

Oracle ADF: desarrollo de una aplicación con ADF, EJB y JSF, en jDeveloper 11

Miguel Velasco Gomez-Pantoja

<http://tecno-aspirinas.blogspot.com>

Contenido

Introducción	6
Oracle Application Development Framework.....	6
EJB 3.0	8
Entorno de desarrollo.....	9
Definición del problema.....	9
Desarrollo.....	9
Organización del trabajo en jDeveloper	9
Creación de la base de datos	14
El proyecto Model.....	24
Creando las entidades	25
Configurando la unidad de persistencia	32
Revisando las anotaciones	34
Creando el EJB de Sesión	37
El proyecto ViewController.....	42
Creando páginas jspx	45
Dando contenido a las páginas	47
ADF Bindings.....	50
Gestión de notas.....	79
Edición de notas	85
Eliminar notas	86
Cuadros de diálogo y popups.....	86
Gestión de estados	94
Probando la aplicación	95
Puliendo detalles	96
Conclusiones	104
Referencias.....	105
Anexo: Abrir una aplicación existente con jDeveloper	106

Tabla de ilustraciones

Ilustración 1: Arquitectura de ADF.....	6
Ilustración 2: Capas de ADF.....	8
Ilustración 3: Nueva aplicación genérica	10
Ilustración 4: Crear una nueva aplicación.....	11
Ilustración 5: Crear proyecto	12
Ilustración 6: Configurar EJBs	13
Ilustración 7: Fichero jws.....	13
Ilustración 8: Nuevo diagrama de base de datos.....	15
Ilustración 9: Crear diagrama de base de datos	15
Ilustración 10: Especificar localización	16
Ilustración 11: Editar tablas – Columnas	17
Ilustración 12: Editar tablas – Clave primaria	18
Ilustración 13: Esquema de datos	19
Ilustración 14: Script de creación de base de datos	19
Ilustración 15: Crear conexión a la base de datos	20
Ilustración 16: Generar SQL a partir de objetos de la base de datos.....	21
Ilustración 17: Generar SQL a partir de objetos de base de datos 2.....	22
Ilustración 18: Generar SQL a partir de objetos de base de datos 3.....	23
Ilustración 19: Navegador de base de datos	23
Ilustración 20: Crear secuencia	24
Ilustración 21: Esquema Entity Manager	25
Ilustración 22: Galería de nuevos elementos 2.....	26
Ilustración 23: Nueva unidad de persistencia	26
Ilustración 24: Crear entidades a partir de tablas	27
Ilustración 25: Crear entidades a partir de tablas 2	28
Ilustración 26: Crear entidades a partir de tablas 3	29
Ilustración 27: Crear entidades a partir de tablas 4	30
Ilustración 28: Crear entidades a partir de tablas 5	31
Ilustración 29: Crear entidades a partir de tablas 6	31
Ilustración 30: Navegador de aplicaciones	32
Ilustración 31: Edición de persistence.xml.....	33
Ilustración 32: Edición de persistence.xml 2.....	33
Ilustración 33: Clase Temas	34
Ilustración 34: Relación uno a muchos en JPA.....	36
Ilustración 35: Nuevo EJB de Sesión	37
Ilustración 36: Crear session bean	38
Ilustración 37: Crear session bean 2	39
Ilustración 38: Crear session bean 3	40
Ilustración 39: Navegador de aplicaciones	40
Ilustración 40: Anotación Remote.....	41
Ilustración 41: Anotación Local.....	41
Ilustración 42: Anotaciones del session bean.....	41
Ilustración 43: Especificar opciones del session facade	42
Ilustración 44: Menú nuevo proyecto.....	43
Ilustración 45: Crear proyecto ADF ViewController	44
Ilustración 46: Crear proyecto ADF View Controller 2.....	45
Ilustración 47: Nueva página en el faces-config.xml	46
Ilustración 48: Crear página JSF	46
Ilustración 49: Paleta de componentes.....	47
Ilustración 50: Vista estructura	48
Ilustración 51: Paleta de componentes.....	49
Ilustración 52: Editor jspx de diseño	49
Ilustración 53: Crear nuevo menú	50

Ilustración 54: Crear Data Control	51
Ilustración 55: Seleccionar la interfaz del EJB	52
Ilustración 56: Nuevos elementos del proyecto Model.....	52
Ilustración 57: Cabecera JavaBean.....	53
Ilustración 58: Atributos.....	53
Ilustración 59: Atributo objeto	53
Ilustración 60: Métodos	54
Ilustración 61: Fichero DataControls	54
Ilustración 62: Paleta DataControls	55
Ilustración 63: Estructura EJB frente a DataControls.....	56
Ilustración 64: Editar columnas de la tabla	57
Ilustración 65: Navegador del proyecto ViewController	57
Ilustración 66: BindingContext y DataBindings	58
Ilustración 67: Fichero DataBindings	59
Ilustración 68: Sección de ejecutables	60
Ilustración 69: Sección de Bindings.....	60
Ilustración 70: Código fuente de un tabla	61
Ilustración 71: DataControls	62
Ilustración 72: Editar columnas de la tabla	63
Ilustración 73: Propiedades de la tabla.....	64
Ilustración 74: Crear formulario ADF	65
Ilustración 75: Editar campos de formulario.....	66
Ilustración 76: Enlace de datos de página.....	67
Ilustración 77: Menú en vista estructura	68
Ilustración 78: Crear regla de navegación	68
Ilustración 79: Asignar regla de navegación	69
Ilustración 80: Confirmar enlace de componentes.....	70
Ilustración 81: Arrastrar botón	70
Ilustración 82: Regla de navegación global	70
Ilustración 83: Asignar regla de navegación	71
Ilustración 84: Confirmar enlace de componentes.....	72
Ilustración 85: Editar enlace de lista	72
Ilustración 86: Diseño de nuevoTema	73
Ilustración 87: Código de persistTemas	73
Ilustración 88: Arrastrar persistTemas	74
Ilustración 89: Editar enlace de acción	75
Ilustración 90: Código de methodAction	75
Ilustración 91: Faces-config.xml	76
Ilustración 92: Editar campos de formulario.....	77
Ilustración 93: Código de list	79
Ilustración 94: Código de mergeTemas	79
Ilustración 95: Vista estructura del menú.....	80
Ilustración 96: Faces-config.....	80
Ilustración 97: Confirmar enlace de componentes.....	81
Ilustración 98: Editar campos de formulario.....	82
Ilustración 99: Código de richTextEditor	83
Ilustración 100: Editar enlace de acción	83
Ilustración 101: Códigos de persistNotas	84
Ilustración 102: Código de removeNotas.....	84
Ilustración 103: Propiedades de botón	85
Ilustración 104: Propiedad disabled.....	85
Ilustración 105: Editar enlace de acción	86
Ilustración 106: Componente popup.....	87
Ilustración 107: Propiedades de dialog.....	88
Ilustración 108: Código de popup.....	88

Ilustración 109: Crear Managed Bean	89
Ilustración 110: Código de Consulta.....	90
Ilustración 111: Propiedades de popup	90
Ilustración 112: Código para cerrar popup.....	91
Ilustración 113: Editar enlace de acción	92
Ilustración 114: Código de MethodAction	92
Ilustración 115: Propiedades de botón	92
Ilustración 116: Insertar action	93
Ilustración 117: Crear enlace de acción	93
Ilustración 118: Código de yesAction y noAction.....	94
Ilustración 119: Propiedades de showPopupBehavior.....	94
Ilustración 120: Aplicación.....	95
Ilustración 121: Propiedad rendered.....	96
Ilustración 122: Crear AccessorIterator	97
Ilustración 123: Constructor de expresiones.....	98
Ilustración 124: Código de named query	99
Ilustración 125: Código de persistTemas.....	99
Ilustración 126: Código de mergeTemas.....	99
Ilustración 127: Propiedades de elemento menú.....	100
Ilustración 128: Excepción ORA-02292.....	100
Ilustración 129: Crear clase java	101
Ilustración 130: Código de CustomErrorHandler	102
Ilustración 131: Menú crear clase.....	102
Ilustración 132: Código de MyDCErrorMessage	103
Ilustración 133: Propiedades de DataBindings	103
Ilustración 134: Propiedad splitterPosition.....	104
Ilustración 135: Propiedad width	104
Ilustración 136: Aplicación.....	104
Ilustración 137: Open Application.....	106
Ilustración 138: Menú Open Application	106
Ilustración 139: Selección del fichero jws	107

Introducción

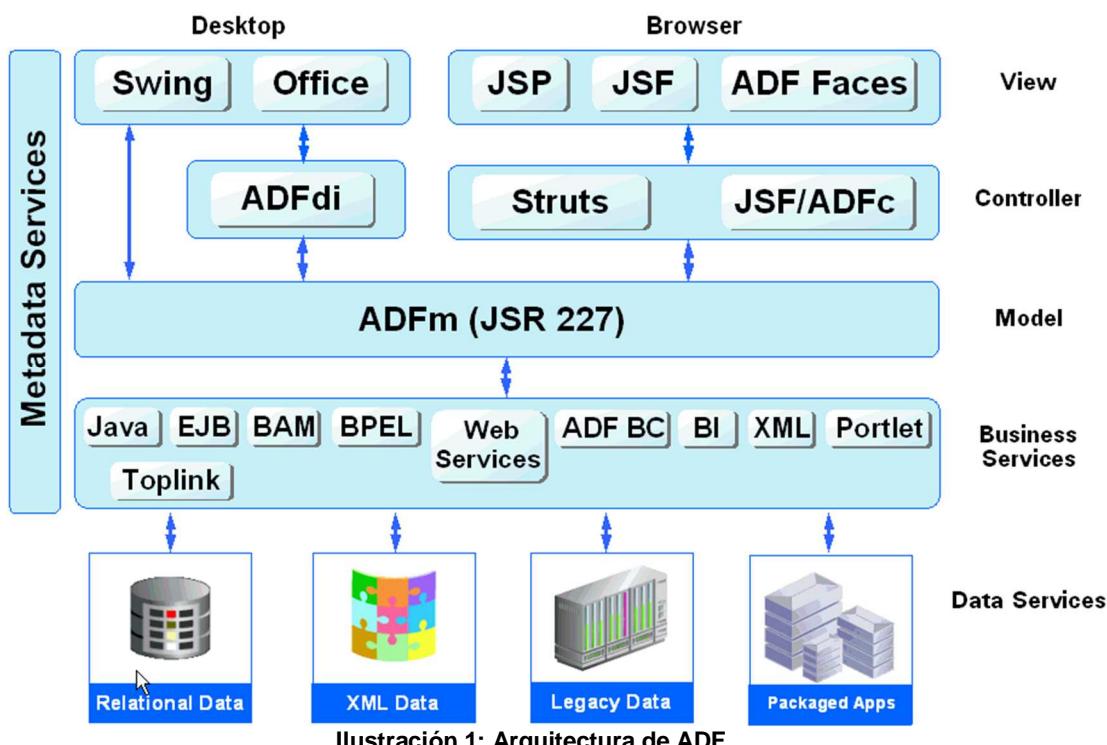
El objetivo de este tutorial es realizar una introducción a las tecnologías EJB 3.0 y Oracle ADF, mediante el desarrollo de una pequeña aplicación Web.

Oracle Application Development Framework, como su nombre indica, es el framework para desarrollo de aplicaciones de Oracle. Es el framework utilizado internamente en Oracle para sus desarrollos propios, si bien es poco conocido comercialmente. Precisamente uno de los objetivos de este tutorial es darlo a conocer, dado que ofrece algunas propuestas muy interesantes.

Oracle Application Development Framework

Oracle ADF es un framework para el desarrollo de aplicaciones J2EE, que está basado en la arquitectura Modelo-Vista-Controlador (MVC). ADF ofrece una posible solución para implementar cada una de las capas de este modelo, si bien permite igualmente el uso de casi cualquier variedad de las existentes en el mundo J2EE.

