre JNDI al Data Source (xxx.xxxDataSource, donde xxx representa al código de aplicación). Además, se usa un Driver XA que soporta transacciones distribuidas. En caso de no requerir transaccionalidad distribuida, se recomienda seleccionar el Driver no XA.

4. Siguiente paso:



Create a New JDBC Data Source

Back Next Finish Cancel

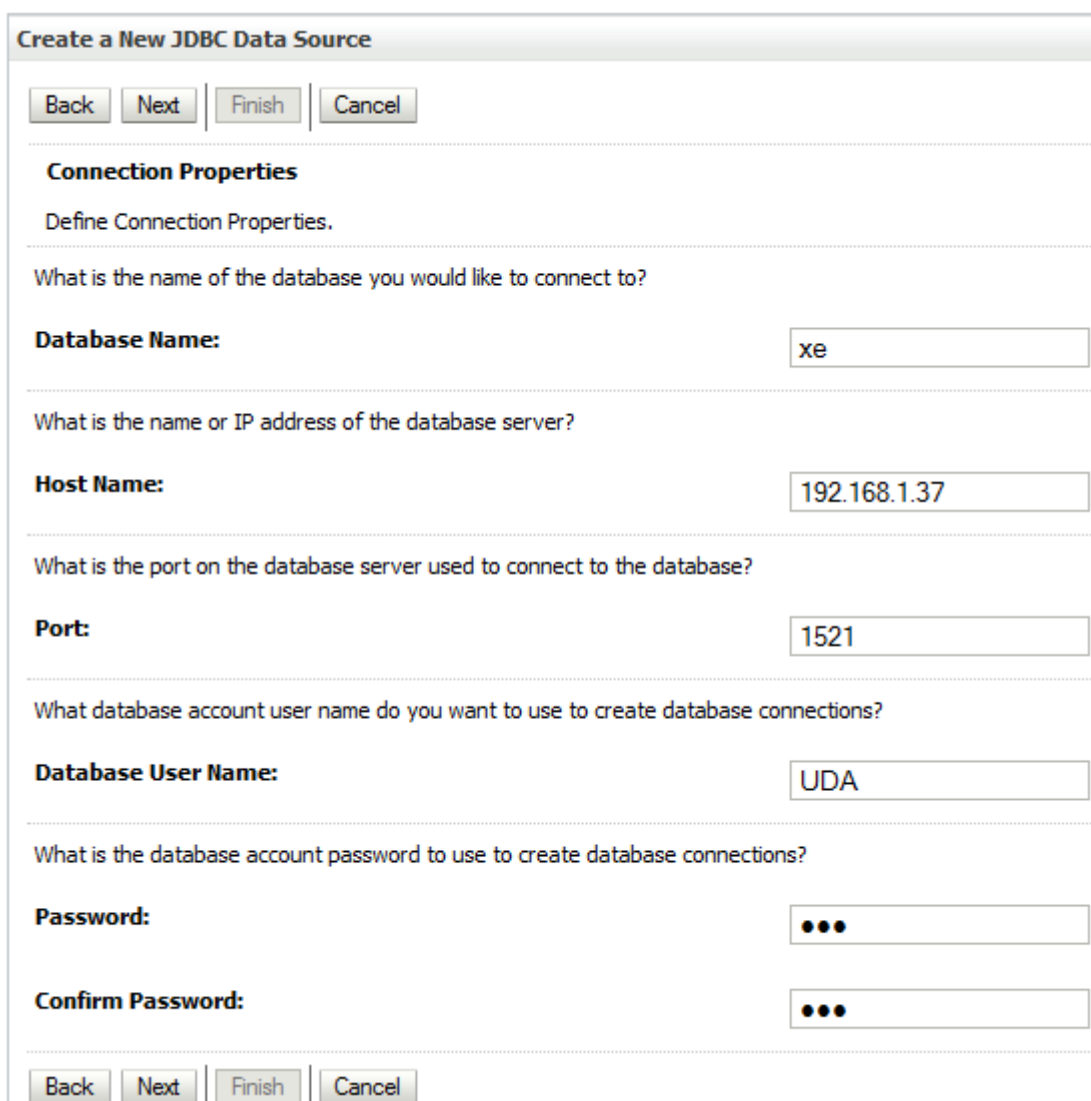
Transaction Options

You have selected an XA JDBC driver to use to create a new data source. Other transaction configuration options are available.

Back Next Finish Cancel

Ilustración 56. Siguiendo.

5. Configurar propiedades de la conexión:



Create a New JDBC Data Source

Back Next Finish Cancel

Connection Properties

Define Connection Properties.

What is the name of the database you would like to connect to?

Database Name: xe

What is the name or IP address of the database server?

Host Name: 192.168.1.37

What is the port on the database server used to connect to the database?

Port: 1521

What database account user name do you want to use to create database connections?

Database User Name: UDA

What is the database account password to use to create database connections?

Password: ...

Confirm Password: ...

Back Next Finish Cancel

Ilustración 57. Propiedades de la conexión.

6. Probar la configuración:

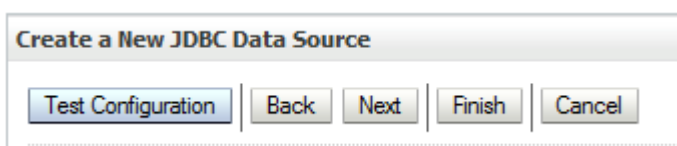


Ilustración 58. Probar configuración.

7. Confirmación de conexión correcta:

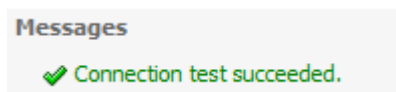


Ilustración 59. Confirmación.

8. Asignar el Data Source a un servidor y terminar:

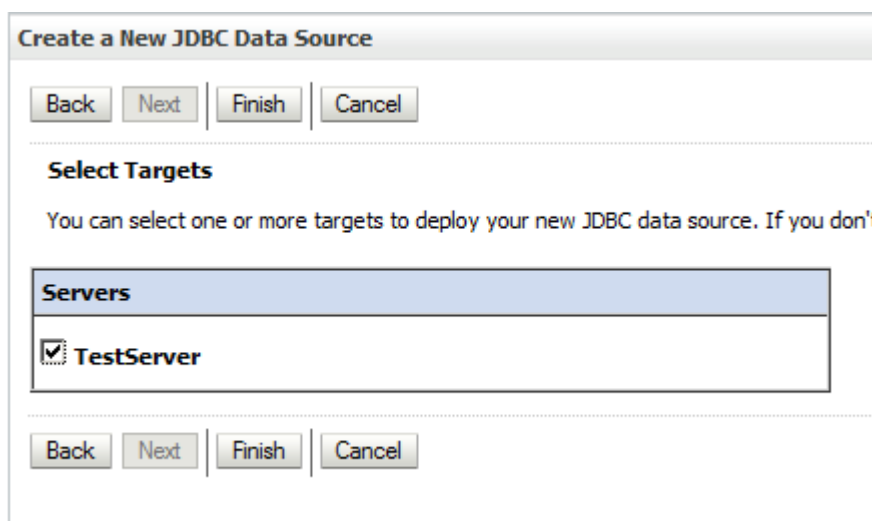


Ilustración 60. Seleccionar el Target.

9. Resumen:

Data Sources(Filtered - More Columns Exist)

New Delete

| <input type="checkbox"/> | Name ↕ | JNDI Name | Targets |
|--------------------------|----------------|---------------------|------------|
| <input type="checkbox"/> | x21aDataSource | x21a.x21aDataSource | TestServer |
| <input type="checkbox"/> | x21bDataSource | x21b.x21bDataSource | TestServer |

Ilustración 61. Resumen.

6.9.1 Uso del Data Source desde JDBC

En las aplicaciones de tipo JDBC se utiliza un tag de Spring donde se define el nombre JNDI del Data Source a utilizar y se inyecta en los componentes de la capa de acceso a datos (o DAOs) de manera declarativa.

```
<jee:jndi-lookup id="dataSource" jndi-name="x21a.x21aDataSource" resource-
ref="false" />
```

Esta etiqueta se declara en el fichero dao-config.xml de los proyectos xxxEARClasses.

Por último, el Data Source se inyecta en los componentes que harán uso del mismo. La inyección de esta dependencia se puede producir tanto por XML como por anotaciones. A continuación cada caso de inyección ilustra con su respectivo ejemplo:

- Inyección del Data Source por XML:

```
<bean id="vendorDao" class="com.ejje.x21a.dao.VendorDaoImpl">
  <property name="dataSource" ref="dataSource" />
</bean>
```

- Inyección del Data Source por anotaciones:

```
@Resource
public void setDataSource(DataSource dataSource) {
    this.jdbcTemplate = new SimpleJdbcTemplate(dataSource);
}
```

6.9.2 Uso del Data Source desde JPA 2.0

En las aplicaciones de tipo JPA 2.0 se utiliza el fichero udaPersistence.xml de los proyectos xxxEARClasses, donde se añade el tag que obtiene las conexiones partiendo del nombre JNDI del Data Source.

```
<persistence-unit name="X21A_JTA" transaction-type="JTA">
  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
  <jta-data-source>x21a.x21aDataSource</jta-data-source>
```

Además se declara el Persistence Unit que hará uso del Data Source. De esta manera, los DAOs solo tendrán que indicar al Entity Manager con que Persistence Unit ha de trabajar, ya que potencialmente existirán varios Persistence Units trabajando con diferentes Data Sources.

```
@Override
@PersistenceContext(unitName = "X21A_JTA")
public void setEntityManager(EntityManager entityManager) {
    super.setEntityManager(entityManager);
}
```

Si se quiere utilizar un Data Source no XA, habrá que declarar el Persistence Unit de esta otra forma:

```
<persistence-unit name="X21A_RL" transaction-type="RESOURCE_LOCAL">
  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
  <non-jta-data-source>x21a.x21aDataSource</non-jta-data-source>
```

Aunque la inyección del Entity Manager en los DAOs es prácticamente idéntica:

```
@Override
@PersistenceContext(unitName = "X21A_RL")
public void setEntityManager(EntityManager entityManager) {
    super.setEntityManager(entityManager);
}
```

7 Remoting

A continuación, se detallan los pasos a realizar para conseguir un escenario de comunicación remota transaccional entre dos (o más) instancias de servidores de aplicaciones.

UDA abarca dos escenarios de interacción:

1. Llamadas de una aplicación UDA a otra aplicación UDA, por ejemplo, de WebLogic 11g a WebLogic 11g. En este escenario, hace falta configurar los dominios de WebLogic 11g para posibilitar la realización de llamadas transaccionales remotas, sin tener que proporcionar las credenciales del usuario administrador del servidor remoto al servidor local.
2. Llamadas de una aplicación UDA a una aplicación Geremua 2 (el anterior entorno J2EE de Ejje/EJGV), es decir, de Weblogic 11g a WebLogic 8. En este caso, no hace falta ninguna configuración adicional de ningún entorno, ya que el cliente EJB utilizará directamente las credenciales de administración del entorno remoto.

UDA dispone de la capacidad potencial de invocar a cualquier Enterprise Java Bean (EJB en adelante) remoto desplegado en cualquier servidor de aplicaciones, aunque este documento sólo contempla la configuración necesaria para realizar llamadas remotas transaccionales entre instancias de WebLogic 11g y 8.

Además, también existe la capacidad potencial de invocar a UDA desde cualquier EJB 2.x, realizando las modificaciones pertinentes en los EJB 3.0 generados por UDA. Esta casuística está fuera del alcance de este documento, aunque a modo de ayuda, se recomienda la lectura de esta entrada de [Blog](#).

7.1 Configuración de los servidores de aplicaciones WebLogic 11g

Lo primero a tener en cuenta es que los nombres de los dominios implicados en la transacción, han de tener nombres diferentes. En el ejemplo, se emplean los dominios llamados *base_domain* y *test_domain*.

Además, las instancias de Weblogic implicadas en la comunicación, han de tener diferente nombre. En el ejemplo, la instancia vinculada al dominio *base_domain* se llama *AdminServer* y la instancia vinculada a *test_domain* se llama *TestServer*.

Adicionalmente, cada una de las instancias citadas, podría residir en una máquina diferente, por lo que cada instancia estaría vinculada a una dirección IP diferente y correría sobre una instancia de máquina virtual de Java diferente.

El protocolo a utilizar para realizar búsquedas en el árbol JNDI remoto será T3 y el protocolo para realizar la comunicación será IIOP.

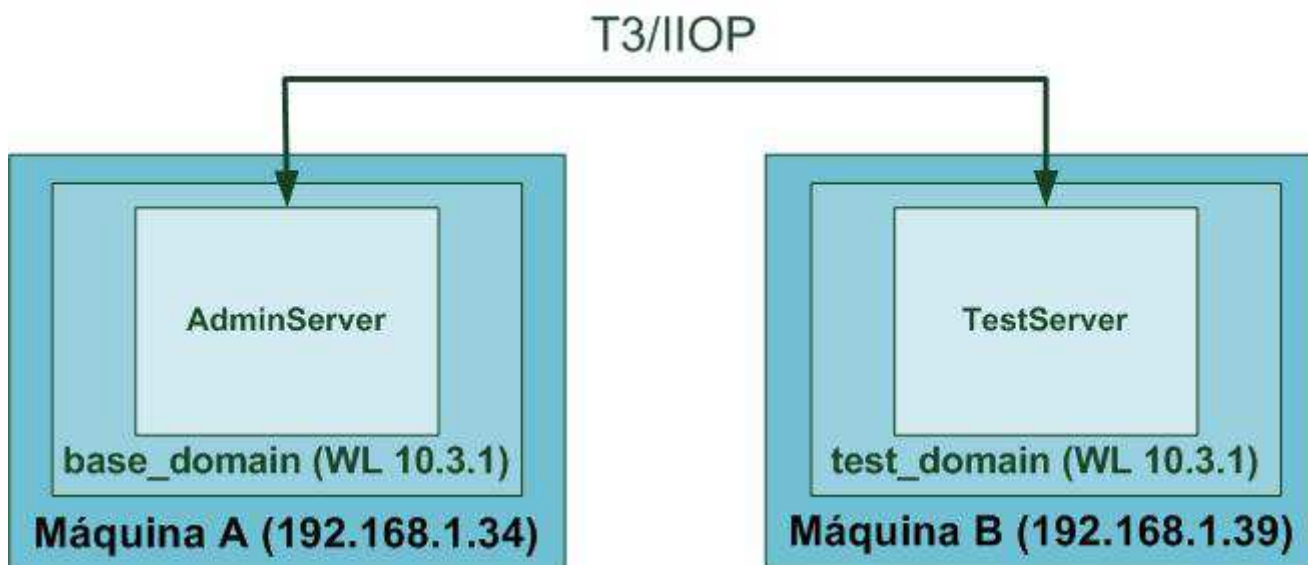


Ilustración 62. Esquema de comunicación remota.

Para que la implementación de este esquema sea posible, hay que realizar los siguientes pasos desde la consola de administración de ambas instancias de WebLogic.

- En la primera operación, se crea en ambos dominios el usuario que realizará las invocaciones remotas y se le asigna el rol CrossDomainConnectors:

1. Acceder a Security Realms

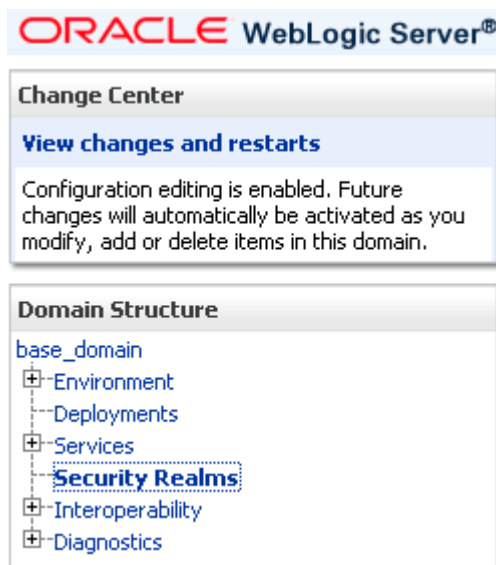


Ilustración 63. Security Realms.

2. Seleccionar myrealm

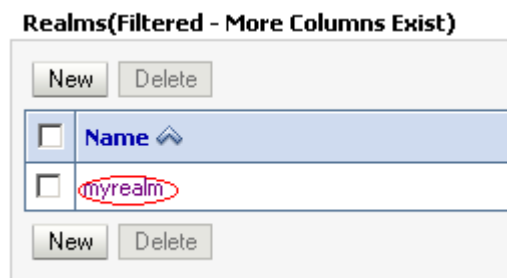


Ilustración 64. myrealm.

3. Acceder a la pestaña Users and Groups

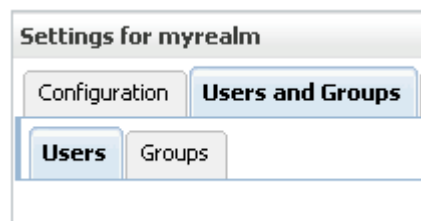


Ilustración 65. Usuarios.

4. Pulsar el botón de Nuevo usuario

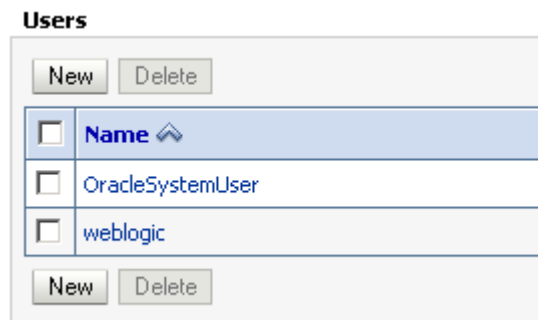
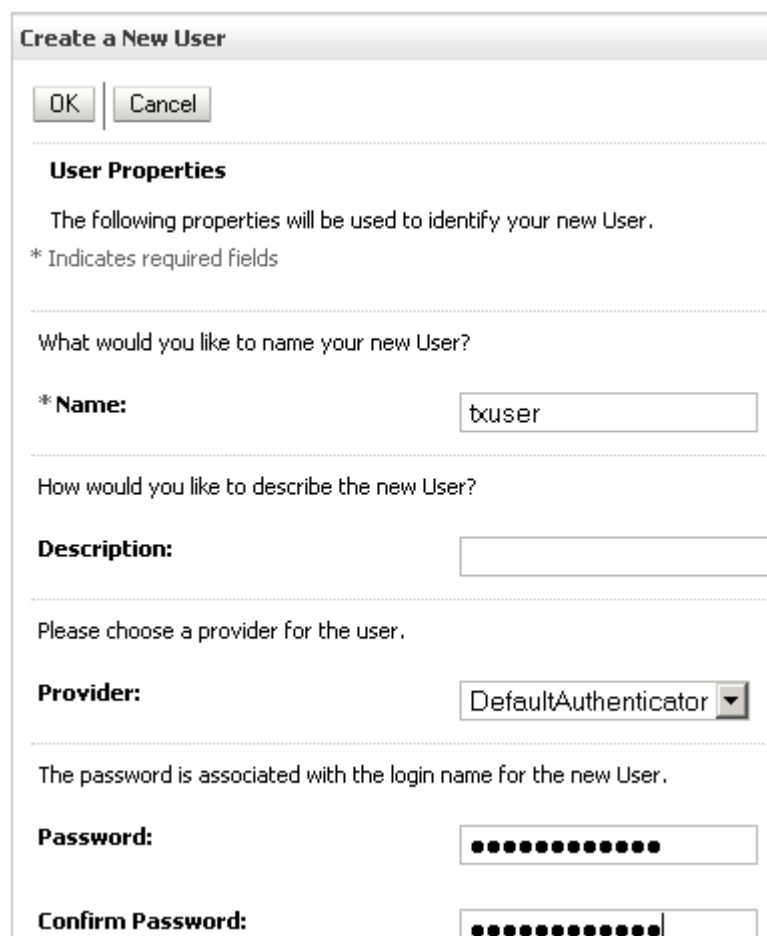


Ilustración 66. Nuevo usuario.