Una de las principales aportaciones de ADF está precisamente en la forma en que trabaja con la capa de Modelo. ADF separa la capa de modelo de la de servicios de negocio, con el objetivo de facilitar el desarrollo orientado a servicios. De este modo, Oracle ADF quedaría dividido en 4 capas.



Esta forma de trabajar con la capa Modelo es la base de la especificación JSR-227, y pretende conseguir la separación entre las capas del Controlador y la Vista, y la capa de servicio de negocios, ocultando su implementación.

De esta forma, el objetivo de cada una de las capas sería el siguiente.

- Capa de servicios de negocio.- contiene la lógica de la aplicación, y maneja el acceso a datos.
- Capa de modelo.- esta capa consigue la abstracción de la capa de servicios de negocio, permitiendo a la vista y el controlador trabajar con diferentes implementaciones de los servicios de negocio de un modo consistente.
- Capa controladora.- controla el flujo de la aplicación.
- Capa vista.- la interfaz de usuario de la aplicación.

Como puede observarse en la figura anterior, la capa de servicios de negocio puede ser implementada con una gran variedad de tecnologías. Además se observa que ADF no es un framework exclusivamente dedicado al mundo Web, dado que también permite el desarrollo de aplicaciones de escritorio, usando Swing para la capa vista.

A la hora de desarrollar aplicaciones Web, como se ha comentado anteriormente, el framework ADF ofrece una solución para cada una de las capas.

Para la capa de servicios de negocio, tenemos a nuestra disposición ADF Business Components (ADF BC). Esta tecnología define tres tipos de objetos para implementar la lógica de negocio de la aplicación y el acceso a datos.

- Objetos de entidad (Entity Objects).- son los objetos persistentes de la aplicación. Generalmente, por tanto, estos objetos tendrán su correspondencia con alguna tabla de la base de datos subyacente. Entre otras cosas, permiten definir reglas de validación.
- Objetos de vista (View Objects).- son los objetos a través de los que se realiza el acceso a datos. Generalmente (aunque no obligatoriamente) tienen asociados una consulta SQL que define los datos que se manejan a través del objeto.

Estos objetos tienen asociados una colección de objetos de tipo entidad, como resultado de la consulta que ejecutan.

- Módulos de aplicación (Application Modules).- encapsulan la lógica de negocio de la aplicación, sirviendo de punto de acceso a la misma. Estos objetos tienen una serie de instancias de los objetos vista, mediante los que acceden a los datos.

Esta sería, muy por encima, la propuesta de ADF para la implementación de la capa de negocio. No vamos a entrar en más detalle en su funcionamiento, puesto que en el desarrollo de este tutorial optaremos por el uso de tecnología EJB, al ser una opción más conocida y utilizada.

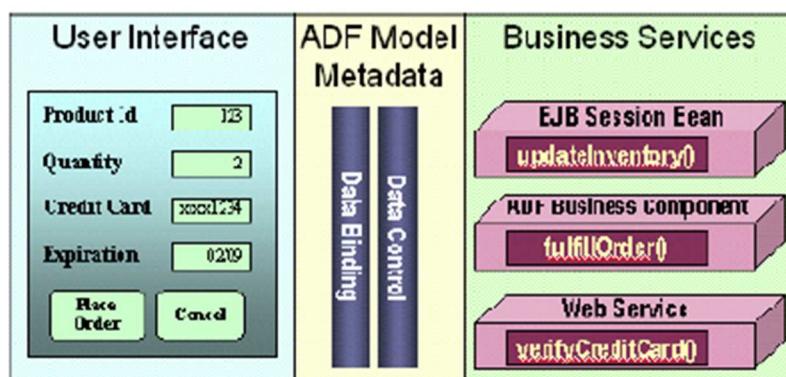
Si pasamos a la capa del controlador, ADF provee su propio controlador como alternativa a JSF o Struts. Las aportaciones de este controlador incluyen la posibilidad de incluir llamadas a métodos en el flujo de la aplicación (generalmente el flujo de las aplicaciones se reduce a cambios de páginas), y el permitir el flujo de fragmentos de páginas, entre otros.

En el desarrollo de este tutorial haremos uso del controlador de JSF.

Respecto a la capa vista, ADF provee una implementación de JSF, denominada ADF Faces. Esta implementación fue donada al proyecto Trinidad, de Apache, y será la que utilicemos para desarrollar este tutorial.

Vamos a ahondar ahora un poco más en el que es el punto central de Oracle ADF, la capa de modelo. El objetivo de esta capa es proveer una interfaz de acceso a la capa de servicios de negocio, independientemente de las tecnologías utilizada para su implementación.

Esta capa se descompone en dos subcapas, a las que haremos referencia generalmente por su nombre en inglés, data controls, y data bindings (se traducirían como control de datos y enlace de datos). El data control es el encargado de la abstracción de los servicios de negocio para el cliente. Por su parte, data bindings ofrece los métodos y atributos a los componentes de la interfaz, separando claramente la vista y el modelo. De esta forma, el desarrollo de la interfaz de la aplicación se haría siempre de la misma forma independientemente de la tecnología que se esté utilizando en la capa de servicios de negocio.



Hasta aquí la introducción teórica, que está bien como punto de partida; pero, cómo se traduce todo esto a la hora de trabajar lo veremos con detalle en el desarrollo del ejemplo de este tutorial.

EJB 3.0

Ya hemos comentado que en este tutorial trabajaremos sobre un ejemplo en el que utilizaremos EJB 3.0 como tecnología para la capa de servicios de negocio. Por tanto, daremos a continuación una pequeña introducción teórica también a esta tecnología.

EJB, o Enterprise Java Beans, es la arquitectura de componentes del lado servidor para la plataforma Java, y pretende habilitar el desarrollo rápido y simplificado de aplicaciones distribuidas, transaccionales y portables. Esto es, el objetivo de esta tecnología es liberar al programador de hacerse cargo de tareas como la concurrencia, el control de transacciones, la persistencia de objetos, la seguridad, etc., y centrarse en la lógica de negocio de la aplicación.

Como casi todo en el mundo Java, EJB 3.0 es una especificación, y hace uso de otra especificación, JPA o Java Persistence API, para gestionar la persistencia y el mapeo objeto – relacional.

A la hora de trabajar con EJB, tendremos principalmente dos tipos de objetos.

- Objetos de entidad.- son los objetos persistentes de las aplicaciones. Esto es, un entity bean será un objeto java que contendrá, en sus atributos, los valores que deben ser almacenados en la base de datos. Generalmente un

objeto entidad se corresponde directamente con una tabla de la base de datos, y contiene atributos, al menos, para cada una de las columnas de dicha tabla.

- EJB de sesión.- sirven como fachada para el acceso a los servicios proporcionados, encapsulando los servicios de negocio. Estos EJB definen una interfaz de servicio, que es usada por los clientes para interactuar con los beans. La implementación de dicha interfaz se realiza en una clase java simple, pero que tiene acceso a los servicios ofrecidos por el contenedor de EJBs.

Existen EJB de sesión con estado (stateful), y sin estado (stateless). La interacción de un cliente con un EJB stateless comienza cuando se efectúa la llamada a uno de sus métodos, y termina con el método. Por tanto, no hay posibilidad de almacenar información de estado entre llamadas. Por el contrario, al trabajar con un EJB stateful, el cliente obtiene una referencia al mismo, y puede trabajar con él hasta que lo libere explícitamente, de forma que es posible almacenar información de estado entre llamadas.

Entorno de desarrollo

El entorno de desarrollo más adecuado para trabajar con Oracle ADF es, sin duda, Oracle jDeveloper. Este IDE ofrece las posibilidades habituales de este tipo de herramientas como completadores de código, capacidades para refactorizar, editores visuales, etc. Pero además, al ser el entorno de Oracle, está particularmente preparado para trabajar con su framework de desarrollo de una forma cómoda y eficiente.

El ejemplo vamos a desarrollarlo utilizando la última versión disponible en la fecha de este tutorial, que es la 11.1.1.3.0, y puede descargarse desde <http://www.oracle.com/technetwork/developer-tools/jdev/downloads/index.html>.

En cuanto a la base de datos, usaremos también el sistema de gestión de Oracle 10g, en su versión Express 10.2.0.

Para terminar de cerrar el círculo, usaremos igualmente el servidor de aplicación de Oracle, WebLogicServer 10.3.3.0, que se distribuye junto con la versión comentada de jDeveloper.

Definición del problema

Para ver cómo funciona todo esto, y entrar en mayor detalle en algunos aspectos, vamos a plantear un problema y a desarrollar una simple aplicación Web.

Esta aplicación permitirá mantener un listado de tareas con su estado, e ir anexando a las mismas anotaciones sobre su desarrollo.

Desarrollo

Organización del trabajo en jDeveloper

jDeveloper trabaja siempre con lo que denomina *Módulos de Aplicación*. Este elemento, básicamente, sirve como contenedor de más alto nivel para nuestra aplicación, y se compone de un número variable de proyectos.

Por tanto, el siguiente nivel de organización dentro de jDeveloper sería el *Proyecto*. Un proyecto podrá contener ya directamente elementos de trabajo como clases java, páginas Web, ficheros de configuración, etc. Como es normal en el mundo java, esta información se organiza siempre en paquetes.

Por tanto, para comenzar a trabajar, lo primero que tenemos que hacer es crear un módulo de aplicación. Para ellos accedemos a la opción de menú *File – New*. Entre las categorías ofrecidas, nos vamos a *General – Applications*, y seleccionamos la opción *Generic Application*. Más adelante entraremos en más detalle en la pantalla de creación de nuevos elementos y las opciones que ofrece.

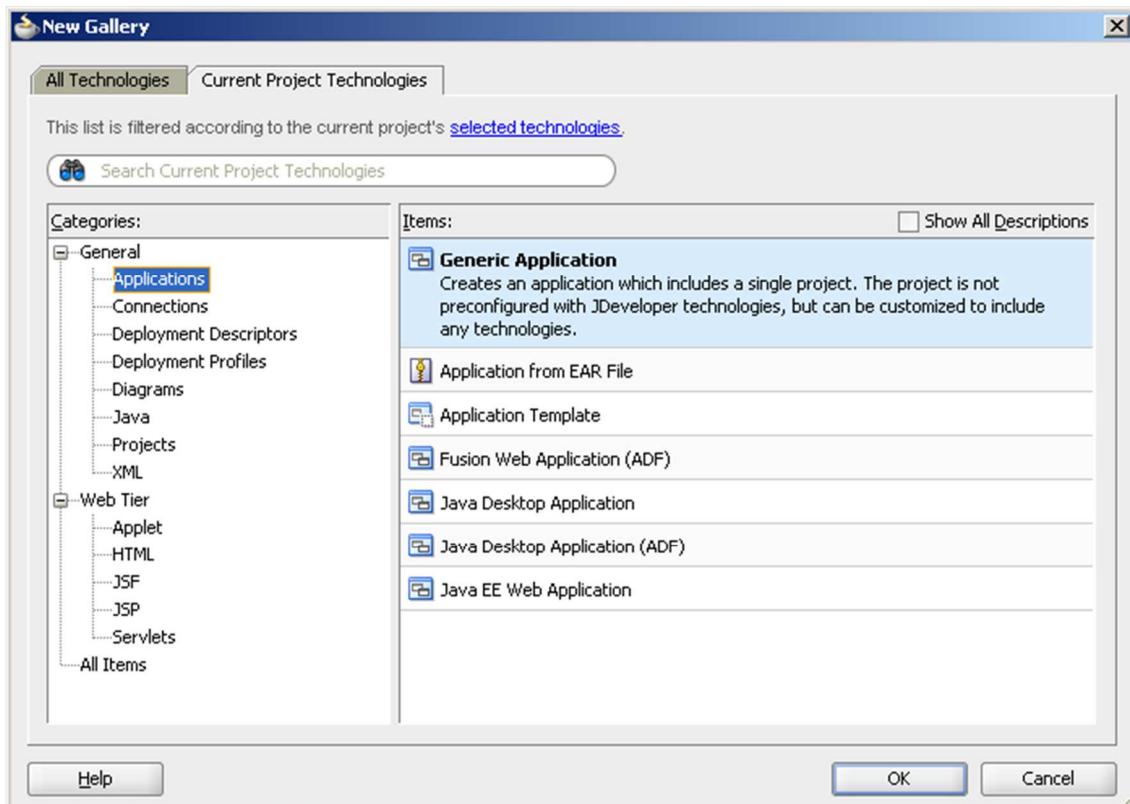


Ilustración 3: Nueva aplicación genérica

Como alternativa, podríamos haber elegido un tipo concreto de aplicación; en este caso, jDeveloper se encargará de forma automática de crear la estructura base de la aplicación, con los proyectos y ficheros de configuración necesarios. Para el desarrollo de este tutorial elegimos la opción genérica con el objetivo de ir construyendo paso a paso los proyectos, y ver qué opciones se nos ofrecen. Si estos pasos son conocidos, se puede optar directamente por la opción *Java EE Web Application*.

A continuación llegamos al diálogo de creación de aplicaciones. En este punto tenemos que elegir un nombre para nuestra aplicación, así como la ubicación en la que queremos que se almacene dentro de nuestro equipo. También se nos ofrece la posibilidad de definir un paquete base, a partir del cual se almacenarán todos los elementos que vayamos creando en la aplicación.

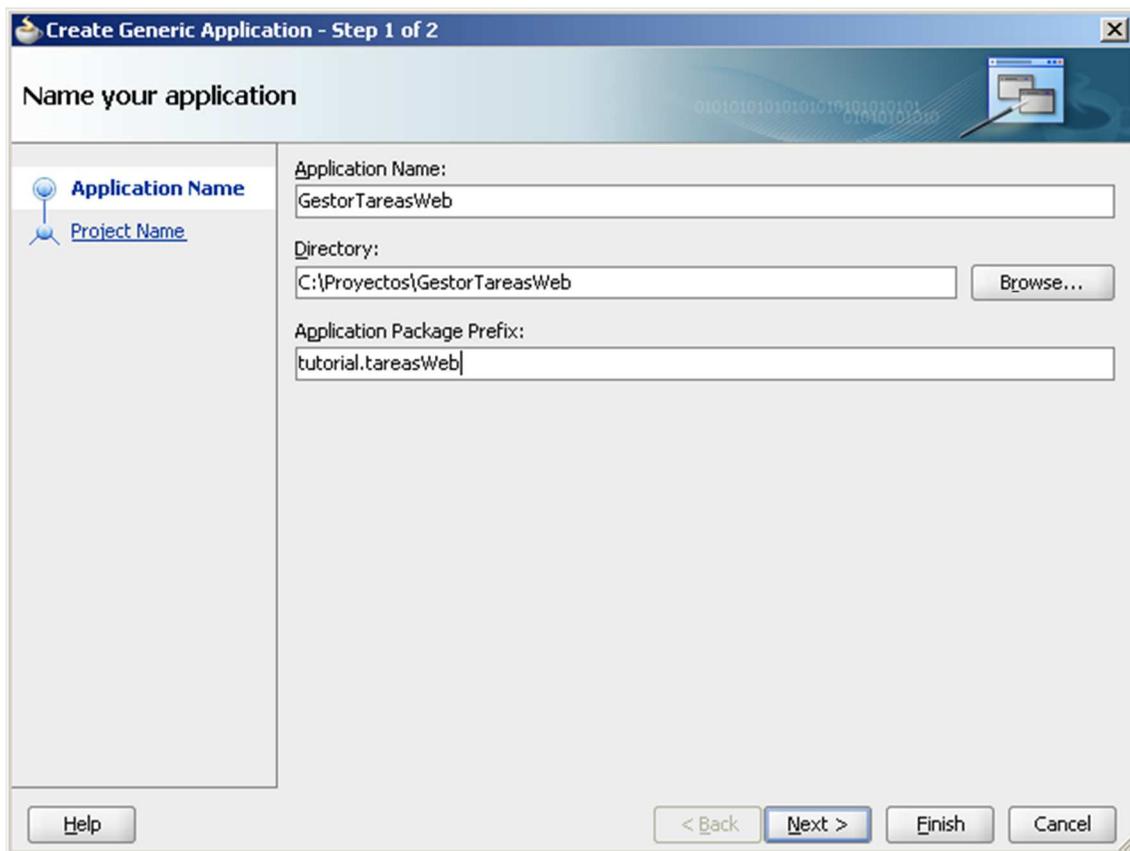


Ilustración 4: Crear una nueva aplicación

En el siguiente paso jDeveloper nos pide que creamos un primer proyecto para la aplicación. Además, tenemos que indicar las tecnologías que estarán disponibles en el mismo. Generalmente las aplicaciones J2EE se separan en dos proyectos, uno para la capa modelo, y otro para las capas vista y controlador. Crearemos aquí el de la capa modelo, que utilizará tecnología EJB.

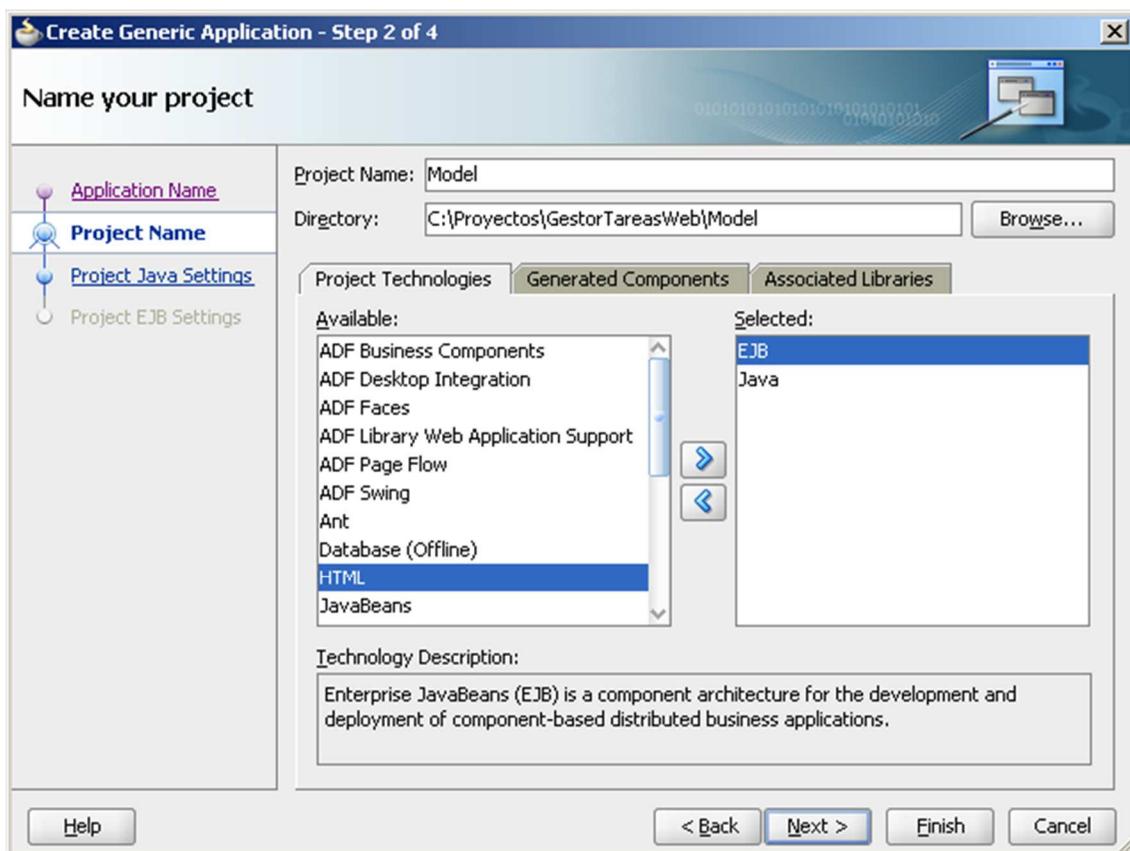


Ilustración 5: Crear proyecto

En el tercer paso nos volverá a pedir el paquete por defecto, además de las rutas para los fuentes y clases, que dejaremos en sus valores por defecto. Una vez en el cuarto paso, tendremos que indicar la versión de EJB que vamos a usar, 3.0, y que utilizaremos anotaciones.

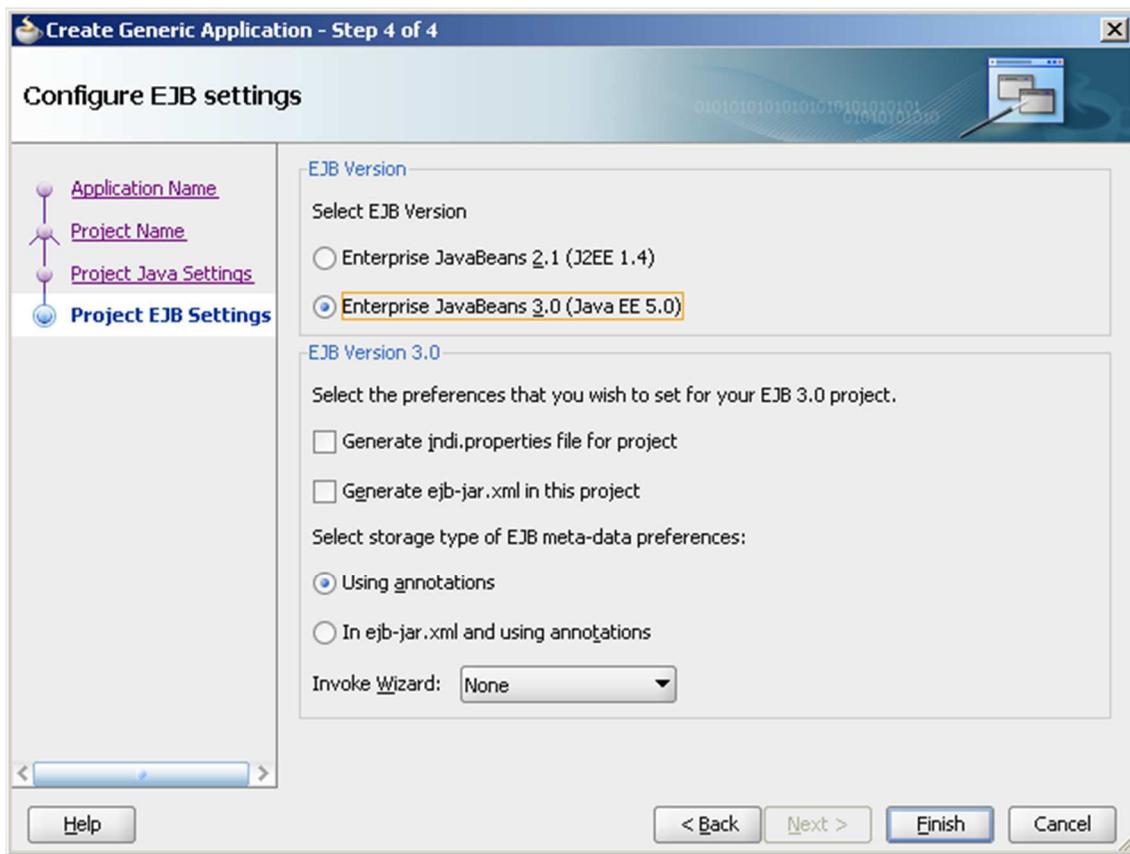


Ilustración 6: Configurar EJBs

En este punto tenemos nuestro módulo de aplicación y proyecto creados. Si somos curiosos, podemos echar un vistazo a la carpeta que hemos seleccionado como destino de nuestra aplicación, para ver qué ha creado el IDE. Hasta el momento sólo podemos ver un fichero, de nombre *GestorTareasWeb.jws*. La extensión *jws* corresponde a *Java WorkSpace*, y este fichero no es más que la definición del espacio de trabajo de nuestra aplicación. Podemos abrirlo con un editor de texto, ya que no es más que un documento *xml*.