5. Crear usuario txuser contra el DefaultAuthenticator



Create a New User

OK Cancel

User Properties

The following properties will be used to identify your new User.

* Indicates required fields

What would you like to name your new User?

* **Name:** txuser

How would you like to describe the new User?

Description:

Please choose a provider for the user.

Provider: DefaultAuthenticator

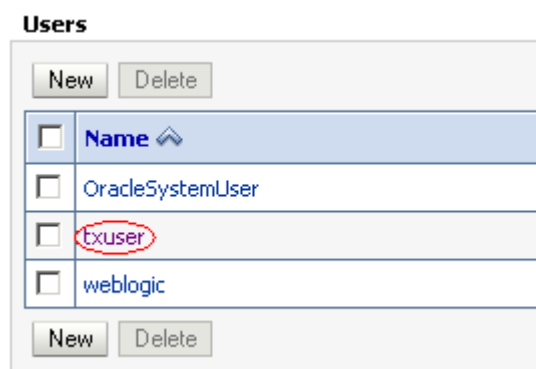
The password is associated with the login name for the new User.

Password:

Confirm Password:

Ilustración 67. Configurar nuevo usuario.

6. Usuario creado



Users

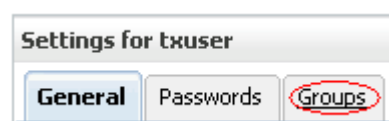
New Delete

| <input type="checkbox"/> | Name |
|--------------------------|------------------|
| <input type="checkbox"/> | OracleSystemUser |
| <input type="checkbox"/> | txuser |
| <input type="checkbox"/> | weblogic |

New Delete

Ilustración 68. Usuario creado.

7. Acceder a la pestaña Groups



Settings for txuser

General Passwords Groups

Ilustración 69. Grupos.

8. Meter al usuario txuser en el grupo CrossDomainConnectors

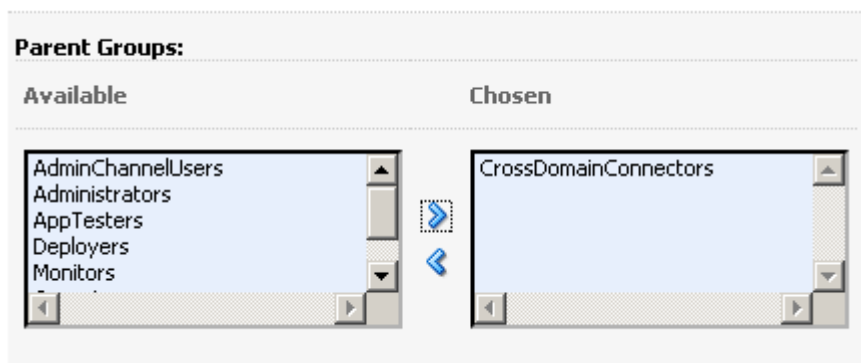


Ilustración 70. Gestionar grupos.

- En esta segunda operación, se configuran los realms de seguridad en ambos dominios para que el usuario txuser se comuniquen con dominios remotos:

1. Acceder a la consola de WebLogic
2. Entrar en Security Realms

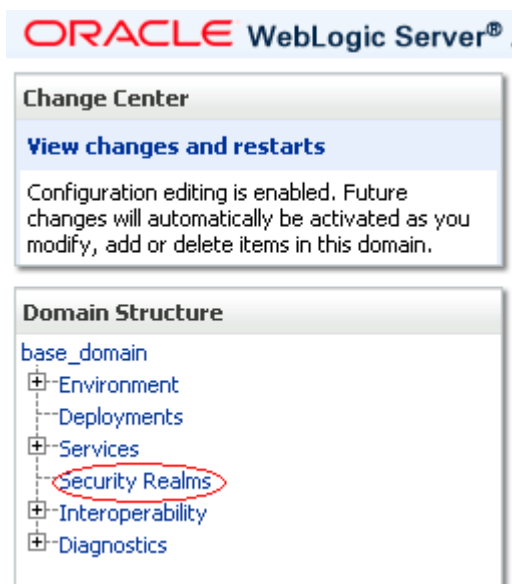


Ilustración 71. Security Realms.

3. Seleccionar myrealm

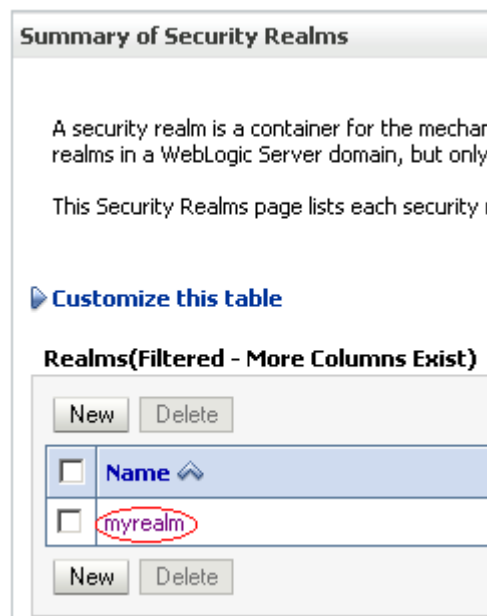


Ilustración 72. myrealm.

4. Seleccionar pestaña Credential Mappings

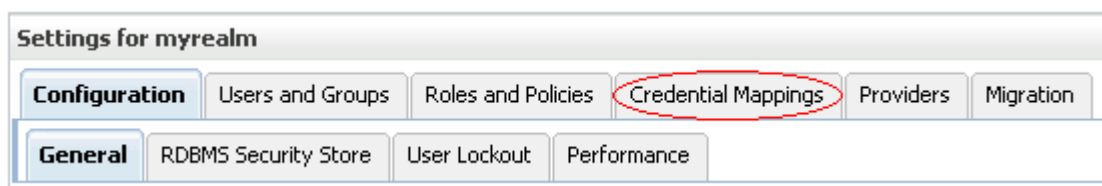


Ilustración 73. Credenciales.

5. Crear un nuevo Credential Mapping

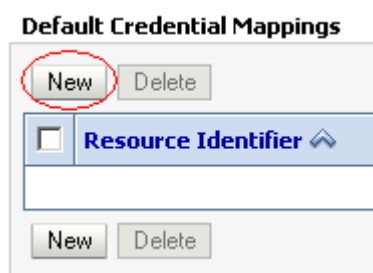


Ilustración 74. Nuevo credencial.

6. Seleccionar cross-domain protocol y añadir el nombre del dominio remoto, en este caso, test_domain

☒ **Use cross-domain protocol**

When not using the cross-domain protocol, remote resources are identified by the protocol, network address and remote resource?

Protocol:

Remote Host:

Remote Port:

Path:

Method:

When using the cross-domain protocol, remote resources are identified by the name of the remote domain.

*** Remote Domain:**

Ilustración 75. Cross domain protocol.

7. Especificar un nombre y una contraseña para el usuario que realizará la invocación remota

Create a New Security Credential Map Entry

Credential mappings let you map WebLogic Server users to remote users. Use this page to map a local user to a remote user.

* Indicates required fields

Specify a local user

*** Local User:**

Specify a remote user

*** Remote User:**

Specify a password for the remote user

*** Remote Password:**

*** Confirm Password:**

Ilustración 76. Configurar nuevos credenciales.

8. Verificar el resumen de la operación

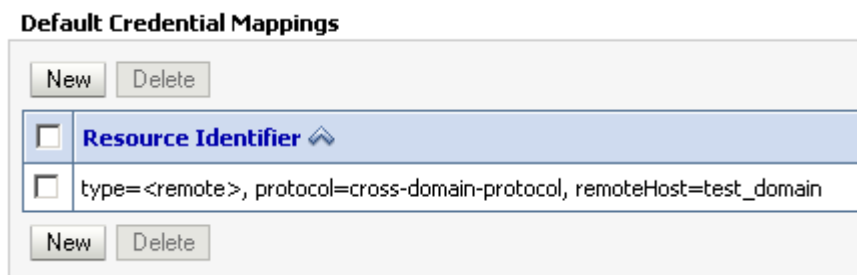


Ilustración 77. Resumen.

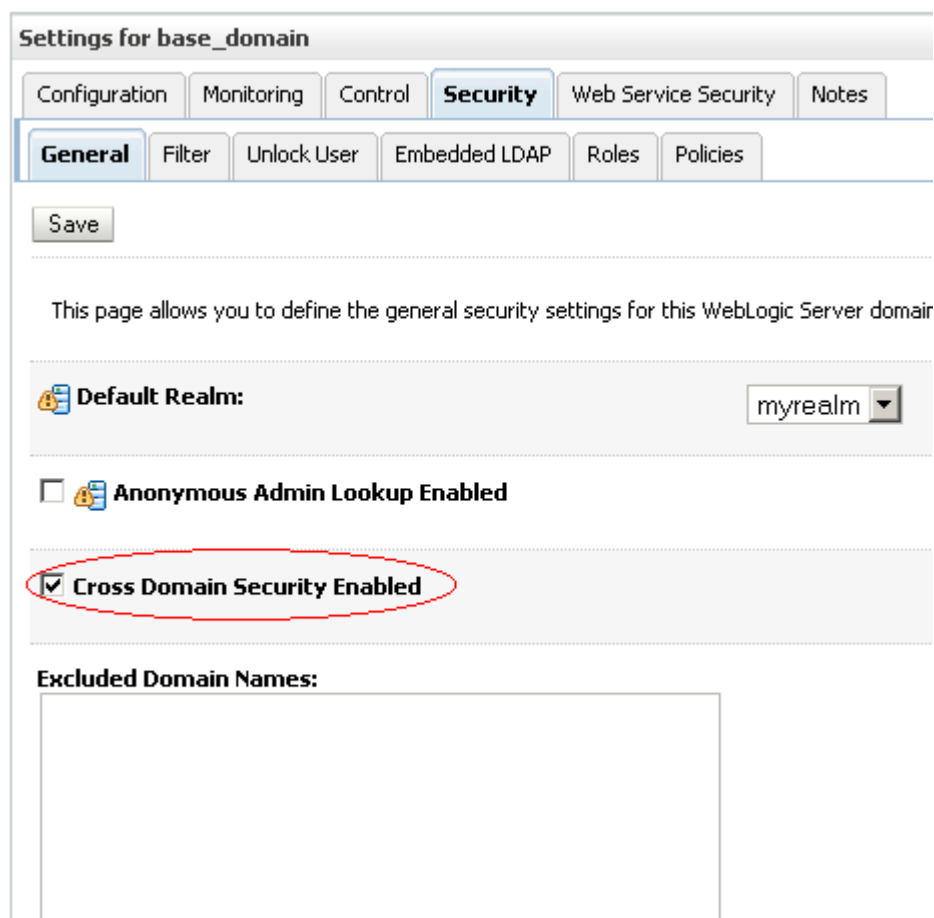
- En la tercera operación, habilita la confianza entre ambos dominios:

1. Seleccionar el dominio base_domain



Ilustración 78. Dominio.

2. Acceder a la pestaña Security y habilitar el check de Cross Domain Security. Opcionalmente, en el campo de texto llamado Excluded Domains Names se pueden enumerar los nombres de los dominios en los que no se confía




Settings for base_domain


Configuration Monitoring Control **Security** Web Service Security Notes

General Filter Unlock User Embedded LDAP Roles Policies

Save

This page allows you to define the general security settings for this WebLogic Server domain

 **Default Realm:** myrealm

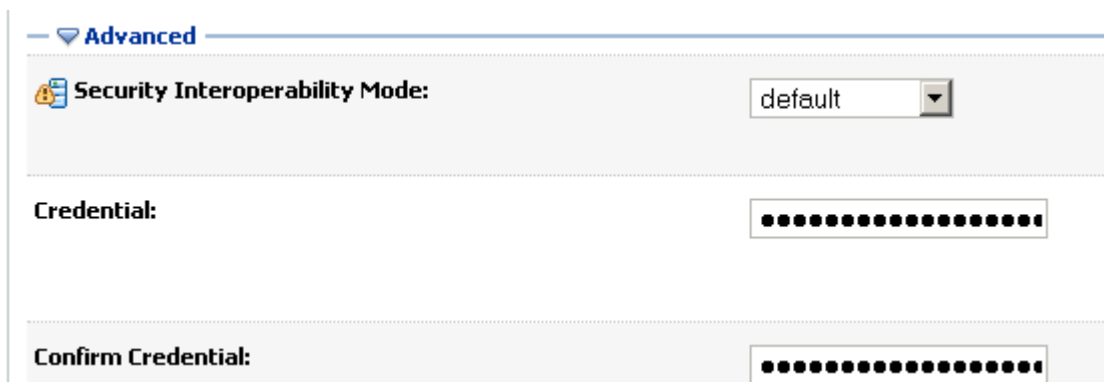
☐  **Anonymous Admin Lookup Enabled**

☒ **Cross Domain Security Enabled**


Excluded Domain Names:

Ilustración 79. Habilitar confianza.

- En opciones avanzadas, se deja el modo de interoperabilidad por defecto y se definen las credenciales del usuario que realiza las invocaciones remotas, en este caso, txuser.



Advanced

 **Security Interoperability Mode:** default

Credential:

Confirm Credential:

Ilustración 80. Credenciales por defecto.

7.2 Consumo de una aplicación UDA remota

Una vez que se disponga de una aplicación que se encuentre lista para recibir invocaciones remotas, surge la necesidad de exportar las clases necesarias para que terceras aplicaciones la puedan consumir.

Básicamente, se trata de crear un artefacto con formato JAR que contendrá las clases correspondientes a las fachadas de los servicios, el modelo y las fachadas de los EJBs.

Para generar dicho artefacto, se accede al Eclipse y se seleccionan primero los paquetes y clases que corresponden del proyecto xxxEARClasses.

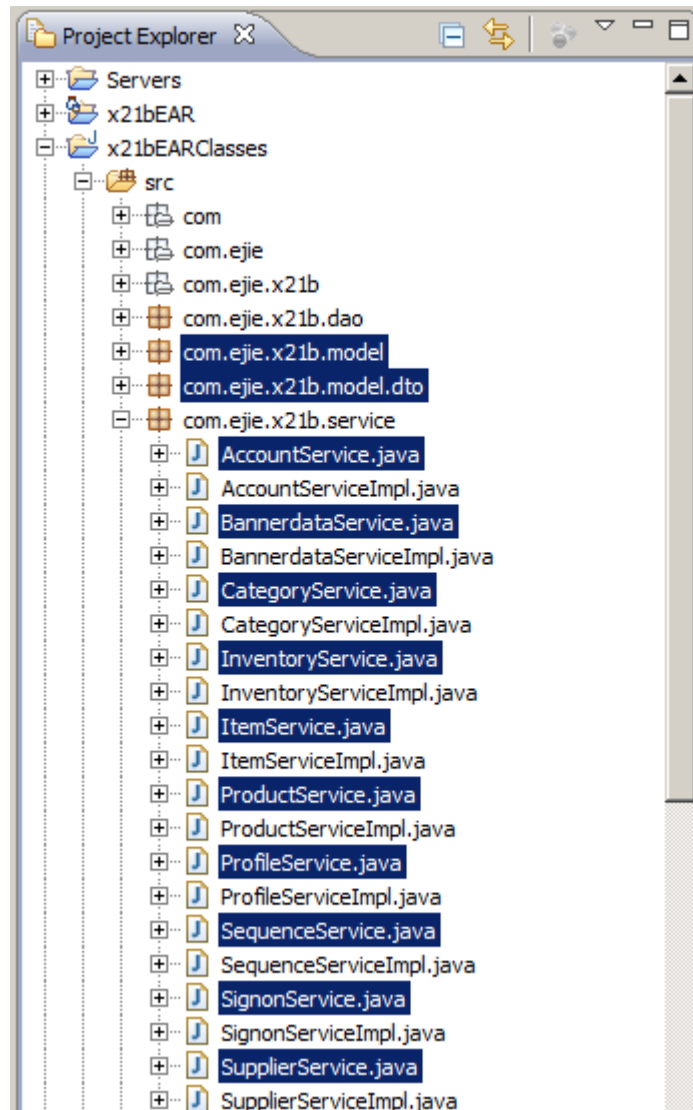


Ilustración 81. Selección del modelo a consumir.

A continuación, y sin soltar la tecla de selección “ctrl”, se seleccionarán las fachadas de los EJBs de los proyectos *EJB correspondientes.

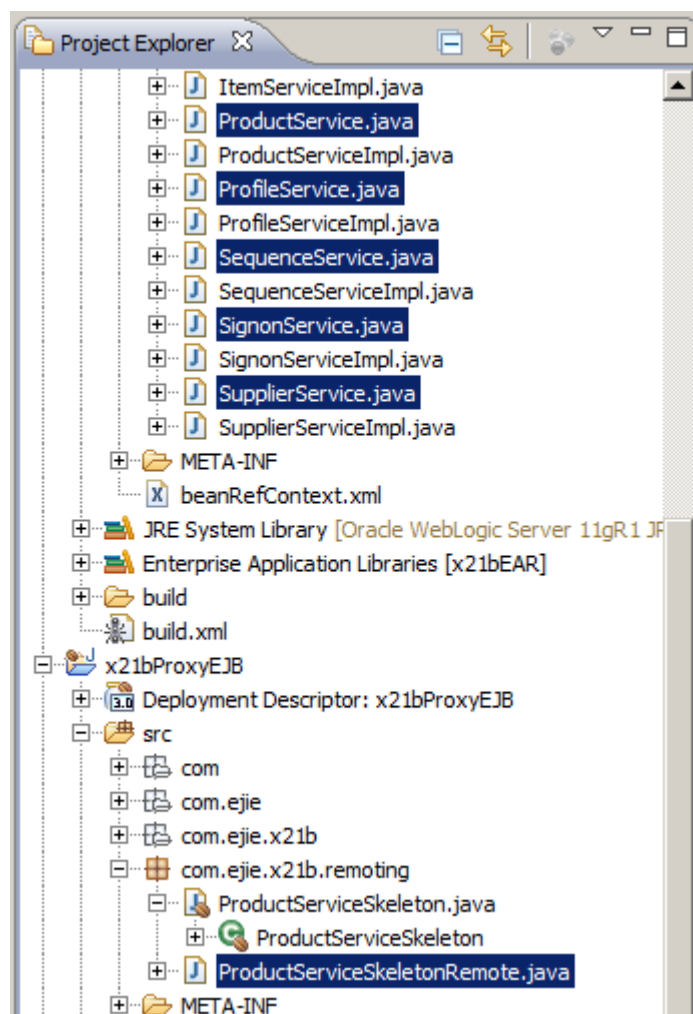


Ilustración 82. Selección de las fachadas a consumir.

Una vez realizada toda la selección de clases, se procede a generar el artefacto JAR, que se llamará xxxRemoting.jar, siendo xxx el código de la aplicación. Para ello, se realiza un click derecho sobre cualquiera de los elementos seleccionados para después seleccionar la opción “Export...”.