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<jws:workspace xmlns:jws="http://xmlns.oracle.com/ide/project">
  <value n="application-package-prefix" v="tutorial.tareasWeb"/>
  <value n="appTemplateId" v="#genericApplicationTemplate"/>
  <hash n="component-versions">
    <value n="oracle.adf.share.dt.migration.jps.JaznCredStoreMigratorHelper" v="11.1.1.1.0"/>
    <value n="oracle.adfdt.controller.adfc.source.migration.SavePointDataSourceForWLSMigrator" v="11.1.1.1.0"/>
    <value n="oracle.adfdtinternal.model.ide.security.wizard.AdfSecurityMigrator" v="11.1.1.1.0.13"/>
    <value n="oracle.ide.model.Project" v="11.1.1.1.0"/>
    <value n="oracle.jdevimpl.deploy.ear.OarMigratorHelper" v="11.1.1.1.0"/>
    <value n="oracle.jdevimpl.deploy.mar.MarMigratorHelper" v="11.1.1.1.0"/>
    <value n="oracle.jdevimpl.xml.oc4j.jps.JpsConfigMigratorHelper" v="11.1.1.1.0.1"/>
    <value n="oracle.jdevimpl.xml.wl.WeblogicMigratorHelper" v="11.1.1.1.0"/>
    <value n="oracle.mds.internal.dt.ide.migrator.MDSConfigMigratorHelper" v="11.1.1.0.5313"/>
  </hash>
  <list n="listOfChildren">
    <hash><url n="URL" path="Model/Model.jpr"/></hash>
  </list>
</jws:workspace>
```

Ilustración 7: Fichero jws

Como podemos ver, en este fichero se especifican básicamente las versiones de los componentes con los que vamos a trabajar, y el paquete base que hemos elegido, además de la lista de proyectos que forman parte de la aplicación.

jDeveloper ha creado también una carpeta con el nombre de nuestro proyecto, Model, que contiene en su raíz un fichero *Model.jpr*. Al igual que teníamos el fichero *jws* para definir el módulo de aplicación, tenemos un fichero *jpr* para definir el proyecto. Si lo abrimos podremos ver que en él se incluye información sobre la tecnología y versiones del proyecto, las rutas en que se almacenan los distintos tipos de fichero, las librerías a las que se tiene acceso, la ruta a la que se enviarán los ficheros compilados, etc. No vamos a entrar en más detalle, puesto que este fichero es gestionado directamente por el IDE.

Creación de la base de datos

Vamos a pasar ahora a diseñar la base de datos con la que trabajará nuestra aplicación. jDeveloper ofrece herramientas que permiten llevar a cabo esta tarea, así que vamos a usarlas. Para crear nuevos elementos en nuestro proyecto, podemos hacer clic con el botón derecho sobre él, y acceder directamente a la opción New. Vamos a ver ahora con mayor detalle este cuadro de diálogo.

Como se puede observar, este diálogo está separado en dos pestañas. En la primera de ellas, *All Technologies*, tenemos disponibles todos los elementos con los que se puede trabajar en el entorno jDeveloper. Sin embargo, por defecto se nos presenta la segunda pestaña, *Current Project Technologies*. La diferencia está en que en esta pestaña sólo tenemos disponibles aquellos elementos relacionados con la tecnología que seleccionamos para nuestro proyecto a la hora de crearlo. Esto simplifica nuestro trabajo eliminando elementos que normalmente no tendremos que utilizar, como podría ser por ejemplo un interfaz Swing, o un Servicio Web. En cualquier caso, si nos interesase crear uno de estos elementos, no tenemos más que acceder a la pestaña correspondiente, y podremos seleccionarlo.

De hecho, en nuestro siguiente paso vamos a necesitar acceder a la pestaña de todas las tecnologías, puesto que queremos diseñar nuestra base de datos, y esa opción no pertenece al conjunto que jDeveloper ofrece para un proyecto EJB. Si alguien lo prefiere, para mayor claridad, podría crear un proyecto independiente para todo lo relacionado con la base de datos.

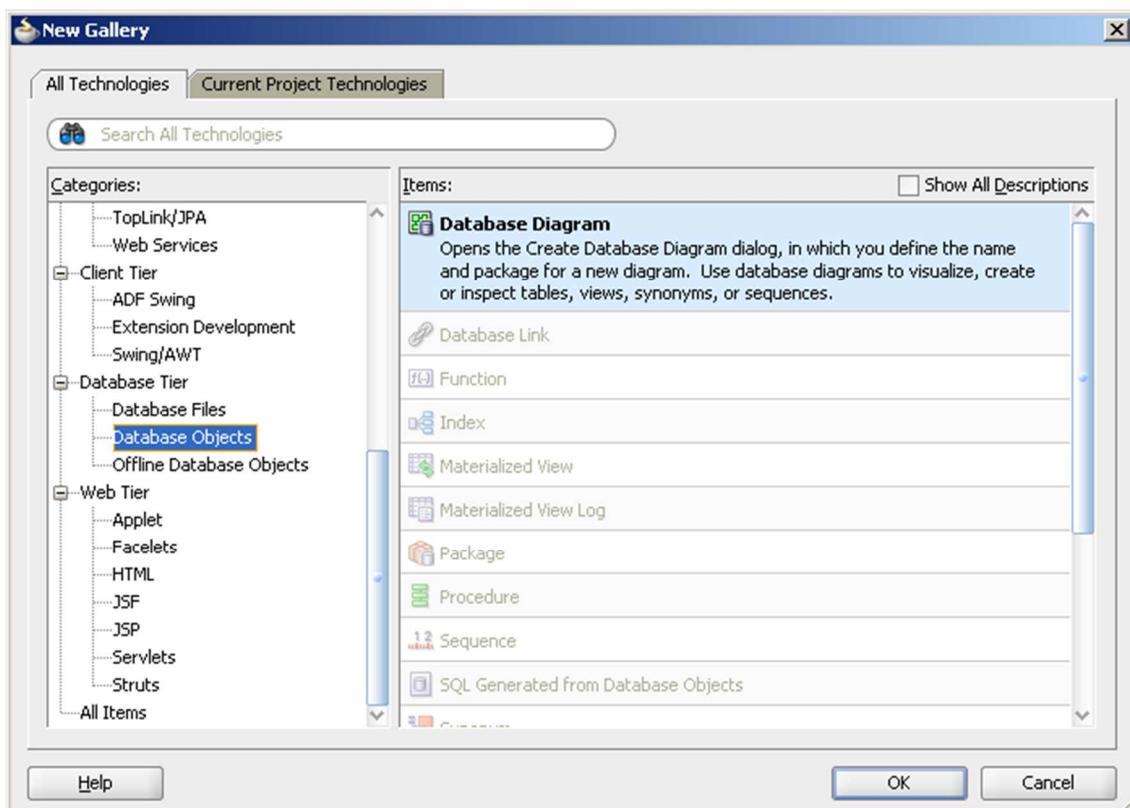


Ilustración 8: Nuevo diagrama de base de datos

Seleccionamos la categoría *Database Tier – Offline Database Objects*, y dentro de ella elegimos *Database Diagram*. Nos aparece un diálogo para darle un nombre (por ejemplo, *TareasDiagram*), y una ubicación, y automáticamente se abre para que podamos comenzar a trabajar.

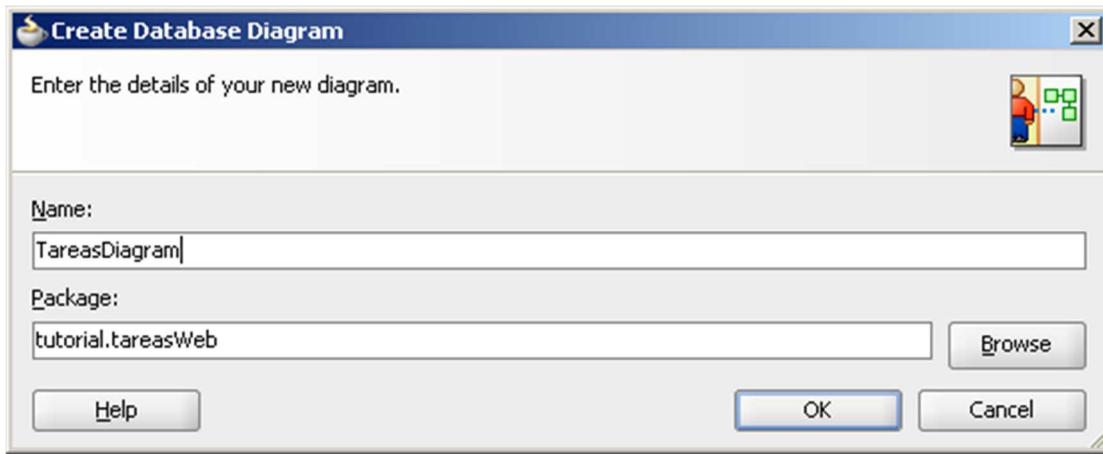


Ilustración 9: Crear diagrama de base de datos

La forma de trabajar con este tipo de diagramas es muy simple. Basta con arrastrar desde la paleta el tipo deseado, y posteriormente hacer doble clic sobre el ítem creado para editar sus propiedades. La primera vez que arrastramos un elemento se nos pedirá que introduzcamos un nombre para la base de datos desconectada (también tenemos la opción de trabajar directamente con una conexión a base de datos, pero vamos a hacerlo por partes). Llamamos a la base de datos *Tareas*.

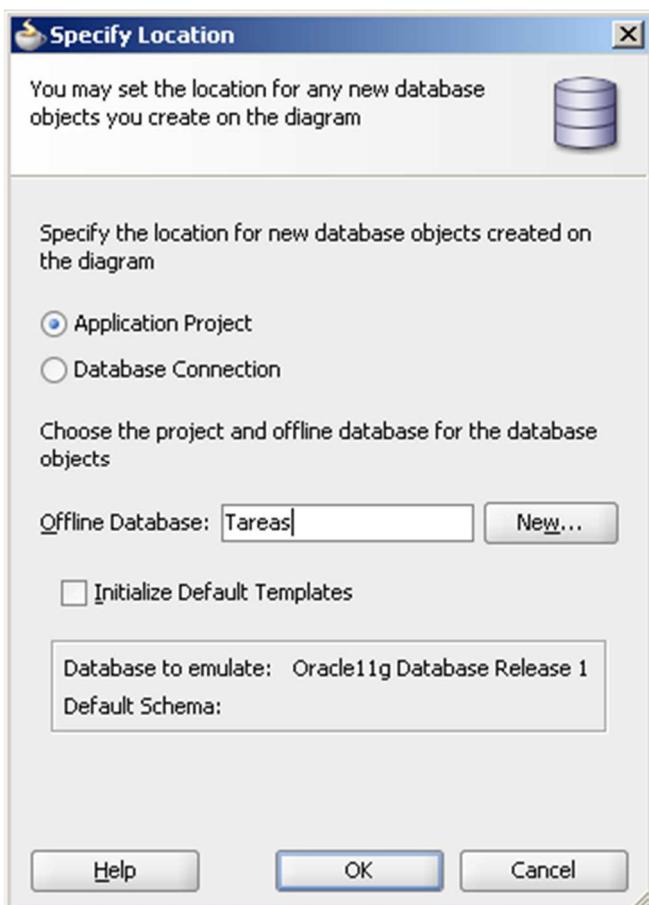


Ilustración 10: Especificar localización

Una vez situada la primera tabla, accedemos a las propiedades para indicar las columnas, así como su clave primaria. Cambiamos también el esquema, ya que por defecto jDeveloper da el valor *schema1*.

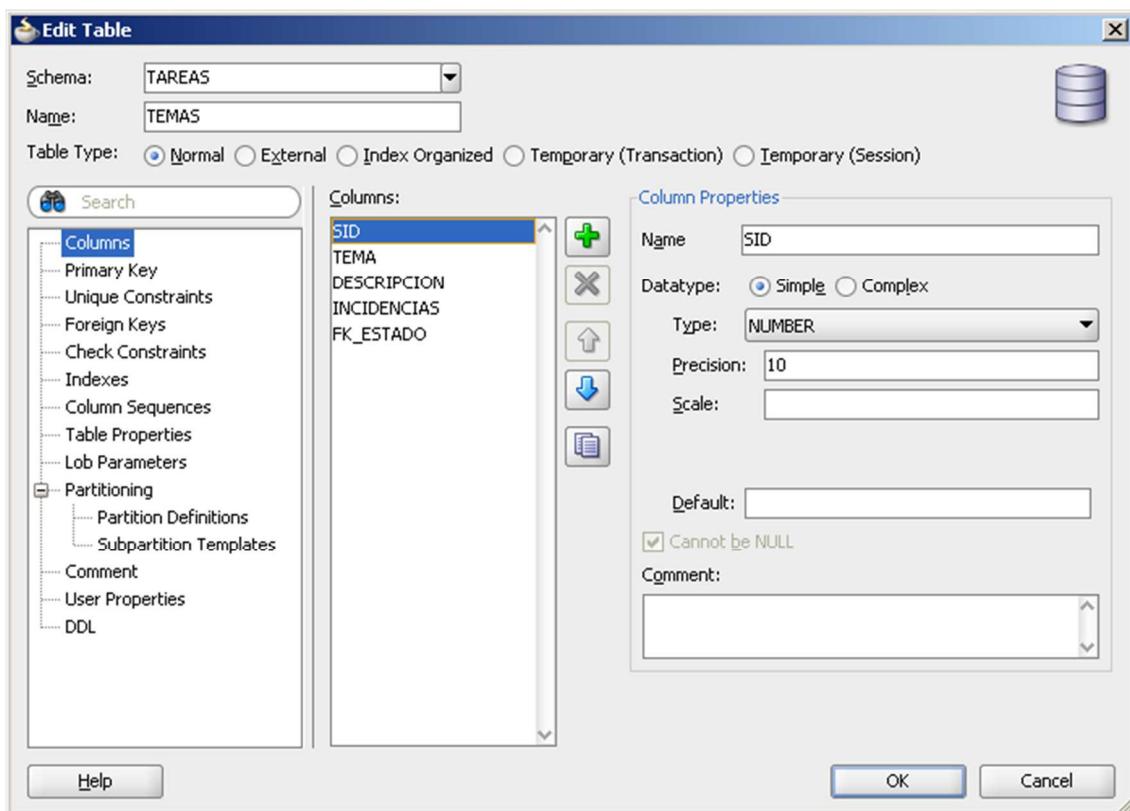


Ilustración 11: Editar tablas – Columnas

Mediante el listado situado en la izquierda podemos ir accediendo a los diferentes apartados que tenemos que configurar de la tabla. En la sección *Primary Key* seleccionamos la columna *SID* como clave primaria. El IDE le asignará un nombre automáticamente.

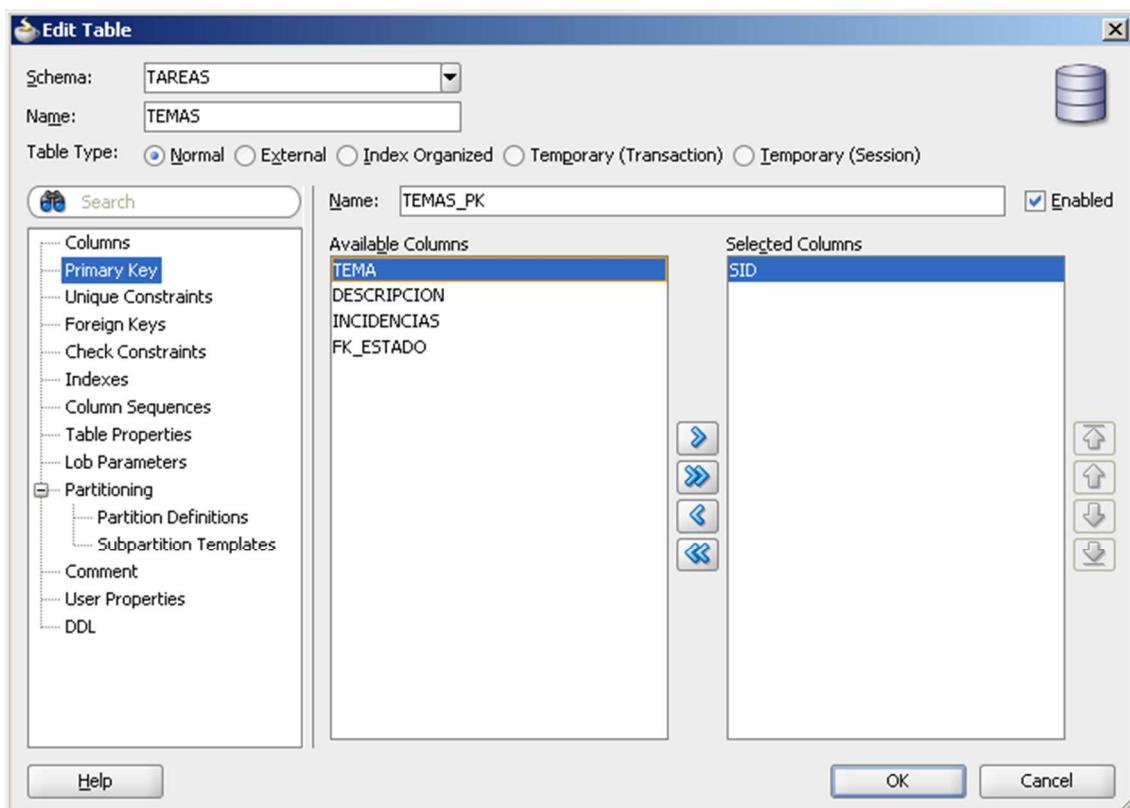


Ilustración 12: Editar tablas – Clave primaria

Vamos a crear tres tablas de acuerdo a la siguiente definición. En todos los casos la clave primaria la compone el campo *SID*.

COLUMNA	TIPO
SID	NUMBER(10)
TEMA	VARCHAR2(255)
DESCRIPCION	VARCHAR2(255)
INCIDENCIAS	VARCHAR2(255)
FK_ESTADO	NUMBER(10)

COLUMNA	TIPO
SID	NUMBER(10)
NOTA	VARCHAR2(255)
FECHA	DATE
FK_TEMA	NUMBER(10)

COLUMNA	TIPO
SID	NUMBER(10)
ESTADO	VARCHAR2(255)

Una vez definidas las tablas, añadimos las claves ajenas que las relacionan. Para ello seleccionamos el componente correspondiente de la paleta (*Foreign Key*), y pulsamos sobre las dos tablas que se van a relacionar (o bien accedemos a la opción Foreign Keys, dentro de la edición de tablas). Después sólo tenemos que indicar qué columnas van a ser las que se relacionen. Así obtenemos el modelo de datos definitivo de nuestro ejemplo.

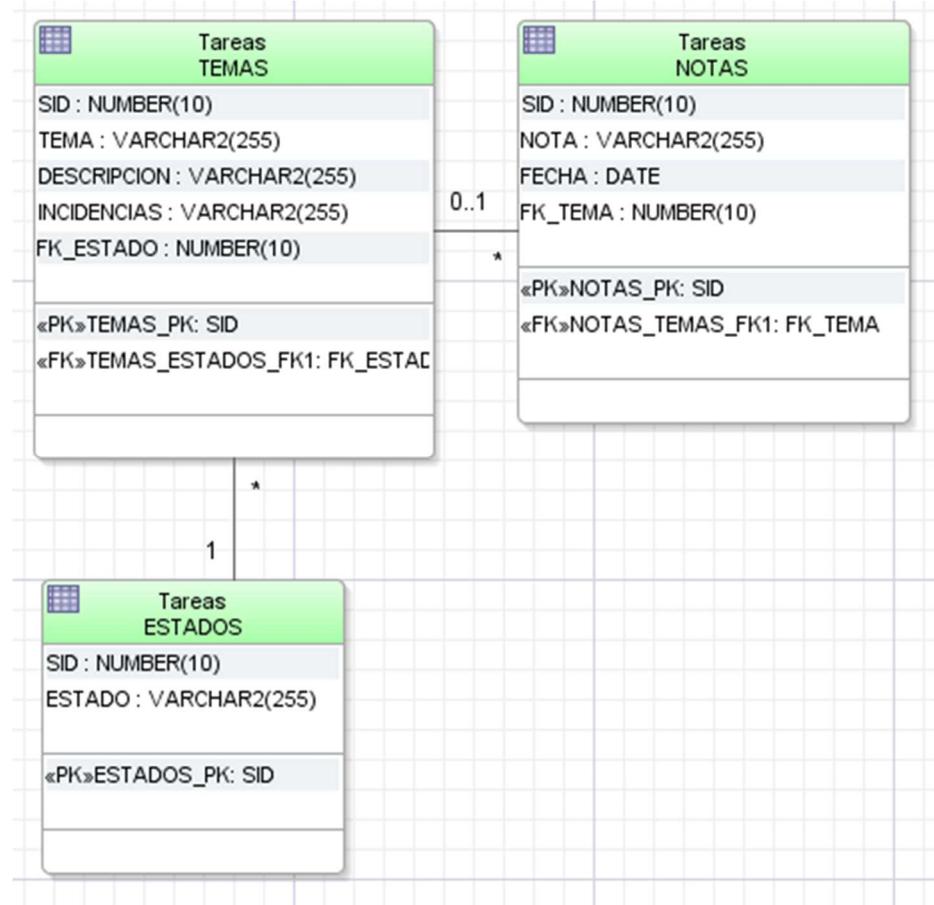


Ilustración 13: Esquema de datos

Ya tenemos nuestro modelo de datos, así que vamos a crear la base de datos. Para ello nos conectamos con el usuario system, y creamos nuestro esquema con el siguiente script. Podemos usar alguna herramienta externa, o el propio jDeveloper.