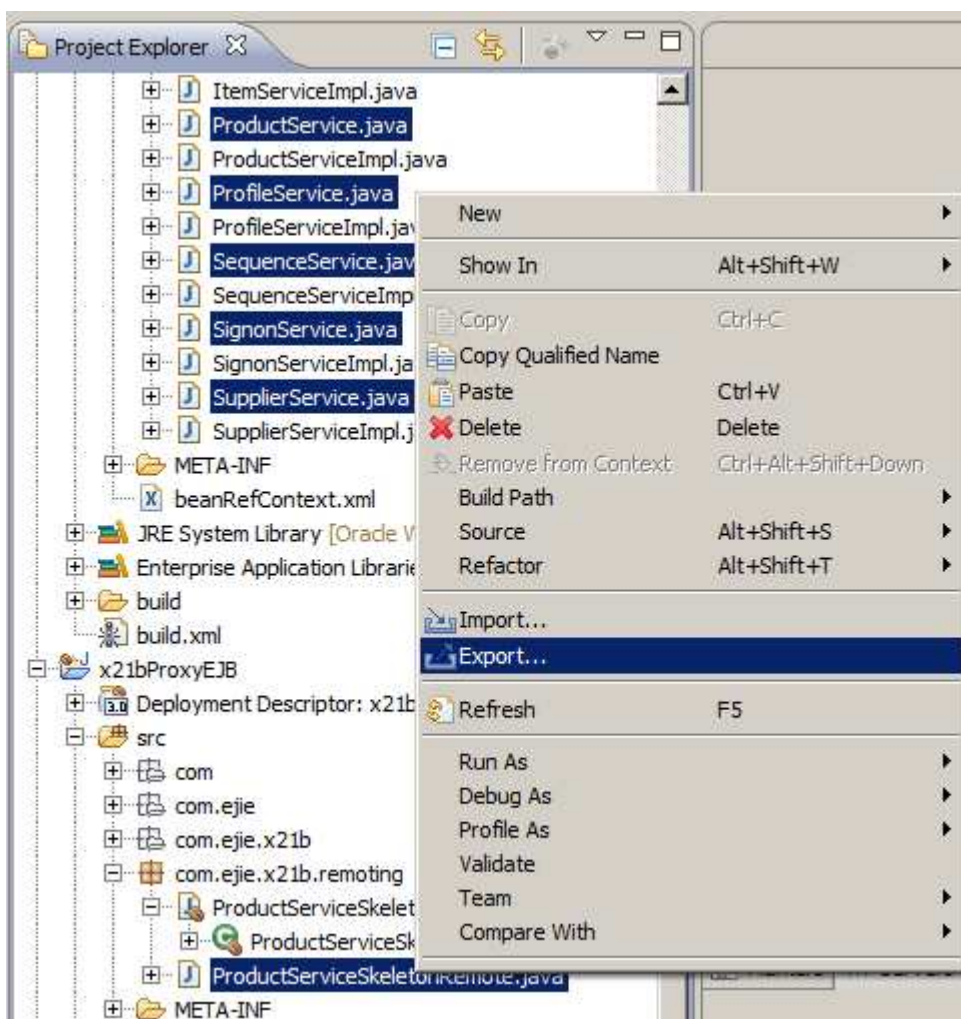


Ilustración 83. Inicio del proceso de exportación.

Después, se indica que se quiere generar un JAR.

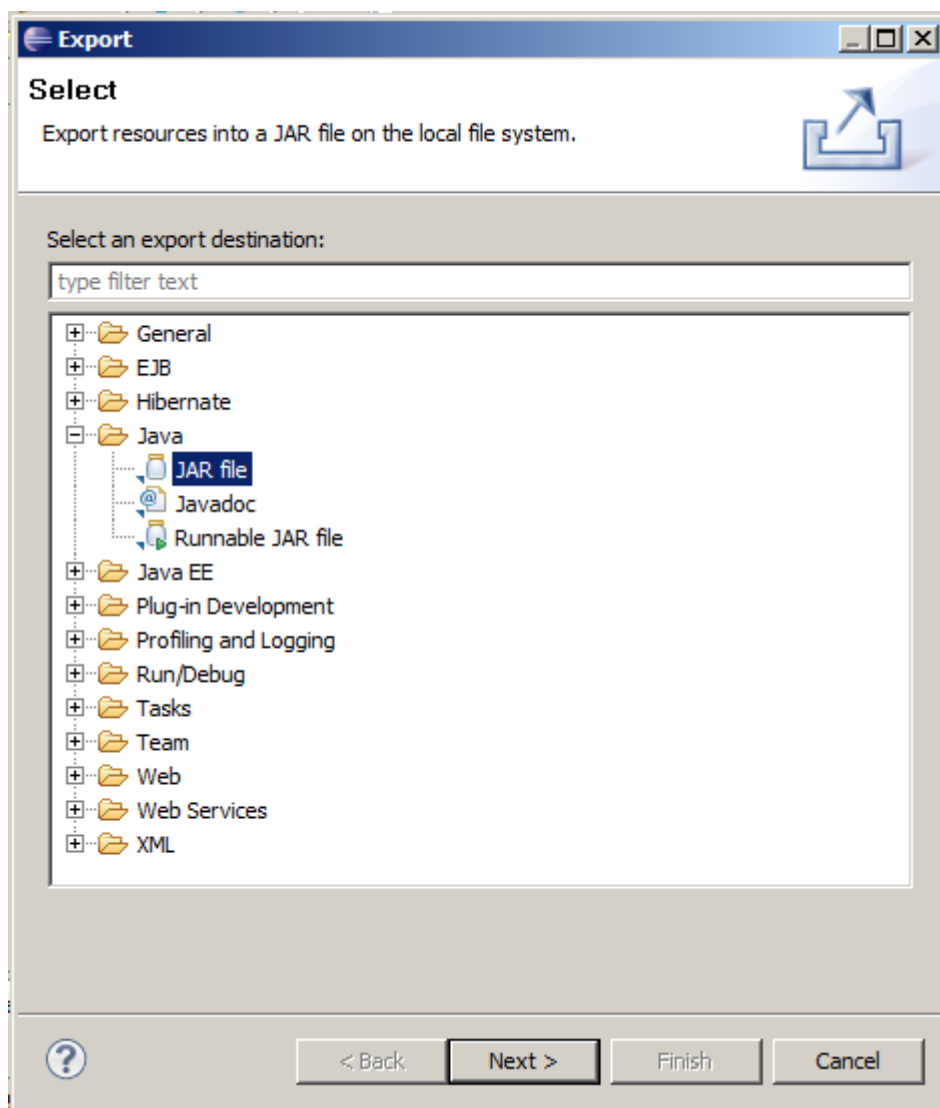


Ilustración 84. Proceso de exportación.

Por último, se establecen la ruta y el nombre del JAR resultante y se pulsa en “Finish”.

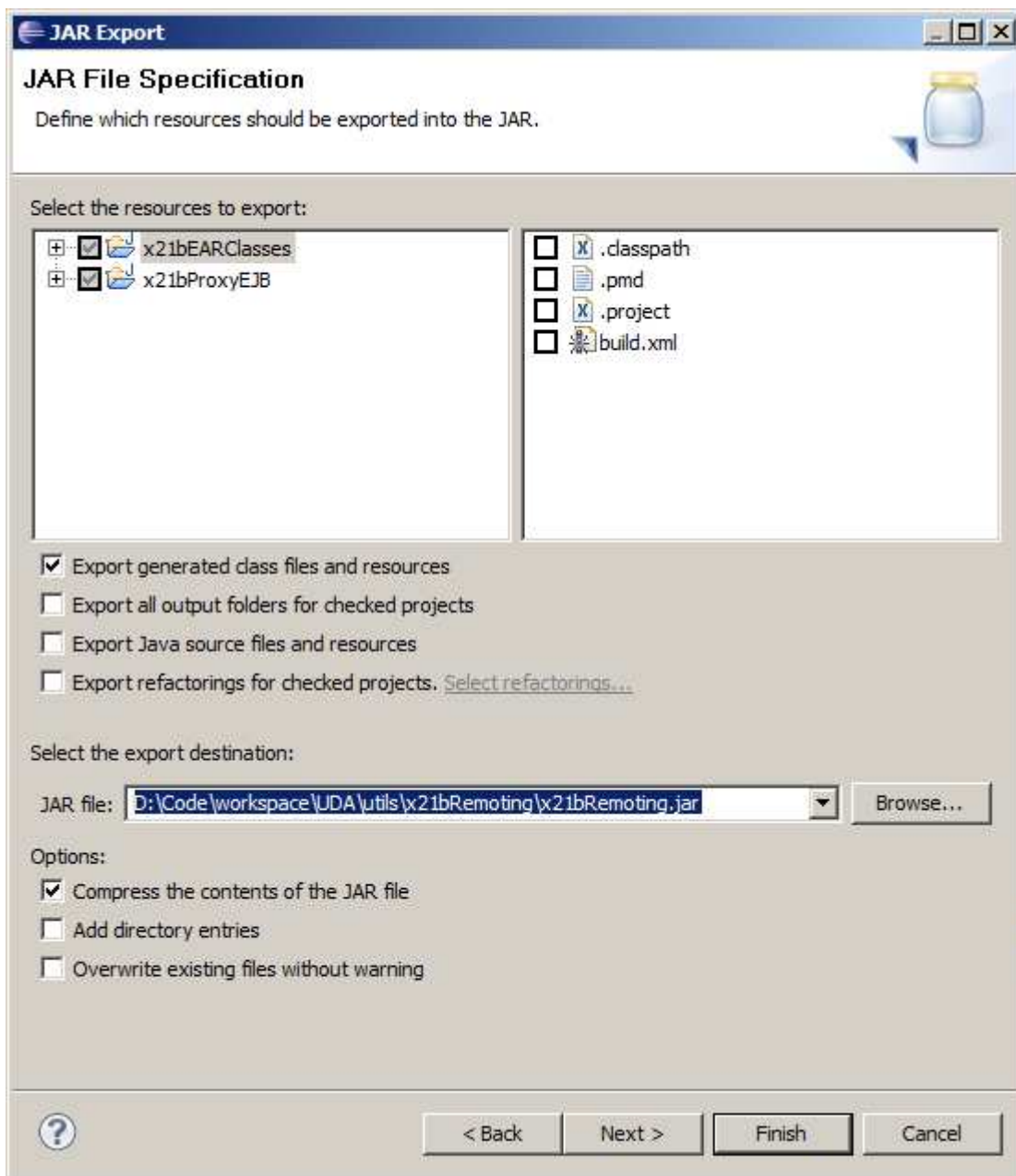


Ilustración 85. Fin del proceso de exportación.

Una vez que el artefacto xxxRemoting.jar esté disponible hay que realizar ciertos pasos para instalarlo en el entorno cliente y así poder consumirlo.

Para poder usar el artefacto xxxRemoting.jar existen dos opciones:

1. El artefacto existe en el repositorio Maven centralizado de la organización: En este caso, basta con añadir una dependencia al fichero pom.xml que se encuentra en el proyecto yyyEAR, siendo yyy el código de la aplicación que va a consumir los servicios de la aplicación xxx.

```
<!-- xxx -->
<dependency>
  <groupId>com.ejje.xxx</groupId>
  <artifactId>xxxRemoting</artifactId>
  <version>?</version>
```

</dependency>

Donde el símbolo '?' representa al número de versión del artefacto (por ejemplo, 1.0).

Por último, desde un intérprete de comandos, hay que situarse en la raíz del proyecto `yyyEAR` y ejecutar la orden **`mvn package`**. Con esto, el artefacto será automáticamente descargado e instalado.

2. El artefacto no existe en el repositorio Maven centralizado de la organización y hay que añadirlo manualmente: En este otro caso, antes de realizar los pasos indicados en el punto anterior, hay que obtener el artefacto `xxxRemoting.jar` e instalarlo manualmente en el repositorio local. Para ello, hay que situar un intérprete de comandos allí donde se encuentre el artefacto a instalar y lanzar el siguiente comando:

```
mvn install:install-file -Dfile=xxxRemoting.jar -DgroupId=com.ejie.xxx -DartifactId=xxxRemoting -Dversion=? -Dpackaging=jar
```

Donde el símbolo '?' representa al número de versión del artefacto (por ejemplo, 1.0).

A continuación, se realizan los pasos indicados en el punto anterior.

Ahora ya solo queda lanzar el asistente de creación de un nuevo cliente EJB que proporciona UDA, que generará automáticamente el Stub (o cliente EJB) con la configuración necesaria para usar dicho cliente desde la capa de servicios de negocio.

7.3 Consumo de una aplicación Geremua 2 remota

El proceso de consumir una aplicación Geremua 2 (G2 en adelante) es prácticamente idéntico al de consumir aplicaciones UDA, con la excepción de que el artefacto JAR con las clases del servidor que el cliente requiere de otro tipo de clases.

A continuación, se muestra la selección de clases necesaria para exportar en un artefacto JAR.

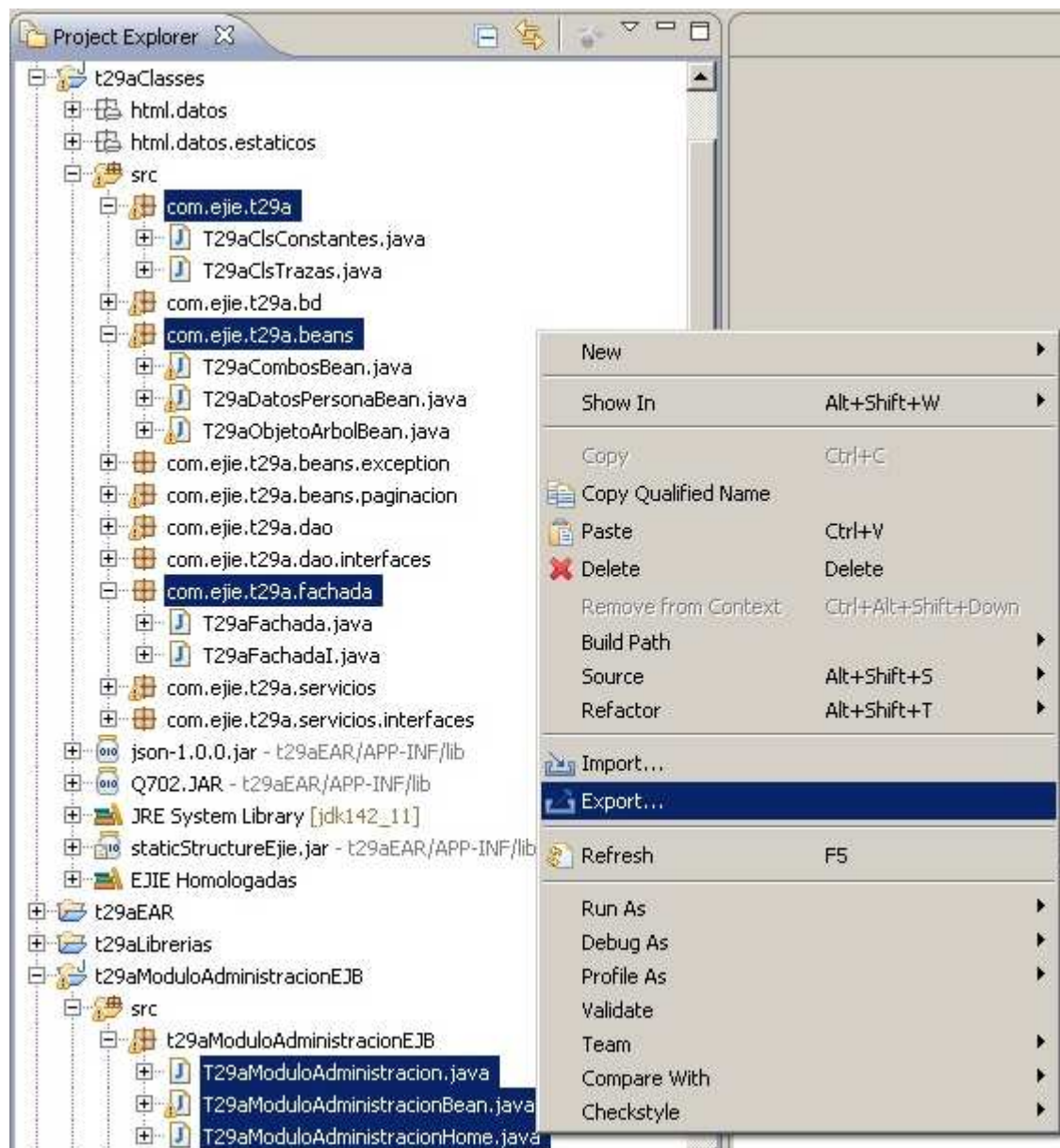
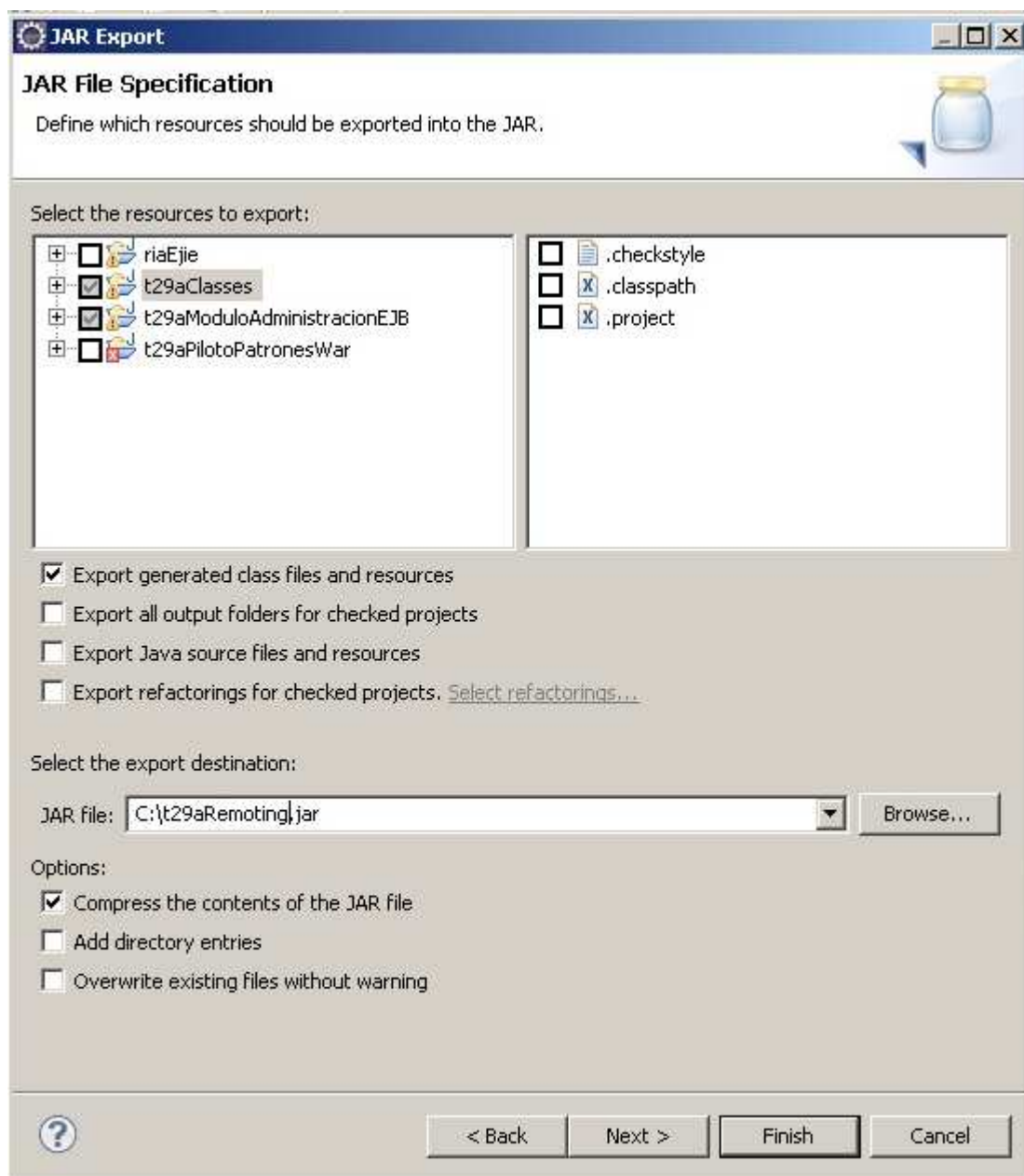


Ilustración 86. Selección de clases G2.

Después se indica que se quiere crear un JAR.



Y por último se indica donde generar el JAR con el nombre xxxRemoting.jar.



Ahora ya solo queda instalar el JAR en la aplicación cliente (como se explica en el capítulo anterior) y lanzar el asistente de creación de un nuevo cliente EJB que proporciona UDA, que generará automáticamente el Stub (o cliente EJB) con la configuración necesaria para usar dicho cliente desde la capa de servicios de negocio.

8 Transaccionalidad

UDA proporciona capacidades transaccionales a las aplicaciones de forma declarativa. Fundamentalmente se aplica a la capa de servicios aunque también se puede utilizar en la capa de acceso a datos.

Otro requisito técnico es que las transacciones son JTA aprovechando el hecho de que se dispone de un servidor de aplicaciones Java EE como WebLogic Server 11g.

8.1 Transacciones con anotaciones

La utilización de anotaciones, aunque por supuesto descentraliza la transaccionalidad, es la recomendada por Spring y la mayoría de autores. Es sin duda la más sencilla e intuitiva para el programador y además es casi idéntica a la transaccionalidad declarativa del modelo EJB 3. Con las anotaciones el método y sus atributos transaccionales residen juntos.

El ejemplo siguiente clarifica este hecho:

```
package com.ejje.app.service;
import com.ejje.app.dao.OrderingDao;
import com.ejje.x38.dto.Pagination;
import java.util.ArrayList;
import java.util.List;
import org.springframework.transaction.annotation.Transactional;

import com.ejje.app.model.Ordering;

public class OrderingServiceImpl implements OrderingService {

    private OrderingDao orderingDao;

    @Transactional(rollbackFor = Throwable.class)
    public Ordering add(Ordering ordering) {
        return this.orderingDao.add(ordering);
    }

    @Transactional(rollbackFor = Throwable.class)
    public Ordering update(Ordering ordering) {
        return this.orderingDao.update(ordering);
    }

    public Ordering find(Ordering ordering) {
        return (Ordering) this.orderingDao.find(ordering);
    }

    public List<Ordering> findAll(Ordering ordering, Pagination pagination) {
        return (List<Ordering>) this.orderingDao.findAll(ordering, pagination);
    }

    public Long findAllCount(Ordering ordering) {
        return this.orderingDao.findAllCount(ordering);
    }

    @Transactional(rollbackFor = Throwable.class)
    public void remove(Ordering ordering) {
```

```

        this.orderingDao.remove(ordering);
    }

    @Transactional(rollbackFor = Throwable.class)
    public void removeMultiple(ArrayList<Ordering> orderingList) {
        for (Ordering orderingAux:orderingList) {
            this.orderingDao.remove(orderingAux);
        }
    }

    public OrderingDao getOrderingDao() {
        return this.orderingDao;
    }

    public void setOrderingDao(OrderingDao orderingDao) {
        this.orderingDao = orderingDao;
    }
}

```

Por defecto, UDA no crea anotaciones a nivel de clase, lo que si hace es anotar los métodos de edición de datos con los valores por defecto, para que toda la operación de manipulación de datos sea transaccional. Los métodos de lectura o finders no han de ser transaccionales. En todo caso, estos últimos han de producirse dentro de transacciones en modo de solo lectura, aspecto del que se encargan los DAOs.

Es por ello que los métodos finder de los DAOs están anotados para ejecutarse dentro de transacciones de solo lectura y así optimizar el uso de la base de datos.

```

@Transactional(readOnly = true)
public List<Ordering> findAll(Ordering ordering, Pagination pagination) {}

```

Por defecto @Transactional toma los valores propagation="required" readOnly="false" isolation="DEFAULT" que son los valores asociados en la mayoría de los casos para operaciones de modificación. Para funciones tipo finders el valor readOnly="true" es el recomendado para optimizar el rendimiento.

Las opciones de propagation soportadas por Spring son equivalentes en funcionalidad y nombre a las de la versión EJB 3.0 que se utiliza en UDA.

8.2 Gestión de transacciones

Todas las aplicaciones desarrolladas con UDA que necesiten de transaccionalidad utilizan el gestor de transacciones centralizado del servidor de aplicaciones, más conocido como TransactionManager.

El uso de este componente, resulta transparente al programador, ya que las aplicaciones construidas con UDA están configuradas de tal manera que Spring trabaja automáticamente contra el TransactionManager del servidor de aplicaciones (por defecto, las aplicaciones se configuran para Oracle WebLogic 11g). Esta configuración se plasma en el fichero tx-config.xml que se incluye en todas las aplicaciones.

```

<context:load-time-weaver/>
<bean id="transactionManager"
class="org.springframework.transaction.jta.WebLogicJtaTransactionManager">
    <property name="transactionManagerName"
value="javax.transaction.TransactionManager"/>
</bean>

```

```
<tx:annotation-driven transaction-manager="transactionManager" />
<aop:aspectj-autoproxy />
```

Así se consigue que cada vez que se utilice la transaccionalidad proporcionada con Spring, mediante la anotación `@Transactional`, se estén utilizando el API del `TransactionManager` del servidor de aplicaciones de manera subyacente y transparente.

8.3 Drivers XA.

Como se comentó anteriormente las transacciones deben ser JTA. Se recomienda utilizar drivers no XA en aquellas aplicaciones (la mayoría) que no lo requieran. Es decir, si se van a utilizar en una misma transacción dos recursos diferentes se utilizarán drivers XA, en caso contrario no, ya que tendría un coste no justificable.

En el estado actual de UDA, los drivers XA solo se utilizan a nivel de drivers de base de datos. Estos drivers son proporcionados por WebLogic, ya que es en el servidor donde se configura el `DataSource` del cual se obtendrán las conexiones a base de datos.

En aplicaciones de tipo JDBC, la conexión al mencionado `DataSource` se realizará utilizando el siguiente tag dentro del fichero `dao-config.xml` y resulta transparente tanto si este `DataSource` utiliza drivers XA como si no.

```
<jee:jndi-lookup id="dataSource" jndi-name="app.appDataSource" resource-
ref="false" />
```

En aplicaciones de tipo JPA 2.0, la conexión al Data Source se hará utilizando las etiquetas estándar de JPA 2.0 en el fichero `udaPersistence.xml`. En este caso, hay que indicar si se trata de un Data Source que utiliza drivers XA o no.

```
<?xml version="1.0" encoding="utf-8"?>
<persistence version="2.0" 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_2_0.xsd">
  <persistence-unit name="X21B_JTA" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>x21a.x21aDataSource</jta-data-source>
    <non-jta-data-source>x21a.x21aDataSource</non-jta-data-source>
    <class>com.ejje.x21b.model.Ordering</class>
    <class>com.ejje.x21b.model.Part</class>
    <class>com.ejje.x21b.model.PartId</class>
    <class>com.ejje.x21b.model.Vendor</class>
    <class>com.ejje.x21b.model.Lineitem</class>
    <class>com.ejje.x21b.model.LineitemId</class>
    <class>com.ejje.x21b.model.Payment</class>
    <class>com.ejje.x21b.model.VendorPart</class>
    <class>com.ejje.x21b.model.Present</class>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
      <property name="eclipselink.target-database"
value="org.eclipse.persistence.platform.database.OraclePlatform" />
      <property name="eclipselink.target-server" value="WebLogic_10" />
      <property name="eclipselink.logging.logger"
value="com.ejje.x38.log.SLF4JLogger" />
      <property name="eclipselink.logging.level" value="FINEST" />
      <property name="eclipselink.weaving" value="false" />
    </properties>
  </persistence-unit>
</persistence>
```

```
</persistence-unit>  
</persistence>
```

8.4 Transaccionalidad distribuida en la invocación a EJB 2.x y EJB 3.0

En una operación distribuida entre dos (o más) aplicaciones UDA, conjuntamente con aplicaciones Geremua 2, la transaccionalidad distribuida está resuelta siempre y cuando todos los recursos implicados en la transacción (Data Sources, colas JMS, etcétera) utilicen drivers XA y todas las fachadas de las aplicaciones (EJB 2, EJB 3 o Spring) soporten la propagación de las transacciones (demarcación transaccional por defecto o PROPAGATE).

Para obtener la demarcación transaccional que permita la propagación de transacciones, en Spring hay que anotar los servicios simplemente con `@Transactional`.

```
@Transactional(rollbackFor = Throwable.class)  
public Vendor add(Vendor vendor) {...}
```

Si se quiere declarar explícitamente que hace falta tener una transacción abierta, hay que aplicar la siguiente anotación:

```
@Transactional(rollbackFor = Throwable.class, propagation=Propagation.REQUIRED)  
public Vendor add(Vendor vendor) {...}
```

También se puede optar por un término intermedio, indicando que si existe una transacción abierta se reutilizará, pero si la llamada no es transaccional, se producirá la invocación en modo no transaccional:

```
@Transactional(rollbackFor = Throwable.class, propagation=Propagation.SUPPORTS)  
public Vendor add(Vendor vendor) {...}
```

El equivalente a estos tres casos en EJB 3.0 sería lo siguiente, respectivamente:

```
@TransactionalAttribute  
public Product add(Product product) {...}
```

```
@TransactionalAttribute(TransactionAttributeType.REQUIRED)  
public Product add(Product product) {...}
```

```
@TransactionalAttribute(TransactionAttributeType.SUPPORTS)  
public Product add(Product product) {
```

En el capítulo de Remoting se aborda con profundidad el tema de las llamadas distribuidas, aunque en lo que atañe a la transaccionalidad distribuida, como resumen, cabe mencionar que UDA se apoya en la tecnología EJB 3.0 para realizar llamadas remotas tanto a EJB 2.x como a EJB 3.0.

Las transacciones son siempre gestionadas por el TransactionManager del servidor de aplicaciones, con lo cual, tanto los Beans de Spring como los EJBs se comunicaran bajo la misma transacción tutelada por el servidor donde todos ellos conviven.

9. Internacionalización (completado en Anexo)

Por defecto, las aplicaciones UDA están preparadas para soportar cuatro idiomas: Euskara, Castellano, Inglés y Francés.

No obstante, es posible (y muy sencillo, de hecho) añadir nuevos idiomas a medida que las aplicaciones los necesiten.

Los recursos (mensajes, etiquetas, etcétera) a internacionalizar pertenecen a dos familias: los ficheros JSON (para proyectos de contenido estático) y los ficheros Properties (para los proyectos de contenido dinámico).

En cualquier caso, el cambio de idioma se lleva a cabo lanzando una petición contra un War determinado, a lo que se le concatena el patrón “?locale=XX”. Por ejemplo:

<http://desarrollo.jakina.ejjes.net:7001/x21aDemoWar/?locale=eu>

Donde XX corresponde a la locale del lenguaje deseado.

También existe la posibilidad de cambiar de idioma utilizando el patrón cambio de idioma (RUP_language), que al fin y al cabo, lanza la URL que se acaba de describir.

9.1 Internacionalización de los proyectos estáticos

Por un lado, están los recursos que son consumidos directamente por la vista. Estos se encuentran en los proyectos de estáticos, como se muestra a continuación.

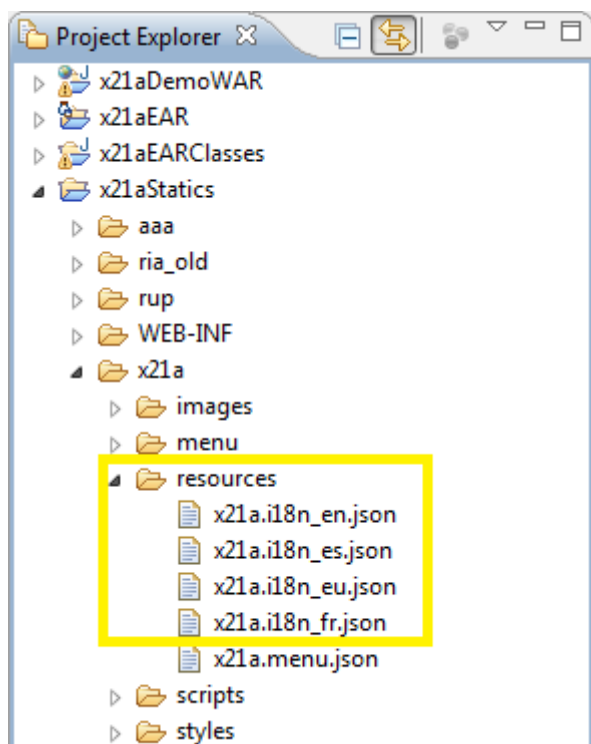


Ilustración 87. Internacionalización de la vista en aplicación x21a.

En estos cuatro ficheros se incluyen, en formato JSON, los literales que mostrará la vista al usuario final. Lógicamente, el fichero que contiene el sufijo “_es” contendrá mensajes en castellano, el que contiene “_eu”

contendrá mensajes en euskara, el que contiene “_fr” contendrá mensajes en francés y el que contiene “_en” contendrá mensajes en inglés.

Por ejemplo, este sería el contenido de los bloques de literales llamados “json_vertical” y “json_horizontal”.

```
{
  "json_vertical" : {
    "RUP" : "RUP",
    "message" : "Mensajes",
    "dialog" : "Ventanas Modales",
    "varias" : "Varias tablas",
    "otros" : "Otros",
    "combos" : "Combos",
    "combos_accesibles" : "Combos Accesibles",
    "combos_enlazados" : "Combos Enlazados",
    "submenu" : "Submenu",
    "formulario": "Formulario" ,
    "wizard" : "Wizard",
    "enlace" : "Enlace"
  },
  "json_horizontal" : {
    "RUP" : "RUP",
    "message" : "Mensajes",
    "dialog" : "Ventanas Modales",
    "feedback" : "Mensajes en línea",
    "toolbar" : "Botonera",
    "mantenimientos" : "Patrón Mantenimiento",
    "simple" : "Mantenimiento Simple",
    "multiseleccion" : "Mantenimiento Multiple",
    "ordering" : "Orden de pedidos",
    "otros" : "Otros",
    "combos" : "Combos",
    "combos_accesibles" : "Combos Accesibles",
    "combos_enlazados" : "Combos Enlazados",
    "submenu" : "Submenu",
    "formulario": "Formulario" ,
    "wizard" : "Wizard",
    "enlace" : "Enlace"
  }
}
```

También es posible internacionalizar literales que se verán reflejados en la vista, utilizando los tags que Spring proporciona a nivel de JSPs para obtener dichos literales desde los ficheros Properties de los proyectos dinámicos.

```
<spring:message code="welcome.title" />
```

9.2 Internacionalización de los proyectos dinámicos

Los mensajes de internacionalización (i18n) residen en los proyectos EARClasses y War de las aplicaciones, bajo el directorio resources.

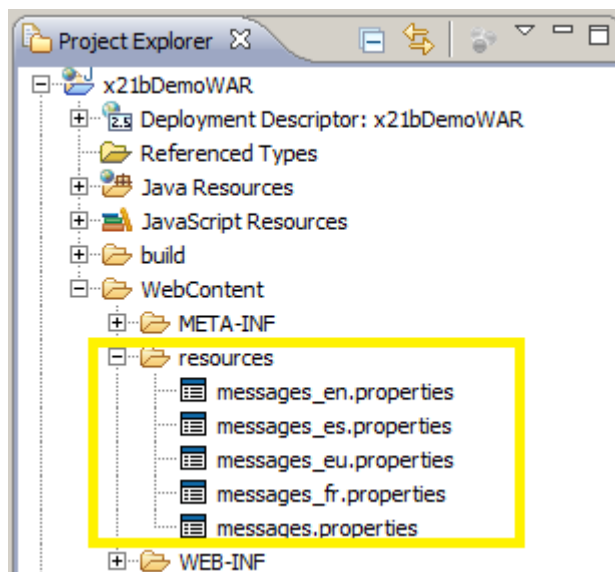


Ilustración 88. Resource Bundle de i18n de los Wars.

El grupo de mensajes que reside en los proyectos War, define los mensajes específicos para cada módulo Web.

Estos mensajes se encuentran bajo el Resource Bundle “messageSource” de cada War:

```
<!-- Application Message Bundle -->
<bean id="messageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
    <property name="parentMessageSource" ref="appMessageSource" />
    <property name="basename" value="/WEB-INF/resources/xxxYYY.i18n" />
    <property name="defaultEncoding" value="UTF-8" />
    <property name="useCodeAsDefaultMessage" value="true" />
    <property name="fallbackToSystemLocale" value="false" />
</bean>
```

Este bean, se puede inyectar en algún componente del propio War (concretamente en los Controllers), utilizando la siguiente porción de código:

```
@Resource
ReloadableResourceBundleMessageSource messageSource;
```

O por XML, de la siguiente manera:

```
<bean id="vendorController" class="com.ejje.x21a.control.VendorController">
    <property name="vendorService" ref="vendorService" />
    <property name="messageSource" ref="messageSource" />
</bean>
```

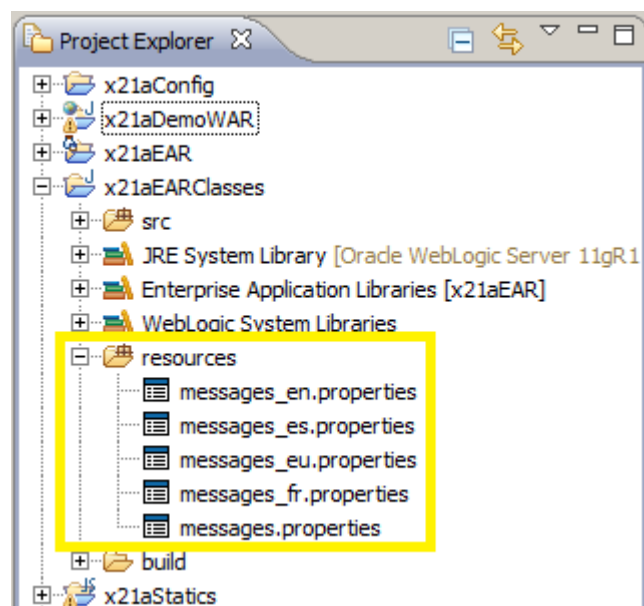


Ilustración 89. Resource Bundle de i18n de los EARClasses.

El grupo de mensajes que reside en los proyectos EARClasses es único por cada aplicación (o artefacto EAR) y define los mensajes comunes a todos los módulos de la aplicación, es decir, el propio proyecto EARClasses y los N módulos War de los que la aplicación puede disponer.

Estos mensajes se encuentran bajo el Resource Bundle “appMessageSource” de cada EARClasses:

```
<!-- Application Message Bundle -->
<bean id="appMessageSource"
class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
    <property name="basename" value="xxx.i18n" />
    <property name="defaultEncoding" value="UTF-8" />
    <property name="useCodeAsDefaultMessage" value="true" />
    <property name="fallbackToSystemLocale" value="false" />
</bean>
```

Este bean, se puede inyectar en el componente deseado de la capa de Servicios de Negocio o Acceso a Datos, utilizando la siguiente porción de código:

```
@Resource
ReloadableResourceBundleMessageSource appMessageSource;
```

O por XML:

```
<bean id="vendorService" class="com.ejje.x21a.service.VendorServiceImpl">
    <property name="vendorDao" ref="vendorDao" />
    <property name="appMessageSource" ref="appMessageSource" />
</bean>
```

Al igual que en el caso anterior, se dispone de cuatro idiomas, uno por cada fichero que contiene el sufijo “_” y adicionalmente se proporciona un fichero que no contiene sufijos para representar al idioma por defecto, que es el inglés.

Al ser ficheros de tipo properties, la información se plasma en formato “clave = valor”.