```

create user tareas identified by tareas;
create tablespace tareas datafile 'c:\oraclexe\oradata\tareas.dbf' size 5m;
alter user tareas default tablespace tareas;
grant connect, resource to tareas;

```

Ilustración 14: Script de creación de base de datos

Ahora podemos configurar la conexión a la base de datos en el entorno de desarrollo. Para ello accedemos a la pestaña *Database Navigator* (si no la encontramos, la mostramos accediendo al menú *View – Database – Database Navigator*). Encontraremos un grupo por cada una de las aplicaciones que hayamos creado. En el grupo de conexiones de nuestra aplicación, *GestorTareasWeb*, pulsamos con el botón derecho y seleccionamos la opción *New Connection*. Rellenamos los datos de conexión, y pulsamos el botón *Test* para comprobar que todo es correcto.

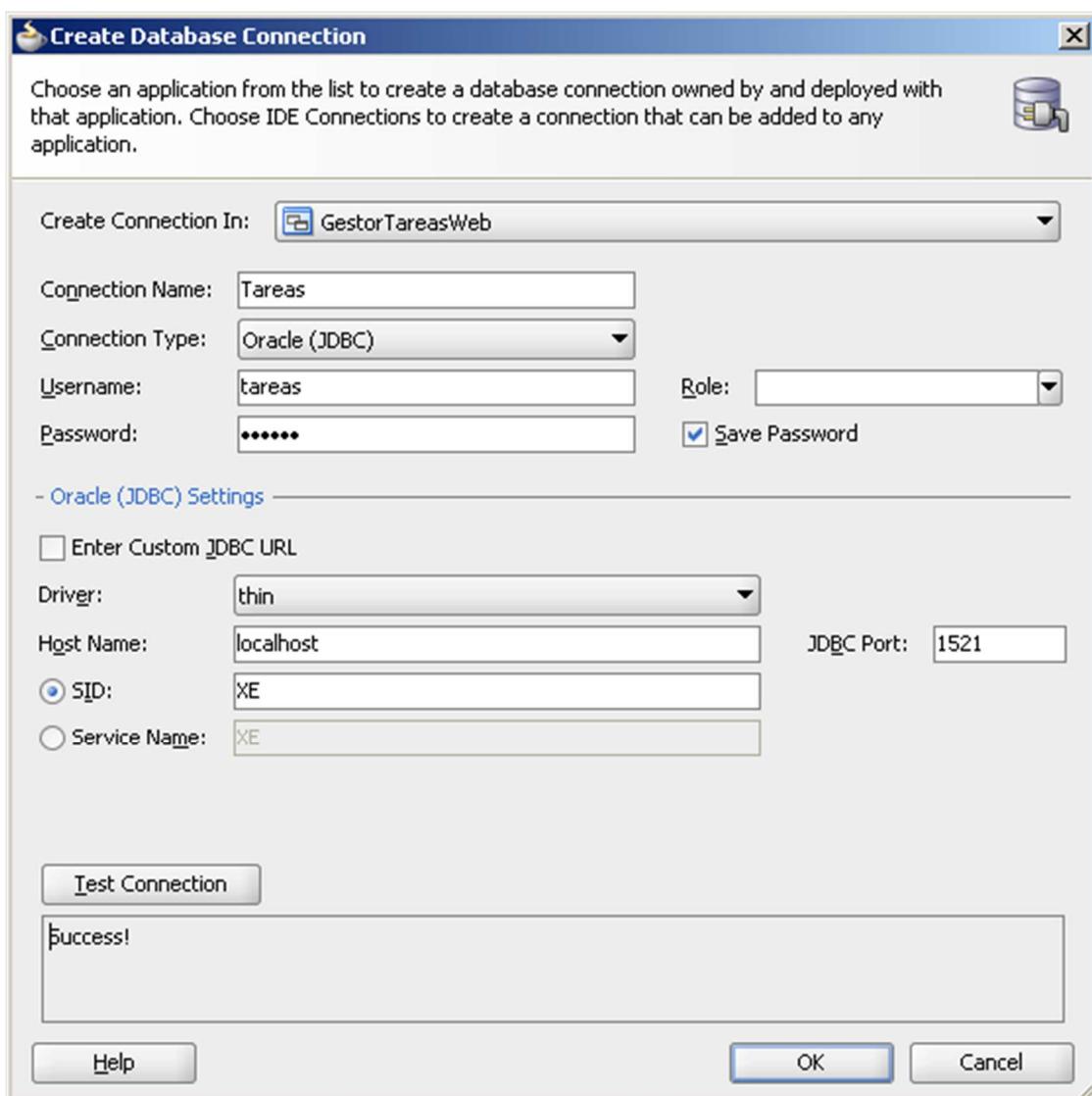


Ilustración 15: Crear conexión a la base de datos

Ahora podemos decirle a jDeveloper que se encargue de generar las tablas que definimos anteriormente sobre este esquema. Para ello, volvemos al diagrama de base de datos, y con el botón derecho sobre el mismo seleccionamos la opción *Synchronize with database – Generate To... - Tareas*. Si estamos trabajando con una base de datos Oracle 10, se mostrará un mensaje indicando que las funcionalidades de Oracle 11 serán ignorada. Aceptamos, y se muestra un diálogo que nos permite definir qué hará el IDE.

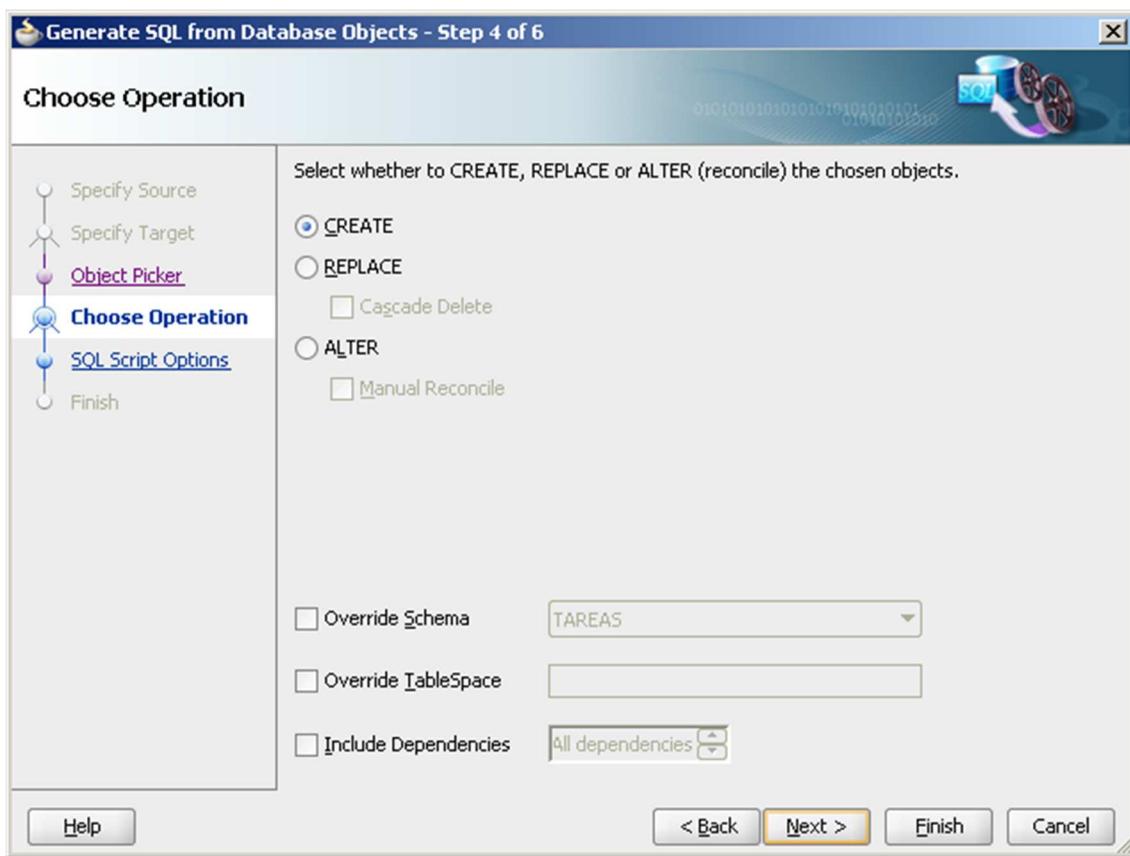


Ilustración 16: Generar SQL a partir de objetos de la base de datos

En el primer paso simplemente indicamos que queremos crear los objetos. La siguiente pantalla nos permite crear, si queremos, un fichero con los scripts que se van a ejecutar sobre la base de datos. Le indicamos que lo cree dentro de nuestro proyecto.

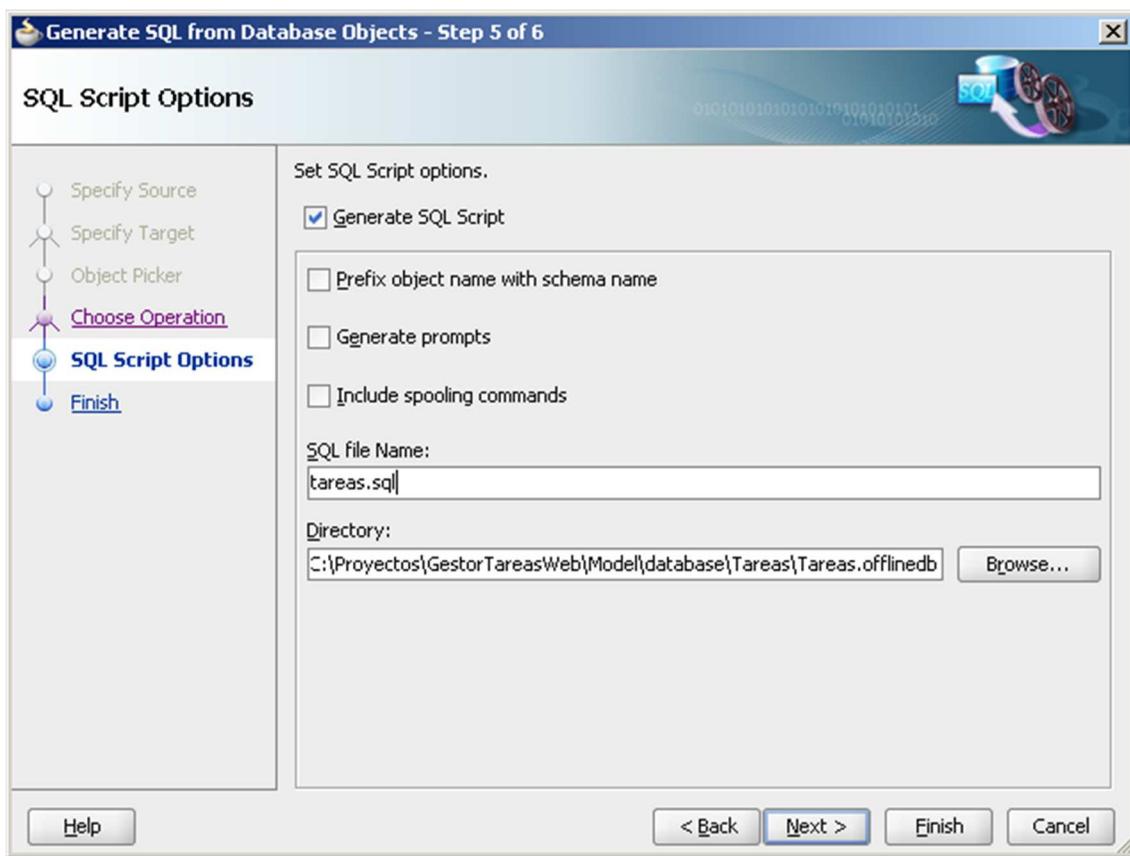


Ilustración 17: Generar SQL a partir de objetos de base de datos 2

Para terminar, se muestra un resumen de las opciones elegidas. Pulsamos en finalizar.

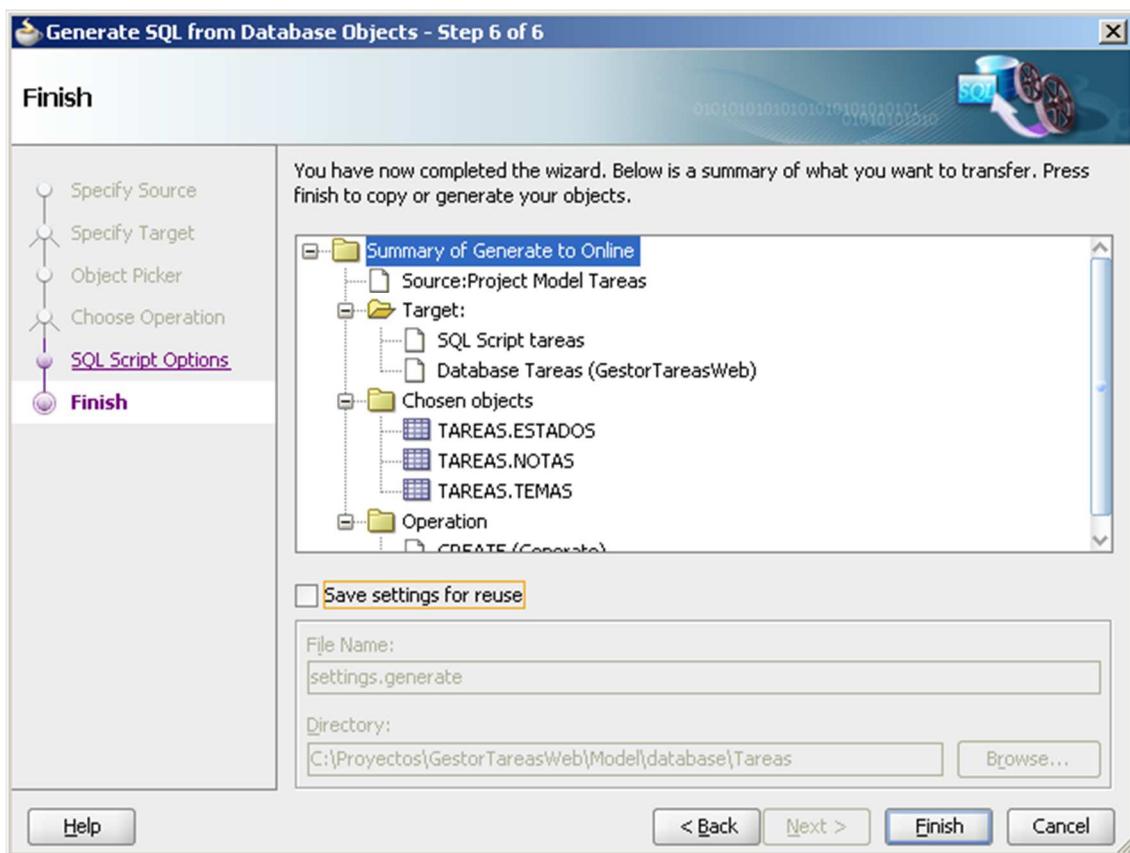


Ilustración 18: Generar SQL a partir de objetos de base de datos 3

Al terminar el proceso, el IDE abrirá automáticamente el script, en caso de haberle pedido que lo genere.

Para terminar con la base de datos, vamos a crear una secuencia para cada una de las tablas, de las que se tomarán los valores para las claves primarias. Podríamos haberlas añadido en el diagrama de base de datos, y jDeveloper las habría creado junto con las tablas, pero en este caso las crearemos nosotros mismos.

Para ello nos vamos de nuevo a la pestaña de conexiones a bases de datos (*Database Navigator*), y desplegamos la que creamos anteriormente, *Tareas*. Podemos ver en la sección de tablas que ya tenemos las nuestras.