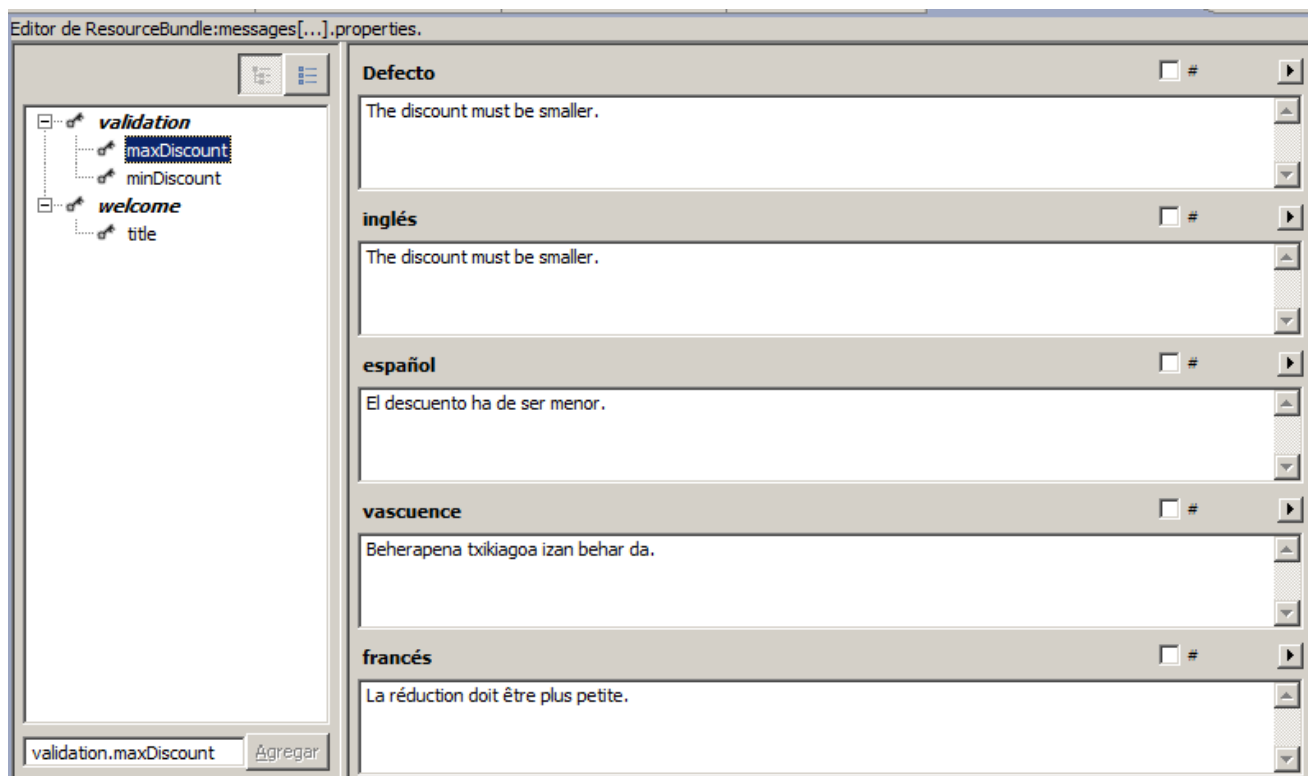


Ilustración 90. Ejemplo de mensaje de validación internacionalizado.

Estas claves se referenciarán tanto en las anotaciones de validación (esto se explica con más detenimiento en el capítulo de validación) como en las JSPs.

Como mecanismo para obtener la locale de la sesión desde cualquier componente de Spring, se propone el uso del siguiente código:

```
Locale locale = LocaleContextHolder.getLocale();
```

10. Validación (desactualizado)

Típicamente, los sistemas de validación de datos de las aplicaciones basadas en Java Enterprise Edition (JEE) suelen ser complejos, ya que se componen de reglas de validación distribuidas por las diferentes capas de las aplicaciones, y son difíciles de mantener, dado que en cada una de la capas las reglas pueden tener diferentes propósitos (formato, contenido, ...). En algunos casos, estas reglas pueden estar definidas en diferentes lenguajes de programación (JavaScript, Java, PL/SQL,...) lo cual añade aún más complejidad.

El sistema de validación de UDA se basa en el principio de centralizar las reglas validación en el modelo de datos. De esta manera, se consigue que las reglas sean consistentes y uniformes, ya que se definen una sola vez en toda la aplicación, van directamente ligadas al objetivo de la validación (los datos que contiene el modelo) y son accesibles desde cualquier capa de las aplicaciones JEE.

Este subsistema de UDA está basado en los conceptos sobre los que se erige el estándar de validaciones de Java Enterprise Edition 6 (JEE6), ya que el diseño de dicho subsistema está alineado con los fundamentos de la Java Specification Request 303 (JSR 303: Bean Validation), es decir, las validaciones se definen mediante anotaciones sobre el modelo.

```
@Min(value = 0, message = "validation.minDiscount")
@Max(value = 100, message = " validation.maxDiscount")
private Long discount;
```

En el ejemplo que acaba de mostrar, se indica que la propiedad “discount” de la entidad “Ordering” no puede encontrarse vacía, y que además, debe no puede ser mayor que cien.

Si alguna de las dos validaciones no funciona, Hibernate Validator generará un mensaje con el texto en el idioma en el que se encuentre la sesión de usuario. Si no existe tal sesión, se usará el idioma por defecto (el usado en el fichero messages.properties).

A continuación un ejemplo de los literales de validación que aplicación en este ejemplo, obtenidos del fichero messages.properties:

```
validation.minDiscount = The discount must be bigger.
validation.maxDiscount = The discount must be smaller.
```

Como se ha explicado en el capítulo anterior, existen dos grupos de mensajes (o Resource Bundles), uno a nivel de War y otro a nivel de EARClasses. Es posible que un literal determinado esté definido en ambos grupos. En ese caso, se le dará preferencia al grupo de mensajes del propio War.

En caso de no existir o no indicar mensaje alguno en ninguno de los dos grupos de mensajes que aplican a un mismo War, el sistema de validación devolverá el mensaje por defecto definido por Hibernate Validator, como en el siguiente caso:

```
@NotEmpty
private String shipmentinfo;
```

Las validaciones se pueden utilizar de desde cualquier capa de una aplicación JEE utilizando el API estándar que proporciona Hibernate Validator.

En cuanto a las anotaciones de validación que se encuentran disponibles para los desarrolladores, se recomienda estudiar directamente las que proporciona la propia herramienta en el correspondiente apartado de la [documentación oficial de Hibernate Validator](#).

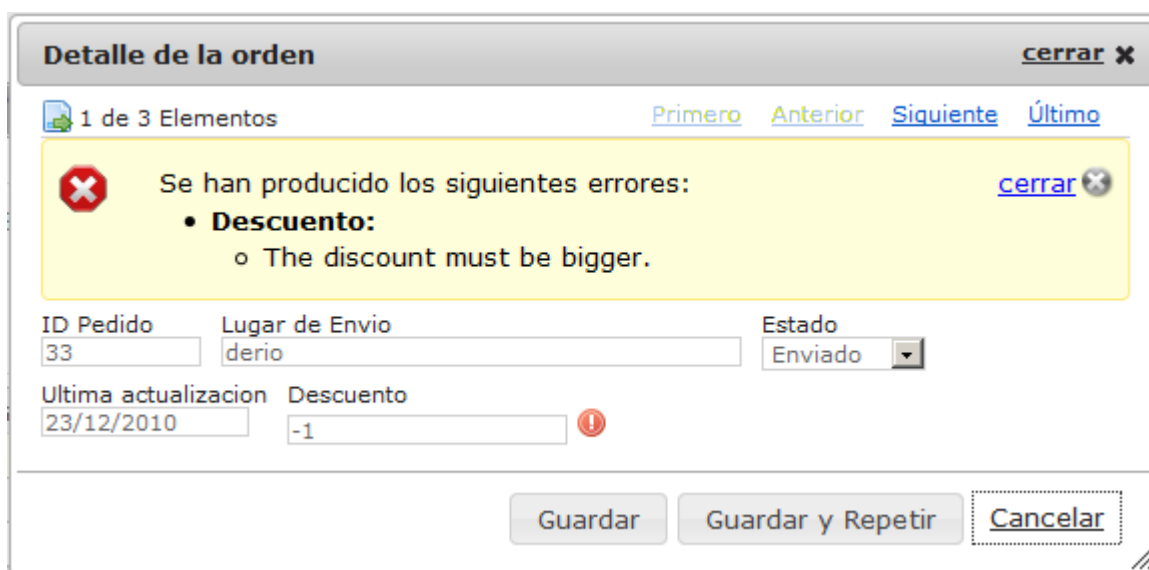
NOTA: Hay que tener cuidado a la hora de aplicar estas anotaciones al modelo, ya que muchas de ellas sólo aplican sobre un tipo de dato concreto. Si por ejemplo, una anotación que requiere String se aplica a un Long, el sistema provocará un error de tipo `javax.validation.UnexpectedTypeException`.

Para poder utilizar las validaciones desde la capa de presentación, más concretamente, desde la vista, UDA proporciona una infraestructura que se encarga de recibir las peticiones Ajax de validación enviadas por la vista y devolver los pertinentes mensajes de confirmación o denegación resultante de la validación de datos. En el caso de denegar la validación, UDA responderá las peticiones Ajax con objetos JSON que contienen los literales de validación internacionalizados que se han definido en el grupo de ficheros `messages.properties`.

10.1 ~~Validación unitaria desde la vista (desactualizado)~~

En ciertos casos, resulta apropiado validar ciertos campos de entrada de datos de usuario antes de realizar una acción de modificación de datos contra el servidor.

Así, el usuario podrá comprobar en el acto si el dato introducido es válido, o no, a través del área de feedback.



Detalle de la orden [cerrar](#) ✕

1 de 3 Elementos [Primero](#) [Anterior](#) [Siguiente](#) [Último](#)

Se han producido los siguientes errores: [cerrar](#) ✕

- **Descuento:**
 - The discount must be bigger.

ID Pedido: 33 Lugar de Envio: derio Estado: Enviado

Ultima actualizacion: 23/12/2010 Descuento: -1

[Guardar](#) [Guardar y Repetir](#) [Cancelar](#)

Ilustración 91. Fallo de validación.

Detalle de la orden

cerrar x

1 de 3 Elementos

[Primero](#)
[Anterior](#)
[Siguiente](#)
[Último](#)

Se han producido los siguientes errores:

• Lugar de Envio:

o may not be empty

cerrar x

ID Pedido

33

Lugar de Envio

Estado

Enviado

Ultima actualizacion

23/12/2010

Descuento

12

Guardar

Guardar y Repetir

Cancelar

Ilustración 92. Fallo de validación con mensaje por defecto de Hibernate Validator.


10.2—Validación masiva desde la vista (desactualizado)

En ocasiones, resulta más práctico validar todos los datos de un formulario, una vez que se ha ordenado la acción de modificación de datos contra el servidor.

En este caso, se validarán todos los datos de forma colectiva y en caso de no pasar alguna regla de validación, toda la acción se rechazará y se devolverá un mensaje de feedback donde se le indicará al usuario que dato no es válido y porque.

Detalle de la orden
cerrar x


1 de 3 Elementos
[Primero](#)
[Anterior](#)
[Siguiente](#)
[Último](#)


Se han producido los siguientes errores:
cerrar x

- Lugar de Envio:**
 - may not be empty
- Descuento:**
 - The discount must be bigger.

ID Pedido
33


Lugar de Envio



Estado
Enviado

Ultima actualizacion
23/12/2010

Descuento
-1



Guardar
Guardar y Repetir
Cancelar

Ilustración 93. Validación masiva.

11 Seguridad (desactualizado)

Como se ha comentado anteriormente en este documento, la línea estratégica en el ámbito de seguridad que se plantea para el desarrollo de aplicaciones basadas en UDA, es la que se apoya en la utilización de Spring Security para la gestión de la autenticación y la autorización. De esta forma, se consigue que los desarrolladores únicamente deban conocer la securización de URLs y métodos de la capa de servicios de negocio. Todo ello sucede bajo el paraguas de Spring Security, de forma que el uso de XLNetS (en caso de tratarse de entornos Ejje/EJGV) es totalmente transparente al programador.

11.1 Fundamentos tecnológicos (desactualizado)

UDA dispone de sólidos pilares en el ámbito de la seguridad, que son:

- Los servicios ya existentes en EJIE y Gobierno Vasco a través de la plataforma XLNetS (aplicación n38). El uso de dichos servicios solo es requerido en entornos Ejje/EJGV.
- Spring Security, gracias a la implementación a medida utilizada en UDA.

De esta forma, la seguridad de las aplicaciones construidas sobre UDA dispondrá de las siguientes funcionalidades:

- El sistema de seguridad central de todas las aplicaciones Java EE continuará siendo XLNetS.
- Las aplicaciones delegarán la seguridad, autenticación y autorización en Spring Security, con todos los beneficios que esto aporta.
- Será Spring Security que a través de los Wrappers de XLNetS que proporciona UDA (en la librería x38ShLibClasses.jar concretamente), realizará las llamadas pertinentes a XLNetS.
- Todo lo referente a XLNetS será transparente para los desarrolladores que únicamente necesitarán conocimientos generales de Spring Security.

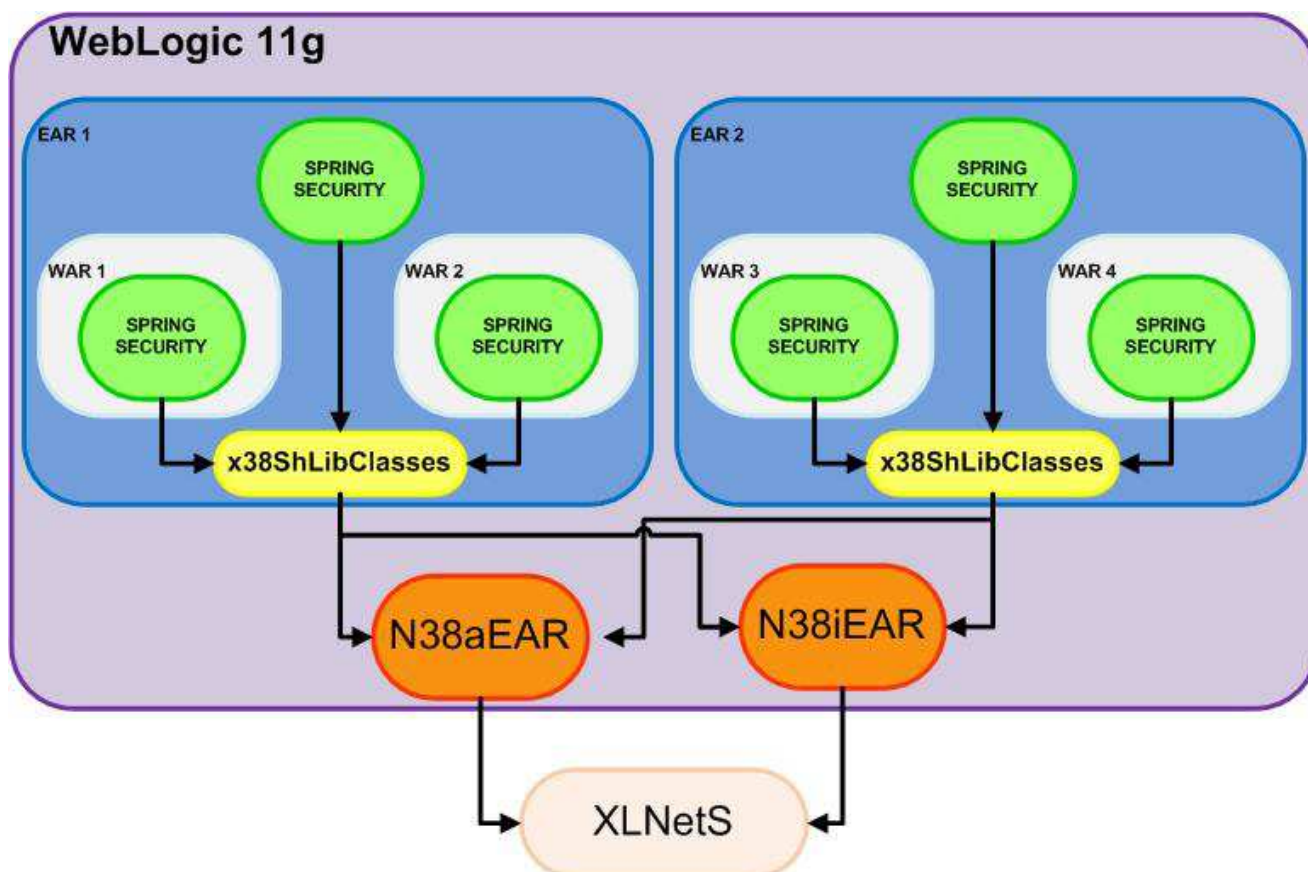


Ilustración 94. Arquitectura del sistema de seguridad.

Aunque no es el objetivo de este documento el detallar el funcionamiento de cada uno de estos elementos, con el objetivo de orientar al lector en las funcionalidades y conceptos de cada uno de estos pilares, a continuación se introducen brevemente cada uno de ellos.

11.1.1 XLNetS (desactualizado)

El sistema XLNetS es un proyecto desarrollado para poder integrar la seguridad de todos los entornos que ahora mismo existen en el gobierno (Unix, NT, C/S...), a través de un único punto de acceso web y cuyo backend está basado en un directorio LDAP.

La arquitectura de XLNetS está basada en un API para el Login, otro para el Toolkit de desarrollo, un servidor de seguridad y de acceso a LDAP y el directorio LDAP. Todas estas capas están desarrolladas con Java y se comunican con XML y un protocolo (pn38) propio del sistema.

Una de las principales funcionalidades, que nos ofrece XLNetS, es la posibilidad de realizar controles de acceso a nuestras aplicaciones y especificar la seguridad de las mismas. Para ello es necesario realizar un análisis previo de las necesidades a nivel de seguridad de nuestra aplicación y, en función de ellas, definir la estructura del árbol de seguridad de nuestra aplicación.

Si se desea profundizar en este ámbito, se pueden consultar la documentación propia de XLNetS y más concretamente, el manual *"Definición y estudio de la seguridad de una aplicación basada en XL-Nets"* y *"XLNetS – Manual de desarrollo.doc"*

11.1.2 Spring Security, Authentication and Authorization Service (desactualizado)

Los servicios de autenticación y autorización en UDA son manejados por Spring Security. Estos servicios facilitados en Spring Security se utilizan con dos propósitos:

- Conocer la identidad del usuario que pretende acceder a los recursos de la aplicación y verificar la autenticidad de dicha identidad.
- La autorización de las acciones que desea realizar el usuario autenticado comprobando los permisos y derechos requeridos para ejecutar dichas acciones.

Para conseguir llevar a cabo estos propósitos, Spring Security implementa la versión Java del framework estándar PAM ([PAM, Pluggable Authentication Modules](#)). Esta implementación introduce dos conceptos al ámbito de seguridad: la **autenticación** y la **autorización**.

La **autenticación** es la encargada de comprobar la veracidad de la identidad del usuario. Aparte, la gran ventaja de la autenticación es permitir que las aplicaciones permanezcan independientes de las tecnologías de autenticación. De esta forma, se pueden acoplar sistemas de autenticación nuevos o actualizados sin que se requieran cambios a la aplicación en sí misma. En el caso de UDA, se oculta el funcionamiento de XLNetS como elemento final para el control de la seguridad.

La **autorización** que permite proteger el acceso a los recursos de la aplicación, para ello aporta dos posibles variantes a utilizar por el desarrollador:

- Protección a nivel de URL: Permite decidir que permisos debe tener el usuario para acceder a un recurso Web. La configuración de esta seguridad se realiza mediante un fichero security-config.xml de los Wars → Más adelante, se profundiza en cómo hacerlo.
- Protección a nivel de servicios de negocio: Permite acceder o no a servicios de negocio dentro de las librerías *appEARClasses.jar* (donde *app* es el nombre de la aplicación) de las aplicaciones. Esto se configura en el fichero security-config.xml de los *appEARClasses* → Más adelante, se profundiza en cómo hacerlo.

Estas dos operaciones se comentarán en profundidad y con ejemplos más adelante.

Antes de entrar en la parte práctica del capítulo, conviene entender algunos conceptos que rodean a Spring Security y hacen comprensible su funcionamiento:

- Authentication. Es la clase principal de este sistema de seguridad. El objeto Authentication representa la agrupación de cierta información relacionada con una única entidad, por lo general una persona (el usuario que se conecta a la aplicación) y está compuesto por su Principal, las Credentials de ese Principal y sus Authorities.
 - Principal. Un Principal representa una identidad única en el sistema, en el caso de UDA es una persona identificada por su nombre de usuario y su UID-Session (identificador único por cada conexión del usuario).
 - Credentials. Las credenciales están relacionadas con el principal y son aquellas pruebas que identifican al principal de forma inequívoca. Por ejemplo, passwords, certificados digitales, identificadores de sesión... etcétera.
 - Authorities. Se trata de aquellos permisos que tiene el principal y que le permiten acceder a recursos protegidos de la aplicación. **Estos permisos corresponden en UDA con las instancias que un usuario tiene asignados en XLNetS.**

Se puede consultar información complementaria sobre Spring Security y otras secciones de la arquitectura de seguridad de la plataforma Java SE y Java EE, en las siguientes referencias:

- <http://java.sun.com/javase/technologies/security/>
- <http://static.springframework.org/spring-security/site/index.html>

11.1.3 Wrapper del Sistema Perimetral para Spring Security (desactualizado)

Un sistema de seguridad perimetral es un sistema externo en el que otro sistema (en este caso Spring Security en UDA) delega para verificar la autenticidad del usuario y obtener sus autorizaciones. El sistema perimetral elegido por Gobierno Vasco es XLNetS.

UDA proporciona un wrapper (envoltorio o conector) que permite el uso de éste o cualquier otro sistema de seguridad perimetral en conjunción con Spring Security. Para ello proporciona una fachada (un interface) que una vez implementado permite conectar Spring Security con el sistema de seguridad perimetral elegido.

En el siguiente código JAVA se pueden ver los métodos de la fachada que se implementa.

```
public interface PerimetralSecurityWrapper {

    public abstract String getUserConnectedUserName(HttpServletRequest
    httpRequest);

    public abstract String getUserConnectedUidSession(HttpServletRequest
    httpRequest);

    public String getUserPosition(HttpServletRequest httpRequest);

    public String getURLLogin(String originalURL);

    public Vector<String> getUserInstances(HttpServletRequest httpRequest);

    public void logout(HttpServletRequest httpRequest);

}
```

UDA proporciona la implementación de la fachada para el sistema XLNetS de Gobierno Vasco. Otros sistemas de seguridad requerirían su propia implementación.

11.2 — Prerrequisitos (desactualizado)

Para poder incorporar seguridad en las aplicaciones Java EE se han de cumplir los siguientes requisitos:

1. Si se desea delegar en XLNetS, el contenedor debe gestionar la autenticación utilizando la infraestructura ya existente de XLNetS, por lo que es necesario disponer de una instalación de este sistema en el entorno local. Este punto queda fuera del alcance de este documento, sin embargo es posible consultar los pasos necesarios en el documento "*Plataforma de desarrollo en Local*".
2. Para utilizar el sistema de seguridad, se ha de disponer, como mínimo, de la instalación de un contenedor de Servlets (por ejemplo Apache Tomcat) o de un servidor de aplicaciones (por ejemplo JBoss).
Sin embargo, UDA en general está especialmente diseñado para funcionar sobre la plataforma WebLogic Server 10.3.1 (Weblogic 11g), por lo que se recomienda el uso de este servidor de aplicaciones.
3. El entorno integrado de desarrollo utilizado tiene que ser Eclipse, versión, Oracle Enterprise Pack for Eclipse 11g.

11.3 — ~~Uso del sistema de seguridad (desactualizado)~~

La configuración de la seguridad en UDA aprovecha la inyección de dependencias que proporciona Spring Framework. Por tanto, tal y como se realiza en el resto de servicios de la aplicación, la configuración del sistema de seguridad se realiza mediante beans y propiedades que se describen en el fichero de configuración correspondiente.

La configuración de seguridad se puede encontrar en los siguientes ficheros:

- WEB-INF/spring/security-core-config.xml → Contiene la configuración de los beans que proporcionan funcionalidad al núcleo del sistema de seguridad. Este fichero no se ha de editar bajo ningún concepto, a no ser que se sepa muy bien lo que se está haciendo.
- META-INF/spring/security-config.xml → Contiene la configuración de seguridad para la capa de servicios de negocio. Se securizan los servicios referenciados por expresiones regulares con los roles indicados.

```
<security:global-method-security>
  <security:protect-pointcut expression="execution( *
com.ejje.*.service...find*(..))" access="ROLE_UDA"/>
  <security:protect-pointcut expression="execution( *
com.ejje.*.service...add*(..))" access="ROLE_UDA"/>
  <security:protect-pointcut expression="execution( *
com.ejje.*.service...remove*(..))" access="ROLE_UDA"/>
  <security:protect-pointcut expression="execution( *
com.ejje.*.service...update*(..))" access="ROLE_UDA"/>
</security:global-method-security>
```

- WEB-INF/spring/security-config.xml → Contiene la configuración de seguridad para las URLs de las JSPs y la capa de control. Se securizan las URLs referenciadas por expresiones regulares, a nivel de Rol y autenticación.

```
<bean id="filterSecurityInterceptor"
class="org.springframework.security.web.access.intercept.FilterSecurityInt
erceptor">
  <property name="authenticationManager" ref="authenticationManager"
/>
  <property name="accessDecisionManager" ref="affirmativeBased" />
  <property name="securityMetadataSource">
    <security:filter-security-metadata-source use-
expressions="true">
      <security:intercept-url pattern="/"
access="isAuthenticated()" />
      <security:intercept-url pattern="/logout"
access="isAuthenticated()" />
      <security:intercept-url pattern="/**" access="permitAll"
/>
      <security:intercept-url pattern="/ordering/**"
access="hasRole('ROLE_UDA')"/>
      <security:intercept-url pattern="/part/**"
access="hasRole('ROLE_UDA')"/>
      <security:intercept-url pattern="/vendor/**"
access="hasRole('ROLE_UDA')"/>
      <security:intercept-url pattern="/lineitem/**"
access="hasRole('ROLE_UDA')"/>
    </security:filter-security-metadata-source>
  </property>
</bean>
```

```

        <security:intercept-url pattern="/payment/**"
access="hasRole('ROLE_UDA')"/>
        <security:intercept-url pattern="/vendorPart/**"
access="hasRole('ROLE_UDA')"/>
        <security:intercept-url pattern="/present/**"
access="hasRole('ROLE_UDA')"/>
        </security:filter-security-metadata-source>
    </property>

</bean>

```

Además, en este mismo fichero se define cuantos minutos se va a cachear la información de seguridad referente a cada petición. Cuanto mayor sea este número, menor número de peticiones se realizarán a XLNetS, con lo que el rendimiento de las aplicaciones mejorará substancialmente. Todo ello irá en detrimento del nivel de seguridad de las aplicaciones, ya que cuanto mayor menor sea el uso de XLNetS, el nivel de sincronización de los permisos modificables en LDAP y las aplicaciones finales, será menor.

```

<bean id="perimetralSecurityWrapper"
class="com.ejje.x38.security.PerimetralSecurityWrapperN38Impl">
    <property name="xlnetCachingPeriod" value="2"/>
</bean>

```

Por tanto, la configuración manual de la seguridad de una aplicación necesitará modificar el contenido de estos últimos dos ficheros.

Además de todo lo indicado, existe la posibilidad de trabajar con un proveedor de seguridad ficticio, muy útil en entornos de prueba o en otros en los que no se dispone de conectividad a XLNetS u otro sistema de seguridad.

Para utilizar el proveedor ficticio, hay que editar el fichero WEB-INF/spring/security-config.xml, desactivar el primer Bean identificado como perimetralSecurityWrapper original y activar el segundo, como se muestra en el siguiente ejemplo.

```

<!-- <bean id="perimetralSecurityWrapper"
class="com.ejje.x38.security.PerimetralSecurityWrapperN38Impl">-->
<!-- <property name="xlnetCachingPeriod" value="2"/>-->
<!-- </bean>-->
    <bean id="perimetralSecurityWrapper"
class="com.ejje.x38.security.PerimetralSecurityWrapperMockImpl">
        <property name="principal" value="USER_UDA"/>
        <property name="roles">
            <list>
                <value>UDA</value>
            </list>
        </property>
        <property name="uidSession" value="1290789636844"/>
    </bean>

```

De esta manera, serializarán todas las peticiones en nombre del usuario UDA, con el rol ROLE_UDA y con el identificador de sesión 1290789636844. Todos estos valores son perfectamente editables.

11.4 Funcionamiento conceptual

Cuando un usuario requiere un recurso securizado, el comportamiento del sistema de seguridad será el siguiente (en el ejemplo, se muestra el comportamiento utilizando XLNetS):

1. La petición entra por la cadena de filtros definida en Spring Security detectando si se ha realizado una petición http a un recurso securizado y comprobando en caso afirmativo si existe o no un usuario autenticado válido.
2. Si no existe usuario autenticado, Spring Security delega la autenticación en el módulo de login proporcionado por XLNetS.



Ilustración 95. Login en XLNetS.

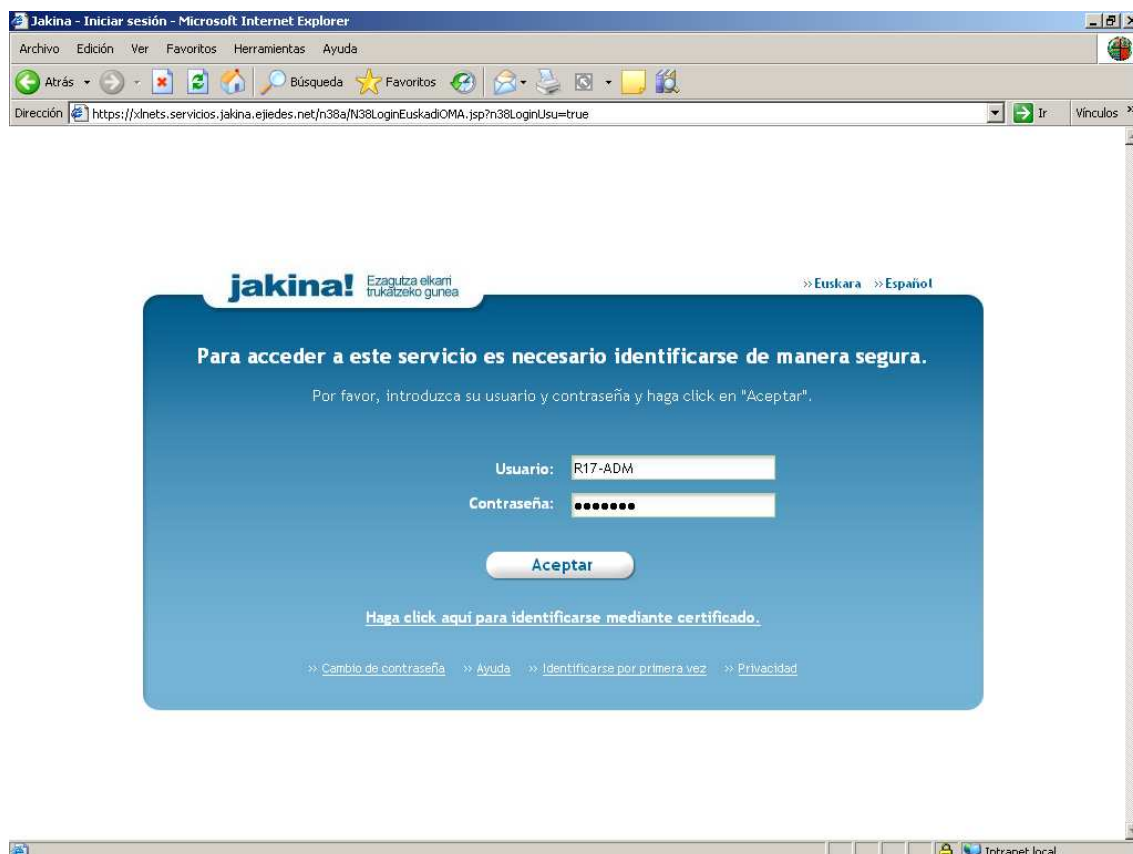


Ilustración 96. Autenticación en XLNetS.

3. Una vez autenticado, Spring Security inserta en sesión las credenciales y autorizaciones del usuario validado, comprobando si dentro de las autorizaciones existe una que le permita acceder al recurso securizado.
4. En caso de que no tenga permisos se redirige a la página de acceso denegado definida en el fichero de seguridad no editable anteriormente mencionado.
5. En caso de si tener permisos se accede al recurso solicitado.

12 Logging

El sistema de *logs* es la infraestructura de las aplicaciones encargada de gestionar el registro oficial de eventos en el tiempo. Dicho sistema interno registra datos y/o información sobre quién, qué, cuándo, dónde y por qué (*who, what, when, where* y *why*) se produce un evento en una aplicación. Los distintos registros pueden ser almacenados en soportes de diferente naturaleza e índole, aunque el más habitual es el almacenaje en ficheros. La información almacenada, es de considerable utilidad a la hora de rastrear incidencias, obtener información estadística, realizar auditorías,...

Al igual que para otros ámbitos estructurales de un aplicativo, UDA proporciona un sistema de *logs* integrado con el código de utilidades comunes y con el código que genera automáticamente. La información que se recopila, principalmente, procede de **tres fuentes**:

- **Log de infraestructura:** Estos *logs* agrupan toda la información aportada por el código de utilidades/infraestructuras comunes de UDA y las distintas tecnologías que engloba UDA (tanto trazas como incidencias).
- **Log estructurales de aplicación:** Son un conjunto de trazas que UDA integra automáticamente en las aplicaciones. Estos *logs*, se apoyan en [Aspect Oriented Programming](#) (AOP) para de dejar constancia de las operaciones que se realizan en y entre las distintas capas de la aplicación.
- **Trazas de usuario:** *logs* especificados por los programadores de las aplicaciones. Dichos registros tienen como objetivo dejar constancia de ciertos eventos, acciones o estados considerados significativos por los programadores.

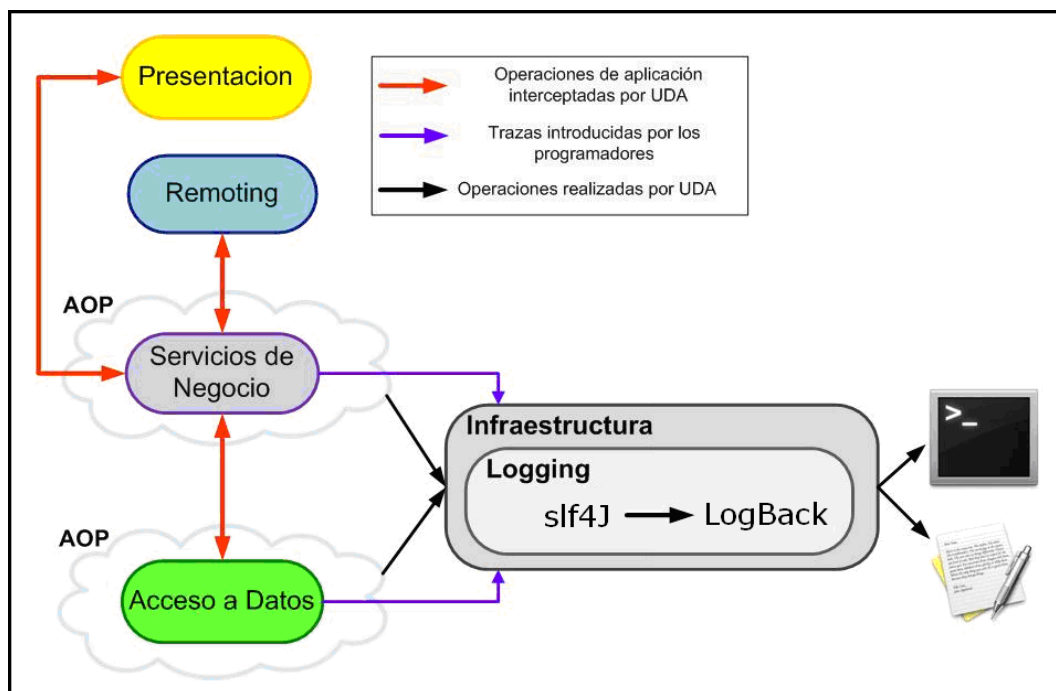


Ilustración 97. Sistema de *Logging*.

La ilustración superior, muestra un diagrama general de los mecanismos y tecnologías que integran el sistema de *logs*. Como se puede observar, el flujo de información se transfiere a la infraestructura, apoyada en *slf4j*, para que esta a su vez la gestione, formatee y almacene de forma apropiada.

La librería *slf4j* (*Simple Logging Facade for Java*), es el núcleo estructural en el que se apoya el sistema de *logs*. Dicha librería, no es más que una fachada o abstracción para varios *logging frameworks* (por ejemplo: *java.util.logging*, *log4j* o *logback*) que proporciona la independencia entre el código generado y el proveedor de *logs* utilizado. Gracias a este patrón de desarrollo, se abstrae el código generado de la tecnología utilizada y se permite, de forma sencilla, el cambio o actualización del *logging framework* sin tener que modificar el código previamente generado.

La conexión de *slf4j* con el *logging framework* se realiza en tiempo de compilación, según la librería puente que se use (para mas información leer la documentación oficial de *slf4j* <http://www.slf4j.org/docs.html>), por lo que no se corren riesgos de interacción cortocircuitada, de uso de *logging frameworks* no deseados o de pérdidas de rendimiento por el sistema de *logs* (como le podía pasar a *Jakarta Commons Logging*).

El propio *slf4j* es la evolución de *Jakarta Commons Logging* y mediante una librería (*jcl-over-slf4j.jar*), incorporada en UDA, se podría redirigir los *logs* de las infraestructuras que usan *Jakarta Commons Logging* a *slf4j*. Gracias a esta medida se maximiza la canalización de los *logs* reportados por las distintas tecnologías integradas en UDA.

12.1 LogBack

Entre todos los posibles *logging frameworks* para la generación de trazas disponibles, incluido *log4j*, se ha elegido a *logBack* como complemento a *slf4j* en el sistema de *logs*. Esta decisión no responde a una selección casual sino a las características y perspectivas evolutivas de las que dispone *logBack*.

Una de las señas de identidad de *logBack*, con la que se presenta en sociedad, es ser la evolución que los creadores de *log4j* aportan al ámbito de los *frameworks* para la generación de trazas. Este nuevo *framework*, como sucesor de *log4j*, busca mejorar y agilizar la gestión de *logs* que realizaba su predecesor. Además aporta un nuevo conjunto de funcionalidades y facilidades de configuración que lo hacen mucho más potente y fácil de usar.

Otro punto importante para su inclusión dentro de la infraestructura de UDA, es la implementación, de forma nativa, que hace del API de *slf4j*. Esto permite que no se deba incluir ninguna librería de conexión para que ambos trabajen (siendo así la conexión mucho mas limpia).

El funcionamiento estructural de *logBack*, conceptualmente, es similar al que tenía el propio *log4j*. Se mantienen los conceptos de *appender*, *logger*, *layout*, *encoder* y de nivel de severidad de traza. Este apartado, en sí, no es una comparativa entre *frameworks*, pero si que va a presentar las diferencias más llamativas que pueden afectar al desarrollo de aplicaciones (para mas información sobre *logBack* es posible visitar su página oficial en <http://logback.qos.ch/index.html>).

Con respecto a la integración y uso de *LogBack* en el sistema de *logs* de UDA, la primera cuestión que ha de considerarse es el nivel de severidad de las trazas. Al contrario de lo que ocurre con *log4j*, que dispone de 7 niveles de severidad (*all*, *debug*, *error*, *fatal*, *info*, *warn* y *off*), *logBack* trabaja con los mismos 6 niveles que *slf4j* (*debug*, *trace*, *info*, *warn*, *error* y *off*).

| level of request | effective level | | | | | |
|------------------|-----------------|-------|------|------|-------|-----|
| | TRACE | DEBUG | INFO | WARN | ERROR | OFF |
| TRACE | YES | NO | NO | NO | NO | NO |
| DEBUG | YES | YES | NO | NO | NO | NO |
| INFO | YES | YES | YES | NO | NO | NO |
| WARN | YES | YES | YES | YES | NO | NO |
| ERROR | YES | YES | YES | YES | YES | NO |

Ilustración 98. Diagrama de correspondencia según nivel de severidad.

Este ajuste de niveles, hace que la integración entre *slf4j* y *logBack* sea perfecta, sin tener niveles diferidos o sin correlación. Gracias a esto, se facilita el trabajo de los desarrolladores, al no tener que hacer asociaciones, y la posterior explotación de las trazas.

La segunda cuestión que varía entre el uso de *logBack* y el de *log4j*, es a nivel de configuración. Por cuestiones estructurales dependientes del ámbito para el que se creo originalmente UDA, *logBack* dispone de dos ficheros de configuración en UDA: *logback-test.xml* y *logback.xml*.