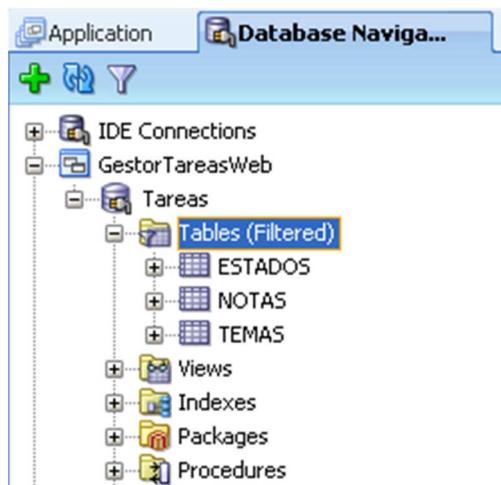


Ilustración 19: Navegador de base de datos

Para crear las secuencias hacemos clic con el botón derecho sobre *Sequences* en la vista anterior, y elegimos la opción *New Sequence*. Se abre el cuadro de diálogo para crear la secuencia. Rellenamos los datos y repetimos la operación hasta tener una secuencia para cada tabla de nuestro esquema (*seq_temas*, *seq_estados*, *seq_notas*).

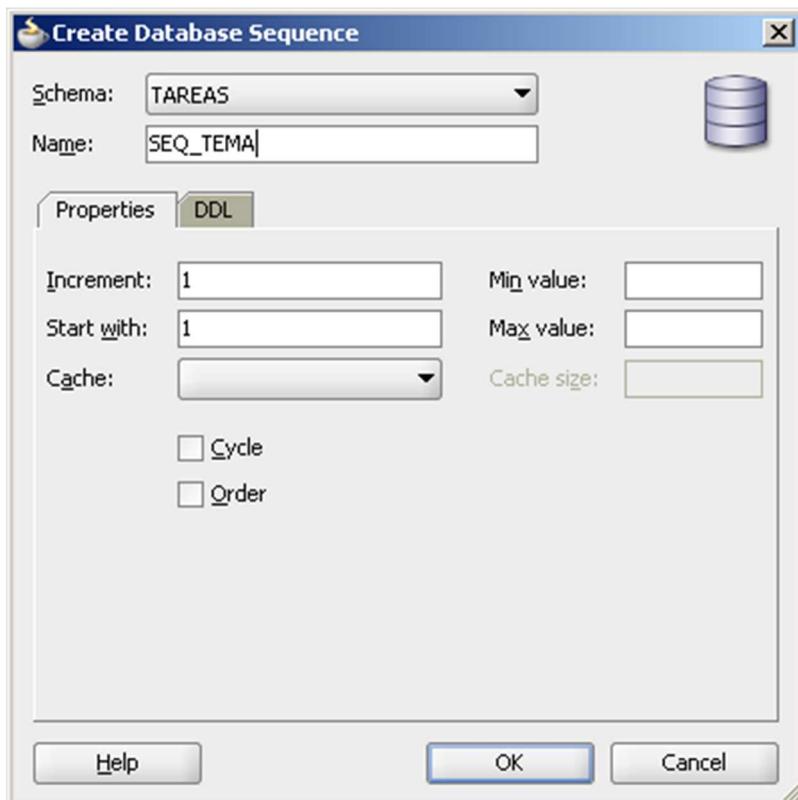


Ilustración 20: Crear secuencia

Con esto tenemos terminada nuestra base de datos. Podemos retomar la construcción de la aplicación.

El proyecto Model

Vamos a centrarnos ahora precisamente en el desarrollo del proyecto model, haciendo antes un pequeño paréntesis para explicar algunos conceptos interesantes.

Hablamos anteriormente de lo que son los *entity objects*, u objetos de entidad, y decíamos que estos objetos son los que se persistirán en nuestra base de datos. Pues bien, todas las tareas relacionadas con la persistencia de estos objetos son llevadas a cabo por un *Entity Manager*, o gestor de entidades. Un *entity manager* no es, por tanto, más que un objeto que implementa la interfaz *EntityManager*, que provee métodos para realizar las tareas relacionadas con la persistencia, y para el que cada proveedor puede dar su propia implementación.

Cada vez que un *entity manager* trabaja con una entidad, ésta pasa a ser una entidad manejada (*managed*) por el gestor. Así, el *entity manager* gestiona una serie de objetos persistentes, que es conocida como el contexto de persistencia (*persistence context*).

Los *entity managers* son configurados para gestionar un conjunto específico de objetos entidad, y trabajar contra una base de datos concreta. Esta información de

configuración es la que se define a través de la unidad de persistencia (*persistence unit*), que no es más que un fichero xml. En él se indica también cuál es el proveedor del que se tomará la implementación de *EntityManager*.

Para cerrar todo no nos queda más que decir que todos los *entity managers* son creados a partir de factorías, del tipo *EntityManagerFactory*. Estas factorías toman de la unidad de persistencia la información necesaria para saber cómo deben crear los objetos.

En la siguiente imagen se pueden ver claramente las relaciones entre estos objetos.

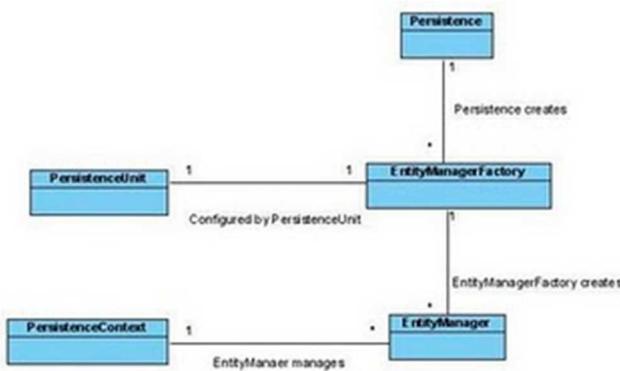


Ilustración 21: Esquema Entity Manager

Creando las entidades

Una vez revisados estos conceptos, vamos a proceder con el proyecto Model. Como trabajaremos con EJBs, crearemos las entidades correspondientes a nuestras tablas de la base de datos.