El *logback-test.xml* alberga la configuración por defecto del sistema de *logs*. Ubicado junto al código para permitir su carga automática al desplegar la aplicación o arrancar el servidor, dicho fichero determina la configuración base que se aplicará a las trazas de inicio de las distintas infraestructuras (*spring framework*, el propio *logBack*,...). Gracias a esta configuración por defecto, se centraliza la salida de todas las trazas, hasta las de arranque, a través del sistema de *logs* (más concretamente, por el fichero general de trazas).

Por otro lado, el fichero *logback.xml*, ubicado junto al resto de los ficheros de configuración y cargado por la infraestructura de *Spring*, alberga la configuración del sistema de *logs* específica de la aplicación. Un ejemplo de la ubicación de estos ficheros en una aplicación generada por UDA sería:

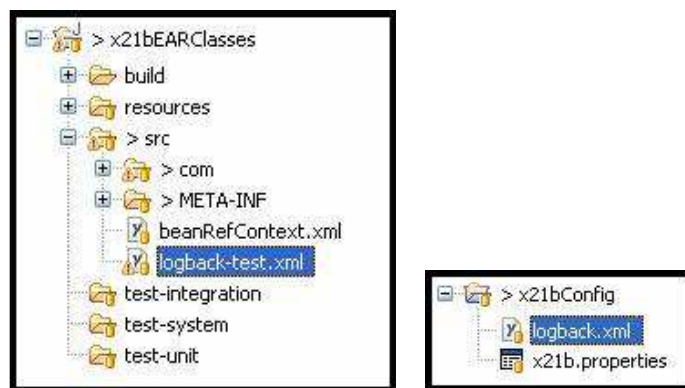


Ilustración 99. Ubicación ficheros de configuración de *logBack*.

En los casos en los que se desee modificar la configuración del sistema de *logs* (cambiar el *layout*, la ubicación o nombre de los ficheros,...), el fichero que se debería manipular es el *logback.xml*. En cualquier caso, si por circunstancias del proyecto, se deseara cambiar algún aspecto de la salida por defecto del sistema de *logs*, también se podría editar y cambiar el fichero *logback-test.xml* pero en principio no sería necesario.

Como ya se ha comentado un poco mas arriba, la carga de la configuración general del sistema de *logs* es realizada por *spring framework*. Dicha carga se apoya en una clase de carga integrada en las funcionalidades aportadas por UDA y en la configuración de los ficheros de carga de *spring*. Concretando un poco mas, la carga de la configuración del sistema de *logs* se especifica en el fichero *log-config.xml* de spring y ubicado en:

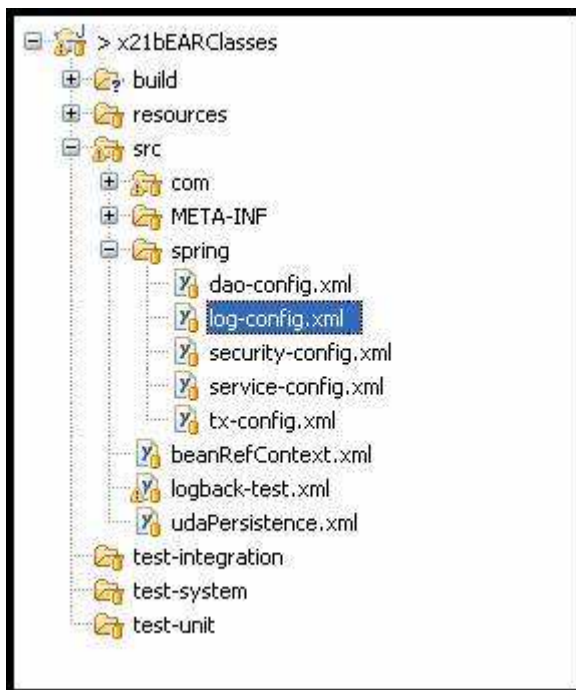


Ilustración 100. Ubicación del fichero *log-config.xml*.

La configuración que se aplica para determinar la carga de la configuración del sistema de *logs*, con respecto a *logBack*, requiere de dos parámetros:

```
<!-- Se especifica la inicialización de los log's mediante logback -->
<bean id="logSystemInitializer"
class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
    <property name="staticMethod"
value="com.ejje.x38.log.LogbackConfigurer.initLogging" />

    <property name="arguments">
        <list>
            <!-- Se especifica la ubicación del fichero de configuración de
logback (puede ser una ruta del classpath o absoluta) -->
            <value>x21b/logback.xml</value>
            <!-- Se especifica si se desea que se pinte el estado de la
configuración de logback por la salida de log correspondiente -->
            <value>true</value>
        </list>
    </property>
</bean>
```

Por un lado, se especifica la ubicación del fichero de configuración de *logBack*, pudiendo especificar tanto la ruta relativa en el *classpath* (recomendado) como la ruta absoluta del mismo; y por otro, se determina la escritura o no de las trazas de *logBack* asociadas a la carga de la configuración.

En caso de precisar el cambio del *framework* para la gestión de trazas, sería necesario cambiar o crear la clase asociada a la carga de la configuración (Spring dispone de forma nativa una clase encargada de cargar la configuración con *log4j*) para que Spring pueda arrancar correctamente el sistema de *logs*.

A nivel de configuración general, la integración de UDA y *logBack* se realiza con una parametrización básica para el sistema de *logs*. Para entender los entresijos de esta configuración básica, se va a analizar la configuración incluida en el fichero *logback-test.xml*. El fichero de configuración de la aplicación utiliza, con un ámbito y separación semántica superior, los mismos conceptos.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="false">

    <!-- ContextName of application -->
    <contextName>x21b</contextName>

    <!-- Loaded of the properties file of the application -->
    <property resource="${CONTEXT_NAME}/${CONTEXT_NAME}.properties" />

    <!-- Definition of the default appenders -->
    <!-- appender name="defaultOut" class="ch.qos.logback.core.ConsoleAppender"-->
    <appender name="defaultOutAppender" class="ch.qos.logback.core.rolling.
RollingFileAppender">
        <File>${log.path}/salidaEstandar_${CONTEXT_NAME}_${weblogic.Name}.log</File>
        <encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
            <layout class="com.ejje.x38.log.LogLayout">
                <appCode>${CONTEXT_NAME}</appCode>
                <instance>${weblogic.Name}</instance>
            </layout>
        </encoder>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <!-- rollover daily -->
            <fileNamePattern>${log.path}/salidaEstandar_${CONTEXT_NAME}
_${weblogic.Name}.%d{yyyy-MM-dd}.%i.gz </fileNamePattern>
            <timeBasedFileNamingAndTriggeringPolicy
class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
                <maxFileSize>100MB</maxFileSize>
            </timeBasedFileNamingAndTriggeringPolicy>
            <!-- 7-day history -->
            <maxHistory>6</maxHistory>
        </rollingPolicy>
    </appender>

    <!-- Root logger -->
    <root level="${log.level.salidaEstandar}">
        <appender-ref ref="defaultOutAppender" />
    </root>

</configuration>
```

Empezando desde el comienzo del fichero, la configuración de UDA para *logBack* determina las siguientes opciones de configuración:

- **ContextName:** Establece el código de aplicación propietaria de la configuración de *logBack*. Partiendo del código que genera UDA, mediante este único parámetro se determinan todos los elementos de configuración que precisa una aplicación asociados al sistema de *logs*. Gracias a esta estructura, se dispone de una configuración genérica que se aplica automáticamente a partir de un único dato.
- **Property resource:** Indica, a partir del *classpath* de la aplicación, el fichero del que se recogen las propiedades de configuración de la aplicación. A partir del *contextName* (`${CONTEXT_NAME}`) se determina la ubicación del fichero de *properties* generado por UDA para cada aplicación.
- **Appender:** Objeto de configuración que determina la forma, el soporte (fichero, BBDD,...), el formato y los diferentes aspectos derivados del almacenaje de las trazas. En los ficheros de configuración provisionados por UDA, para cada uno de estos *appenders*, además de determinar el fichero de almacenaje (*File*) y el formato (*layout*) que se aplicará a las trazas, se especifica la política de gestión que se aplicará a los ficheros (*rollingPolicy*) de *log*.
- **Loggers:** Los *loggers* son los encargados de asociar las distintas entradas de las trazas a los *appender* o *appenders* que finalmente las almacenarán, en función del paquete de la clase donde se invocan.

12.2 Ficheros de *logs*

El objetivo principal del sistema de *logs* es registrar y almacenar la información aportada por las trazas de la aplicación. Pero falta definir dónde se guarda y qué sentido semántico y temporal tiene la información que se guarda.

En general, existen muchas alternativas para almacenar y agrupar la información que recopila el sistema de *logs*. Con idea de facilitar el orden, búsqueda y entendimiento de los *logs*, UDA plantea una estructura base de ficheros con un significado funcional asociado.

Por defecto, todos y cada uno de los ficheros de *log* se ubican en la carpeta *log* asociada a la aplicación en el directorio datos. Esta ubicación, por defecto, se adopta partiendo de la infraestructura donde nace UDA y para separar los componentes de aplicación (código fuente, recursos estáticos, recursos de configuración, datos de salida), pero no es obligatoria y puede cambiarse modificando el fichero *logback.xml*.

Por ejemplo, para una aplicación con código *x21a*, los ficheros del sistema de *logs* se almacenarían en: ***/datos/x21a/log***.



Ilustración 101. Ubicación de los ficheros de log.

Como se puede observar en la imagen anterior, el sistema de *logs* tiene como salida 6 ficheros (5 más uno opcional). Cada uno de ellos, aloja un perfil o varios perfiles asociados a las trazas. Nótese que, para la identificación total de cada uno de los ficheros de trazas asociado a una aplicación y a una instancia de servidor concreta, su nombre se compone del literal asociado al sentido funcional del fichero, del código/nombre de la aplicación a la que pertenecen y del nombre de la instancia en la se alberga la aplicación. Por ejemplo:

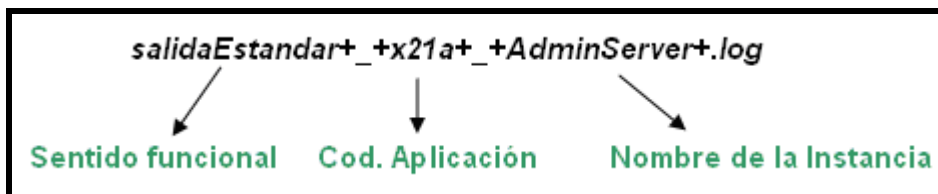


Ilustración 102. Nombre del fichero de trazas.

Se detalla, a continuación, el sentido funcional de cada uno de los ficheros de trazas generados por UDA:

- **Salida Estándar:** Es el fichero de trazas que almacena las entradas de todos los elementos, tanto de la aplicación como de la infraestructura (que usen *slf4j* o *Jakarta Commons Logging*), que utilicen el sistema de *logs* de la aplicación. Es decir, en este fichero se almacenan todas las trazas de todos elementos de la aplicación. Las únicas excepciones a la salida de trazas por el fichero de salida estándar son dos: las trazas de infraestructuras que no usen *slf4j* o *Jakarta Commons Logging* (usarían la consola de salida estándar o un *framework* de trazas directamente) y las trazas asociadas a la auditoria de BBDD. Estas dos circunstancias se deben a cuestiones técnicas insalvables y a la naturaleza sensible de los datos que se gestionen en la aplicación por cuestiones de seguridad.
- **Incidencias:** Por el fichero de incidencias salen todas las trazas asociadas a incidencias producidas en la aplicación y en las infraestructuras que usen *slf4j* o *Jakarta Commons Logging*. La configuración por defecto de UDA marca que, toda traza de *error* o *warning* producida por la aplicación se reporte en el fichero de incidencias y en el de salida estándar. Con esta configuración se centraliza la salida de incidencias y se facilita la revisión y/o explotación de posibles problemas (para el desarrollo es muy interesante ya que permite la consulta de errores de forma ágil y centralizada). Además no se emborriona el resto de ficheros de trazas con información de incidencias.
- **UdaTrazas:** La misión de este fichero de trazas es concentrar las distintas trazas generadas por el código de utilidades/infraestructuras comunes de UDA. Gracias a este tipo de configuración, se separan las trazas de usuario/aplicación de las trazas de UDA.
- **Cod. Aplic Trazas:** Este fichero recogerá todas las trazas de usuario/aplicación. Este fichero permite separar las trazas de usuario/aplicación de las de utilidades/infraestructuras de UDA lo que facilita enormemente la consulta de las trazas generadas por los propios usuarios/desarrolladores y las trazas

de gestión aportadas por UDA (por ejemplo: las trazas de interacción entre capas generadas a partir de aplicar aspectos).

Los ficheros de auditoría tienen como misión recopilar información de procesos más delicados o críticos. UDA ha considerado como tales el acceso a la aplicación y las consultas de base de datos.

- **Auditoría de Acceso:** El fichero de auditoría de acceso tiene como misión recopilar todas las trazas que tengan que ver con peticiones realizadas al sistema de gestión de seguridad implementado en la aplicación.
- **Auditoría de BBDD:** El fichero de auditoría de BBDD tiene como misión recopilar todas las *queries* realizadas sobre base de datos y los datos que se obtienen en cada una. Por cuestiones de seguridad asociadas a la protección de datos (LOPD), por defecto, UDA no crea la infraestructura necesaria para la ejecución de la auditoría de datos y mantiene separada estas trazas de la salida estándar. En un apartado posterior se explica qué pasos se deben dar para activar este sistema.

Como en la mayoría de los sistemas de *logs*, UDA marca por defecto una política de gestión de los ficheros de trazas. Cada uno de los ficheros activos de trazas se versionará con el cambio de día y con tamaños superiores a 100 MB. Para reducir el espacio necesario para guardar los ficheros de traza, cuando el fichero deja de ser activo se comprime en formato *GZip*. Y para que los ficheros de *log* respeten la capacidad de disco físico y mantengan un orden temporal apropiado, UDA, por defecto, mantiene los ficheros por un máximo de siete días.

12.3 Formato de las trazas

Por defecto UDA genera las trazas con un formato específico:

```
##|Fecha y hora ~~ UID Sesión ~~ Dirección IP cliente ~~ Usuario ~~ Puesto ~~
Código de aplicación ~~ Instancia Weblogic ~~ Subsistema funcional ~~
Identificador único de hilo ~~ Clase ~~ Criticidad ~~ Mensaje ~~ Información
adicional |##
```

- Fecha y hora: Momento en el que se genera la traza.
- UID Sesión: El identificador de sesión de XLNetS.
- Dirección IP del cliente: Dirección IP del que realiza la petición.
- Usuario: Usuario XLNetS conectado.
- Puesto: Puesto de XLNetS.
- Código de aplicación: Código de la aplicación que genera la traza.
- Instancia WebLogic: Nombre de la instancia de WebLogic que genera la traza.
- Subsistema funcional: Nombre del subsistema se genera la traza, que puede ser:
 - Data subsystem: Subsistema de librerías de gestión de base de datos, como Spring JDBC o EclipseLink.
 - Logic subsystem: Subsistema de servicios de negocio y acceso a datos.
 - Web subsystem: Subsistema de control.

- Incidence subsystem: Subsistema de captura de incidencias.
- Trace subsystem: Subsistema de trazas de usuario y resto de componentes de UDA.
- Access subsystem: Subsistema de acceso a aplicaciones, principalmente filtros.

NOTA: La variable app representa al código de aplicación.

- Identificador único de hilo: Cada hilo de ejecución lleva asociado un identificador único que se reflejará en todas las trazas que produzca dicho hilo. Así, se da cabida a la trazabilidad de las operaciones.
- Clase: Especificación del nombre de clase, con su *package*, que genera la traza.
- Criticidad: Nivel de gravedad de la traza:
 - TRACE: Solo para desarrollo.
 - DEBUG: Solo para debug de la aplicación
 - INFO: Solo para debug y pruebas.
 - WARN: Para incidencias a tener en cuenta (no bloqueante).
 - ERROR: Para indicar errores (bloqueante).
- Mensaje: Texto que se desea mostrar.
- Información adicional: Texto extraordinario que se desea mostrar (es opcional).

La naturaleza del formato de trazas que se generan por defecto responde a las necesidades derivadas de la infraestructura sobre la que nace UDA (concretamente para facilitar el uso de un sistema de explotación de logs). Este formato de traza, lo determina UDA a través de un *layout* específico asociado a cada uno de los *appenders* de la aplicación.

Mantener este formato de traza no es, ni mucho menos, obligatorio. Los usuarios que así lo precisen pueden definir su propio *layout*, o usar el genérico de *logBack*, y aplicarlo sin ningún tipo de restricción. Es posible cambiar los datos de salida y el formato o el orden de los mismos. Pero si el marco de trabajo es Ejje, lo más razonable es mantener el esquema por defecto ya que este está diseñado para cubrir todas las necesidades del entorno.

A la hora de personalizar el formato de salida de las trazas, es importante tener en cuenta que el sistema de *logs* integra y utiliza el contexto MDC "Mapped Diagnostic Context". Este contexto permite almacenar datos en diferentes puntos de la aplicación y luego extraerlos para escribir las trazas. A nivel estructural, UDA almacena directamente en el MDC, para su posible posterior uso, los siguientes datos:

- **Datos de usuario:** En relación a los datos asociados a los usuarios que acceden a la aplicación, UDA, a través de la librería de utilidades, almacena en el MDC el nombre del usuario, el puesto y el código de la sesión de seguridad.
- **NOINTERNALACCES:** Cuando se produce un acceso externo a las aplicaciones, tanto vía *http* como vía *rmi*, la infraestructura de UDA almacena el tipo de acceso para que este pueda ser reflejado en las trazas.
- **IP de acceso:** Cada vez que las peticiones generadas con UDA reciben una petición, la infraestructura de esta guarda la IP del usuario peticionario para poder presentarla en las trazas de la aplicación.

Además de estos datos estructurales recopilados en el MDC, el sistema de trazas de UDA tiene en cuenta la posible especificación de datos adicionales (sobre todo trazas de excepciones) y de subsistemas funcionales

determinados. En caso de estar almacenados estos datos en el MDC, el sistema de trazas los tendrá en cuenta y los plasmará en las trazas correspondientes.

12.4 Trazas de usuario

Como se ha comentado en apartados anteriores, la infraestructura de *logs* se apoya en *slf4j* para abstraer el sistema de *logs* del *logging framework* utilizado. Gracias a este modelo estructural, se independiza la gestión de *logs* del proveedor de los mismos pudiendo sustituirlo de forma sencilla. Cualquier nueva traza especificada por los usuarios, no debe ser programada mediante el *logging framework*, si no mediante la infraestructura de *slf4j*. Si no se procede de esta manera, las trazas inicialmente seguirán apareciendo pero se incurrirán en dependencias de código que a futuro pueden derivar en revisiones, cambios y reingeniería de código totalmente innecesarios.

Cuando cualquier programador desee especificar cualquier tipo de traza o incidencia mediante el sistema de *logs*, sea en la capa de control, en la de servicio o en la de acceso a datos, previamente deberá dotar a la clase que escribirá las trazas de su correspondiente *logger*.

```
package com.ejje.x21a.service;

...

import org.slf4j.LoggerFactory;
import org.slf4j.Logger;

...

public class OrderingServiceImpl implements OrderingService {

    private static final Logger logger =
        LoggerFactory.getLogger(OrderingServiceImpl.class);
```

Una vez especificado el *logger* asociado a la clase "OrderingServiceImpl.class", éste puede ser utilizado para escribir trazas en cualquier método de la clase de la siguiente manera:

```
logger.info("Logging data: "+ordering);
```

La traza que se generará a raíz de ejecutar la sentencia anterior, aparecerá en el fichero asociado a las trazas de la aplicación (*Cod. Aplic Trazas*) con el siguiente aspecto:

```
##| 2011/01/26 15:39:13:942 ~~ 1290789636844 ~~ 127.0.0.1 ~~ USER_UDA ~~ myPosition ~~ x21a ~~
TestServer ~~ Logic Subsystem ~~ 2 ~~ com.ejje.x21a.service.OrderingServiceImpl ~~ INFO ~~ Logging
data: com.ejje.x21a.model.Ordering Object {
  orderid: null
  lastupdate: null
  status: S
  shipmentinfo: null
  discount: null
  present: null
} ~~ |##
```

El resto de parámetros que se reflejan en la traza y que el desarrollador no ha incluido en la llamada al logger, son incluidos automáticamente por UDA.

12.5 Logs de aplicación e incidencia

El sistema de logging integra un conjunto de interceptores AOP. Dichos interceptores realizan una doble función, por un lado, capturan todas las llamadas que se realizan a los servicios de negocio y a la capa de acceso a datos, y por otro lado, recogen las excepciones que pudieran surgir en ejecución.

Por ejemplo, la llamada a un servicio tendrá como resultado un conjunto de trazas, fruto de la intercepción, similares a esta:

```
##| 2011/01/26 15:39:13:925 ~~ 1290789636844 ~~ 127.0.0.1 ~~ USER_UDA ~~ myPosition ~~ x21a ~~  
TestServer ~~ Logic Subsystem ~~ 2 ~~ com.ejje.x38.log.LoggingAdviceImpl ~~ TRACE ~~  
OrderingServiceImplexecution(OrderingService.findAll(..)) ~~ [com.ejje.x21a.model.Ordering Object {  
 orderid: null  
  lastupdate: null  
  status: S  
  shipmentinfo: null  
  discount: null  
  present: null  
}, com.ejje.x38.dto.JQGrid Object {  
  _search: null  
  nd: null  
}] |##
```

Se puede observar, que la traza muestra una llamada al servicio “OrderingService”, más concretamente, al método “findAll”, que recibe los objetos “Ordering” y “JQGrid”, que se desglosan en la última parte de la traza.

Dentro de los datos que se muestran en las trazas de los interceptores, se incluyen los datos resultantes de los servicios o accesos a base de datos. Dichos datos, según los requisitos de las aplicaciones, puede que estén bajo la protección de la ley orgánica de protección de datos (LOPD) y no deban ser almacenados en el sistema de *logs*. Para evitar este tipo de situaciones y cumplir de forma ágil con la protección de datos, la infraestructura de UDA solo presenta los datos resultantes de las llamadas si el *appender* asociado a las trazas de UDA (**UdaTrazas**) tiene asignado el nivel de trazas a “**TRACE**”.

En los casos en los que UDA captura excepciones, las trazas que se plasman como resultado de la intercepción tendrán un aspecto similar a este:

```
##| 2011/01/26 15:39:14:052 ~~ 1290789636844 ~~ 127.0.0.1 ~~ USER_UDA ~~ myPosition ~~ x21a ~~  
TestServer ~~ Incidence Subsystem ~~ 2 ~~ com.ejje.x38.log.IncidenceLoggingAdviceImpl ~~ ERROR ~~  
java.lang.RuntimeException: [com.ejje.x21a.model.Ordering Object {  
 orderid: 33  
  lastupdate: 2010-12-23 00:00:00.0  
  status: S  
  shipmentinfo: deffws  
  discount: 12  
  present: com.ejje.x21a.model.Present Object {  
   orderid: 0  
    description: null  
    name: null  
    barcode: null  
    available: false
```

```
}
}, com.ejje.x21a.model.Ordering Object {
 orderid: 4422
  lastupdate: 2010-12-29 00:00:00.0
  status: S
  shipmentinfo: aaaaaa
  discount: 45
  present: com.ejje.x21a.model.Present Object {
   orderid: 0
    description: null
    name: null
    barcode: null
    available: false
  }
}, com.ejje.x21a.model.Ordering Object {
 orderid: 6666
  lastupdate: 2010-12-29 00:00:00.0
  status: S
  shipmentinfo: aaaa
  discount: 56
  present: com.ejje.x21a.model.Present Object {
   orderid: 0
    description: null
    name: null
    barcode: null
    available: false
  }
}
}1
    at com.ejje.x21a.service.OrderingServiceImpl.findAll(OrderingServiceImpl.java:76)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
.....
```

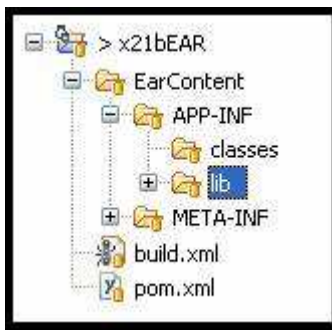
Por defecto, todas estas trazas se generan con el nivel de gravedad ERROR.

12.6 Auditoría de accesos a Base de Datos (jdbclogslog)

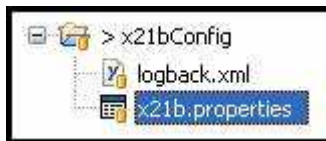
Dentro de los servicios e infraestructuras que ofrece/integra UDA, existe la posibilidad de auditar los accesos y consultas a base de datos. Este sistema de *logs*, adicional, permite recopilar y conocer todas las consultas que se lanzan contra la base de datos y los datos que ésta devuelve. Este sistema de auditoría está diseñado para que se pueda utilizar tanto en aplicaciones que usen tecnología **JPA** como en aplicaciones que usen tecnología **Spring JDBC** para la capa de acceso a datos.

UDA, por defecto, no configura este tipo de *log*, pero, por si se desea activar, a continuación se explican, detalladamente, los pasos que se deben seguir para su activación en una aplicación.

1. Debe estar presente la librería **jdbclogslog-1.0.5.jar** en el proyecto *EAR* de la aplicación. Para obtener la dependencia se puede incluir el *pom* de la aplicación generado por UDA y descargarse con el resto de librerías.



2. Modificar el fichero de configuración de propiedades (xxx.properties) de la aplicación para indicar el nivel de traza del *appender* asociado.



Se debe añadir la siguiente línea:

```
log.level.auditoriaBBDD=INFO
```

3. Se deberá incluir el *DataSource* que interceptará las peticiones a base de datos en los ficheros de spring apropiados. Concretamente, se deberá modificar el fichero "dao-config.xml" ubicado en el proyecto xxxEARClasses de la aplicación.

```
<!--Configuration of dataSource with DDBB audit -->
<jee:jndi-lookup id="aplicDataSource"
    jndi-name="x21a.x21aDataSource" resource-ref="false"/>

<bean id="dataSource"
    class="org.jdbcdslog.ConnectionPoolDataSourceProxy">
    <property name="targetDSDirect" ref="aplicDataSource" />
</bean>
```

NOTA: En este ejemplo el DataSource al que se conecta es x21a.x21aDataSource

En caso de tratarse de una aplicación que utiliza *JPA* en lugar de *Spring JDBC*, se deben realizar las siguientes modificaciones extra:

- a. Añadir la cabecera "xmlns:jee" para soportar la etiqueta <jee> (en negrita):

```
<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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/spring-jee-3.0.xsd">
```

b. Añadir la propiedad "dataSource" (en negrita):

```
<!-- Configuration of the JPA Management -->
<bean id="jtaEntityManagerFactory"
      class="org.springframework.orm.jpa.LocalContainerEntityMan
gerFactoryBean">
    <property name="persistenceUnitName" value="XXX_JTA" />
    <property name="dataSource" ref="dataSource" />
    <property name="persistenceXmlLocation" value="classpath:udaPersistence.xml" />
    <property name="loadTimeWeaver">
        <bean class="org.springframework.instrument.classloading.
weblogic.WebLogicLoadTimeWeaver" />
    </property>
    <property name="jpaVendorAdapter">
        <bean class="org.springframework.orm.jpa.vendor.
EclipseLinkJpaVendorAdapter">
            <property name="database" value="ORACLE" />
            <property name="showSql" value="true" />
        </bean>
    </property>
</bean>
```

4. Modificar el fichero logback.xml (xxxConfig/logback.xml) para incluir el *appender* y los *loggers* necesarios para que se generen las trazas correctamente.

```
<appender name="auditoriaBBDDAppender"
class="ch.qos.logback.core.rolling.RollingFileAppender">
    <File>${log.path}/auditoriaBBDD_${CONTEXT_NAME}_${weblogic.Name}.log</File>
    <filter class="com.ejje.x38.log.UdaLogFilter" />
    <encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
        <layout class="com.ejje.x38.log.LogLayout">
            <appCode>${CONTEXT_NAME}</appCode>
            <instance>${weblogic.Name}</instance>
        </layout>
    </encoder>
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <!-- rollover daily -->
        <fileNamePattern>${log.path}/auditoriaBBDD_${CONTEXT_NAME}
_${weblogic.Name}_%d{yyyy-MM-dd}_%i.gz</fileNamePattern>
        <timeBasedFileNamingAndTriggeringPolicy
class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
            <maxFileSize>100MB</maxFileSize>
        </timeBasedFileNamingAndTriggeringPolicy>
```

```
        <!-- 7-day history -->  
        <maxHistory>6</maxHistory>  
    </rollingPolicy>  
</appender>
```

```
<logger name="org.jdbcdslog" level="${log.level.auditoriaBBDD}" additivity="false">  
    <appender-ref ref="auditoriaBBDDAppender"/>  
</logger>  
  
<logger name="org.eclipse.persistence-connection" level="${log.level.auditoriaBBDD}"  
additivity="false">  
    <appender-ref ref="auditoriaBBDDAppender"/>  
</logger>  
  
<logger name="org.eclipse.persistence" level="${log.level.auditoriaBBDD}" additivity="false">  
    <appender-ref ref="auditoriaBBDDAppender"/>  
</logger>
```

13 Glosario de anotaciones

A continuación se comentan resumidamente las anotaciones utilizadas por UDA:

| Anotación | Clase | Propósito |
|---------------------|--|---------------------------|
| @Autowired | org.springframework.beans.factory.annotation.Autowired | Inyección de dependencias |
| @Transactional | org.springframework.transaction.annotation.Transactional | Transaccionalidad |
| @Service | org.springframework.stereotype.Service | Ciclo de vida |
| @Controller | org.springframework.stereotype.Controller | Ciclo de vida |
| @RequestMapping | org.springframework.web.bind.annotation.RequestMapping | Navegación |
| @ResponseBody | org.springframework.web.bind.annotation.ResponseBody | Paso de parámetros |
| @RequestParam | org.springframework.web.bind.annotation.RequestParam | Paso de parámetros |
| @PathVariable | org.springframework.web.bind.annotation.PathVariable | Paso de parámetros |
| @ExceptionHandler | org.springframework.web.bind.annotation.ExceptionHandler | Gestión de excepciones |
| @RequestMethod | org.springframework.web.bind.annotation.RequestMethod | Navegación |
| @Repository | org.springframework.stereotype.Repository | Ciclo de vida |
| @Override | JDK | Metadatos |
| @PersistenceContext | javax.persistence.PersistenceContext | Persistencia |
| @StaticModel | javax.persistence.metamodel.StaticMetamodel | Persistencia |
| @Generated | javax.annotation.Generated | Metadatos |
| @Entity | javax.persistence.Entity | Persistencia |
| @Table | javax.persistence.Table | Persistencia |
| @EmbeddedId | javax.persistence.EmbeddedId | Persistencia |
| @Column | javax.persistence.Column | Persistencia |
| @AttributeOverrides | javax.persistence.AttributeOverrides | Persistencia |
| @AttributeOverride | javax.persistence.AttributeOverride | Persistencia |
| @FetchType | javax.persistence.FetchType | Persistencia |
| @JoinColumn | javax.persistence.JoinColumn | Persistencia |

| | | |
|-----------------------|---|---------------|
| @ManyToOne | javax.persistence.ManyToOne | Persistencia |
| @Embeddable | javax.persistence.Embeddable | Persistencia |
| @OneToOne | javax.persistence.OneToOne | Persistencia |
| @Id | javax.persistence.Id | Persistencia |
| @OneToMany | javax.persistence.OneToMany | Persistencia |
| @ManyToMany | javax.persistence.ManyToMany | Persistencia |
| @JoinColumns | javax.persistence.JoinColumns | Persistencia |
| @Lob | javax.persistence.Lob | Persistencia |
| @Temporal | javax.persistence.Temporal | Persistencia |
| @PrimaryKeyJoinColumn | javax.persistence.PrimaryKeyJoinColumn | Persistencia |
| @UniqueConstraint | javax.persistence.UniqueConstraint | Persistencia |
| @JsonDeserialize | org.codehaus.jackson.map.annotate.JsonDeserialize | Serialización |
| @JsonSerialize | org.codehaus.jackson.map.annotate.JsonSerialize | Serialización |
| @JsonIgnore | org.codehaus.jackson.annotate.JsonIgnore | Serialización |

A éstas, habría que añadir las anotaciones de validación que los desarrolladores pueden añadir al modelo, que se pueden encontrar en un apartado de la [documentación oficial de Hibernate Validator](#).

14 Equivalencias conceptuales con Geremua 2

Para aquellos desarrolladores que provengan de trabajar con el Framework de EJIE/EJGV, Geremua 2, se establece una equivalencia conceptual entre los componentes de Geremua 2 y UDA, con el propósito de suavizar la curva de aprendizaje al iniciar un nuevo desarrollo con las nuevas Utilidades de Desarrollo de Aplicaciones.

El modelo arquitectónico implementado por Geremua 2 es el siguiente:

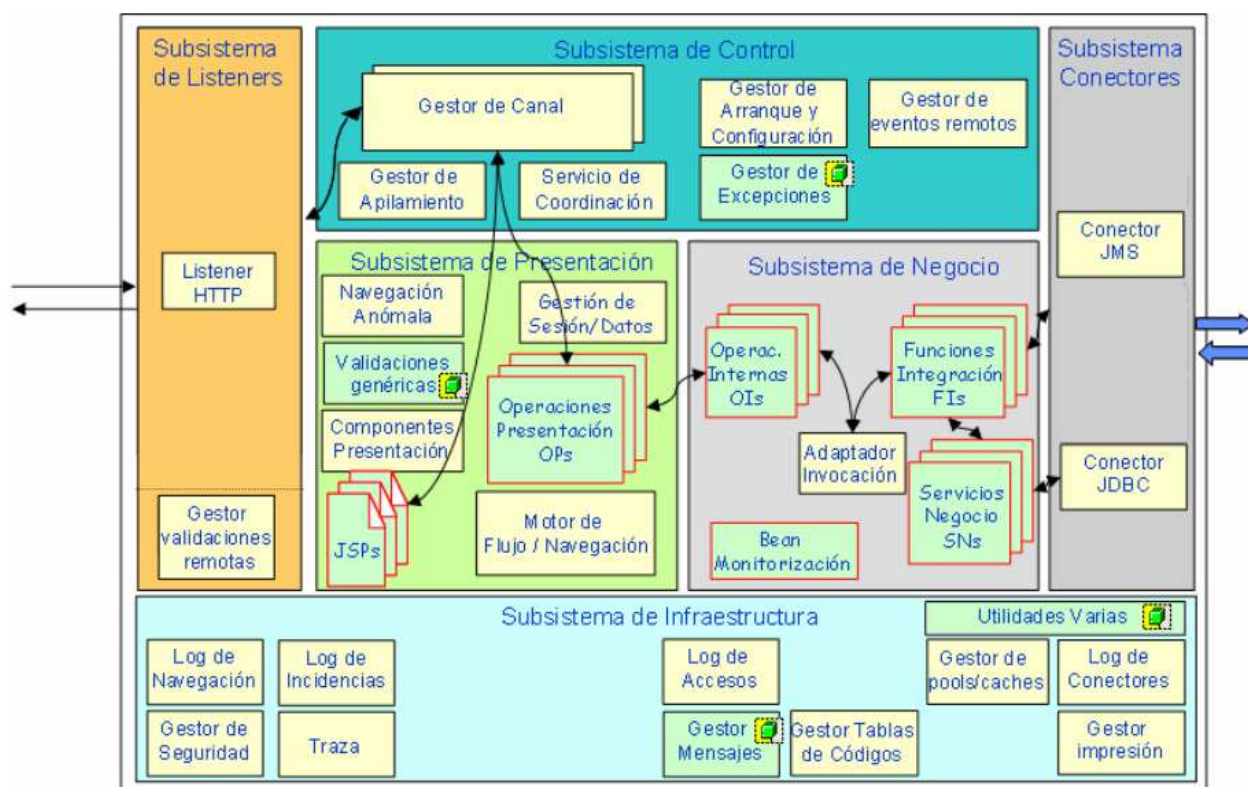


Ilustración 103. Modelo arquitectónico G2.

Es evidente que esta arquitectura es mucho más compleja que la de UDA, por lo que muchos de los componentes de los subsistemas no tendrán correspondencia en las nuevas Utilidades de Desarrollo de Aplicaciones.

A continuación, se hace una comparación, a grandes rasgos, entre los componentes mostrados en el diagrama superior y los componentes de UDA (explicados a lo largo del presente documento). Con el propósito de realizar una comparación más comprensible, se dividirá en tres bloques:

14.1 Componentes proporcionados por las aplicaciones

| Geremua 2 | UDA |
|-----------------------------|----------------------------------|
| Operaciones de Presentación | Llamadas a los componentes de la |

| | |
|--------------------------|---|
| | subcapa de control (o Controllers) |
| JSPs | JSPs |
| Operaciones Internas | Llamadas a los componentes de la capa de servicios de negocio |
| Funciones de Integración | Llamadas a los componentes de la capa de Remoting. |
| Servicios de Negocio | Servicios de Negocio |
| Bean de Monitorización | No aplica |

14.2 Componentes configurables por las aplicaciones

| Geremua 2 | UDA |
|----------------------------------|---|
| Operaciones Internas | Llamadas a los componentes de la capa de servicios de negocio |
| Operaciones de Presentación | Llamadas a los componentes de la subcapa de control (o Controllers) |
| Gestor de Validaciones Genéricas | Sistema de validación. |
| Gestor de Navegación Anómala | No aplica. |
| Gestión de excepciones | Sistema de Logging. |
| Mapeos | Traducción de parámetros HTTP a modelo de datos realizado por la capa de Control. |
| Gestor de mensajes | Resource Bundles de ficheros properties. |
| Gestor de caché | No aplica. |
| Gestor de tablas de códigos | No aplica. |
| Traza | slf4j y LogBack. |
| Log de Incidencias | Sistema de Logging. |
| Log de navegación | Sistema de Logging. |

| | |
|----------------------|---------------------|
| Log de accesos | Sistema de Logging. |
| Log de transacciones | Sistema de Logging. |
| Log de conectores | No aplica. |

14.3 Componentes estructurales

| Geremua 2 | UDA |
|-----------------------------|---|
| Gestor de Arranque | No aplica. |
| Servicio de coordinación | No aplica. |
| Gestor de eventos remotos | No aplica. |
| Gestor de apilamiento | No aplica. |
| Gestor de excepciones | Excepciones generadas por la subcapa de control, con información para el usuario. |
| Conector JMS | No aplica. |
| Conector JDBC | Proporcionado por el servidor. |
| Gestor de mensajes | Resource Bundles de ficheros properties. |
| Gestor de tablas de códigos | No aplica. |
| Gestor de seguridad | Sistema de seguridad. |
| Gestor de cachés | No aplica. |
| Gestor de Pool | Proporcionado por el servidor. |
| Gestor de Impresión | No aplica. |
| Gestor de Mail | No aplica. |
| Gestor de SMS | No aplica. |

14.4 Custom Tags

| Geremua 2 | UDA |
|---------------------------|---|
| Tags de Struts y añadidos | Tags de Spring, JSTL y patrones visuales RUP. |

15 Bibliografía

- Spring Framework: <http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/>
- JPA 2.0: <http://download.oracle.com/javaee/6/tutorial/doc/bnbpy.html>
- Hibernate Validator: <http://docs.jboss.org/hibernate/validator/4.1/reference/en-US/html/>
- EJB 3.0: http://download.oracle.com/docs/cd/E12840_01/wls/docs103/ejb30.html
- Oracle WebLogic: http://download.oracle.com/docs/cd/E12840_01/wls/docs103/sitemap.html
- SLF4J: <http://www.slf4j.org/>
- Logback: <http://logback.qos.ch/>