Hacemos clic con el botón derecho sobre nuestro proyecto Model, y accedemos al diálogo para crear nuevos elementos. Entre nuestras tecnologías, accedemos a las de la capa de servicios de negocio, y tanto en la categoría *EJB* como en *Toplink/JPA*, tenemos la opción para crear las entidades directamente a partir de las tablas de la base de datos (*Entities from tables*). La seleccionamos.

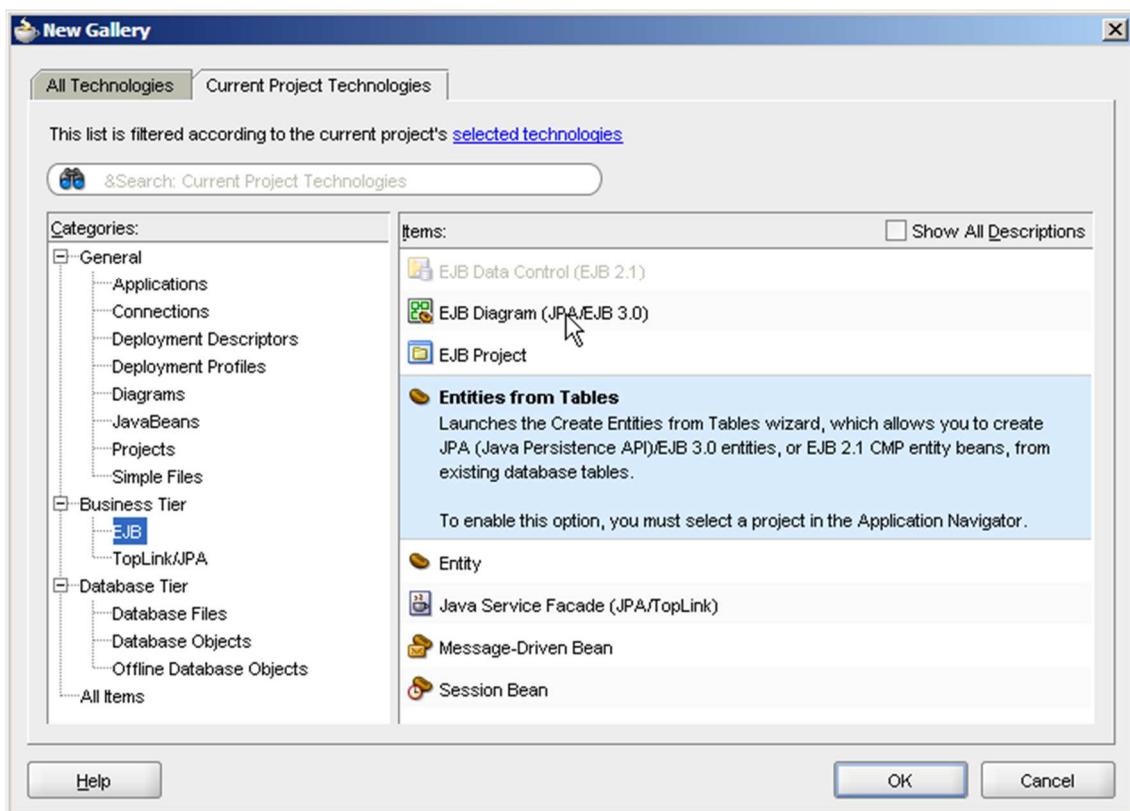


Ilustración 22: Galería de nuevos elementos 2

Como siempre, entramos en un wizard que nos facilitará mucho la tarea. Lo primero que nos pide es que indiquemos la unidad de persistencia para las entidades que se van a crear. Por defecto nos propone crear una con el nombre del proyecto. Pulsamos el botón *New...*, para indicar los datos de la unidad de persistencia.

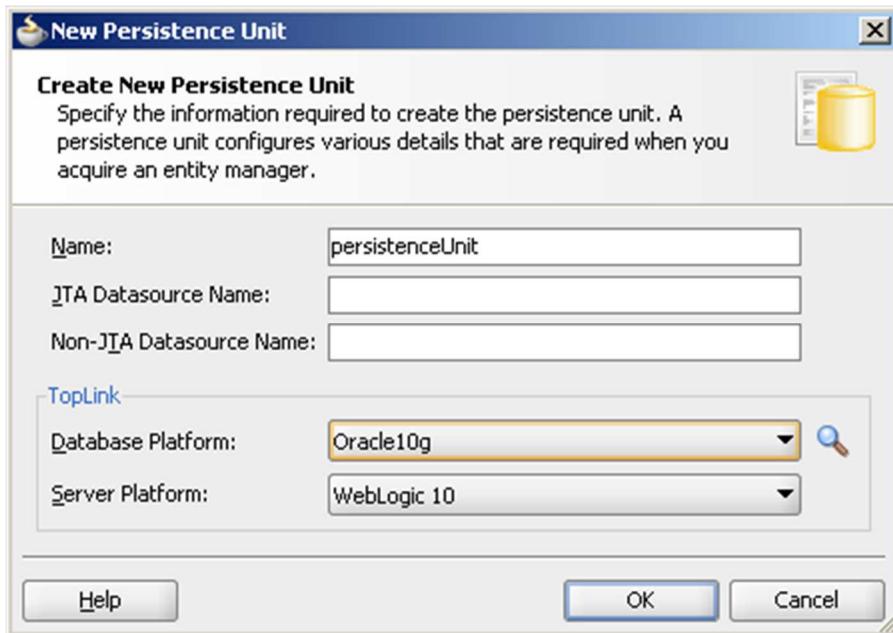


Ilustración 23: Nueva unidad de persistencia

Damos un nombre que servirá para identificarla, y será el valor que se dará a la factoría de *entity managers* para decirle cómo crearlos. Además seleccionamos las plataformas de base de datos y servidor que vamos a utilizar.

En el siguiente paso se nos pregunta el tipo de conexión de la que sacaremos nuestras entidades, que es una conexión en línea a base de datos.

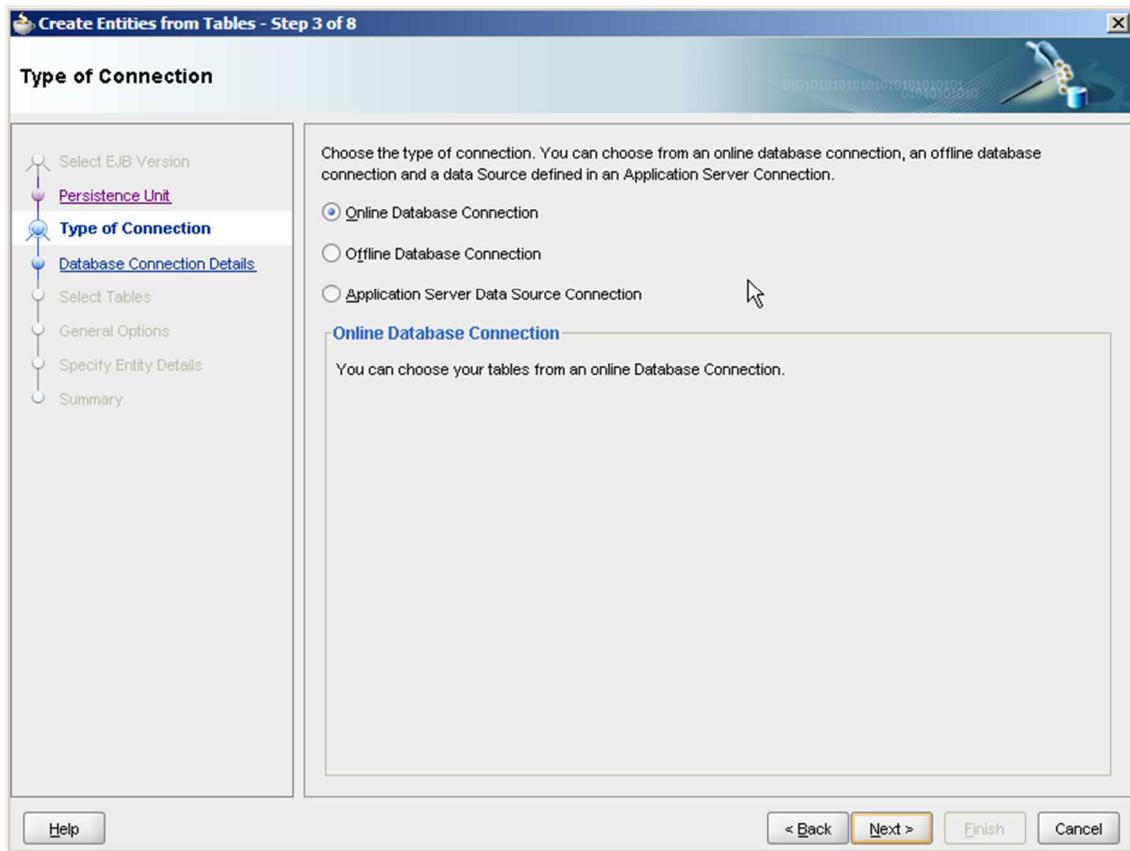


Ilustración 24: Crear entidades a partir de tablas

Como era de esperar, lo siguiente es indicar qué conexión es la que usaremos. Volvemos a seleccionar la nuestra, *Tareas*.

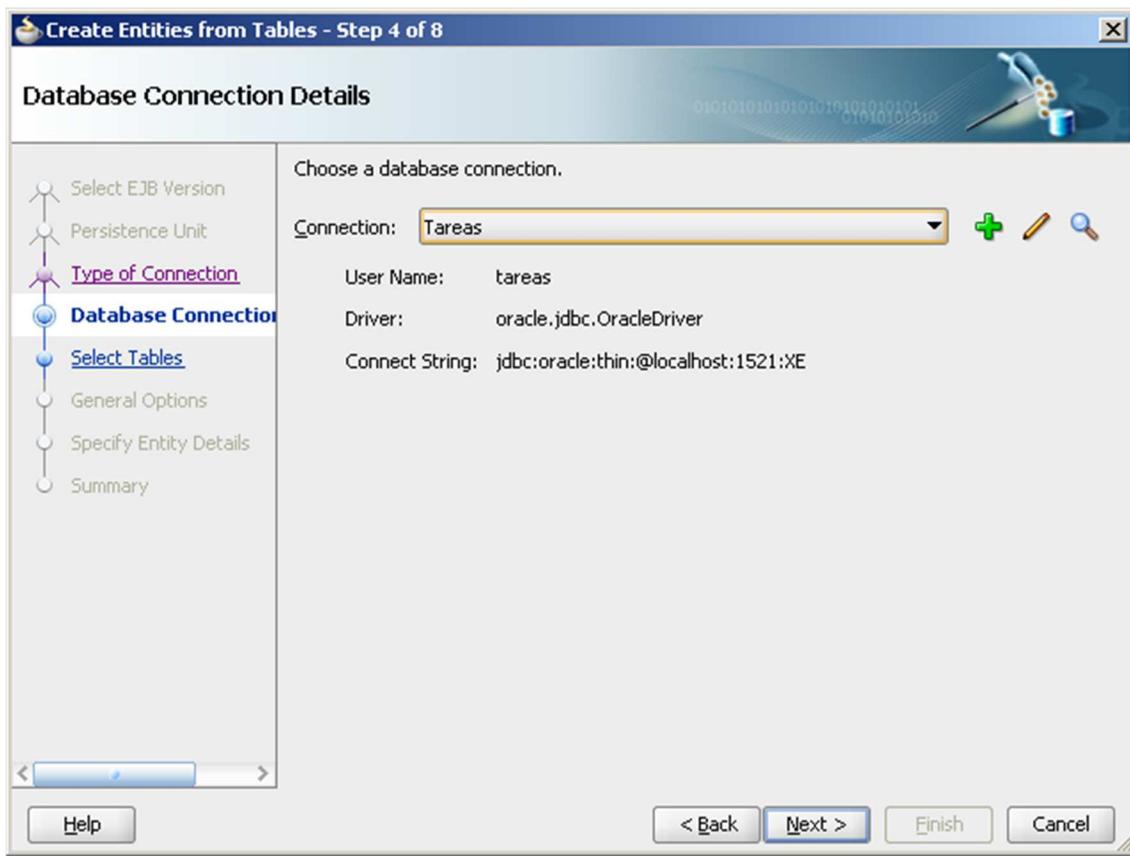


Ilustración 25: Crear entidades a partir de tablas 2

Ahora tenemos que seleccionar las tablas para las que vamos a crear entidades, que para nuestro caso son todas. Si no aparecen en la lista de la izquierda, seleccionamos la opción *auto-query* primero.

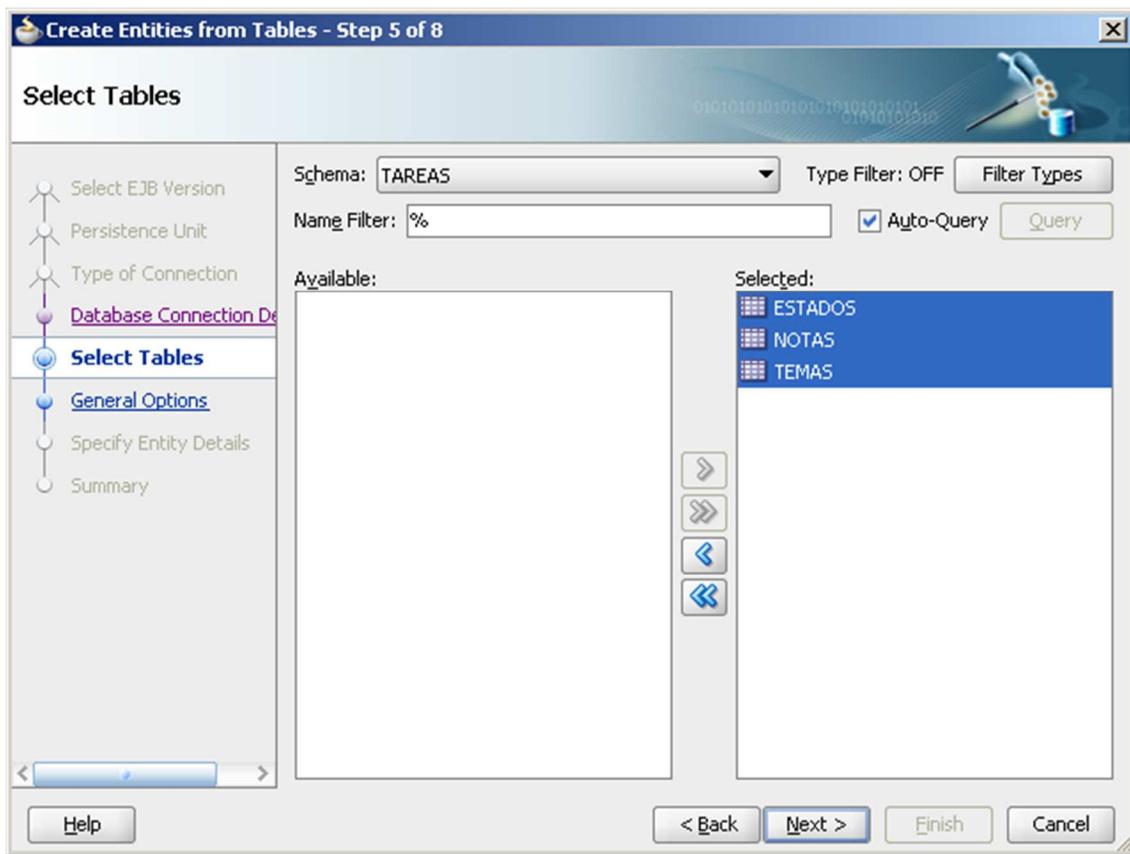


Ilustración 26: Crear entidades a partir de tablas 3

Continuamos indicando el paquete en el que queremos que se guarden nuestras entidades. Por defecto nos muestra el que indicamos al crear el proyecto, y dentro de él pondremos nuestras entidades en el paquete *entities*.

En esta misma pantalla nos dan a elegir a qué nivel queremos situar las anotaciones JPA. Las entidades son anotadas para indicar de qué forma se debe almacenar la información que contienen en la base de datos, y estas anotaciones se pueden poner en los atributos o métodos de la clase. Personalmente me resulta más clara la opción de anotar los atributos, así que será la que elija, pero debería ser indiferente.

Lo siguiente que tenemos que elegir en este paso es qué tipo usaremos para las relaciones que sean colecciones. De nuevo es cuestión de la preferencia de cada uno, y he optado por el tipo *List*.

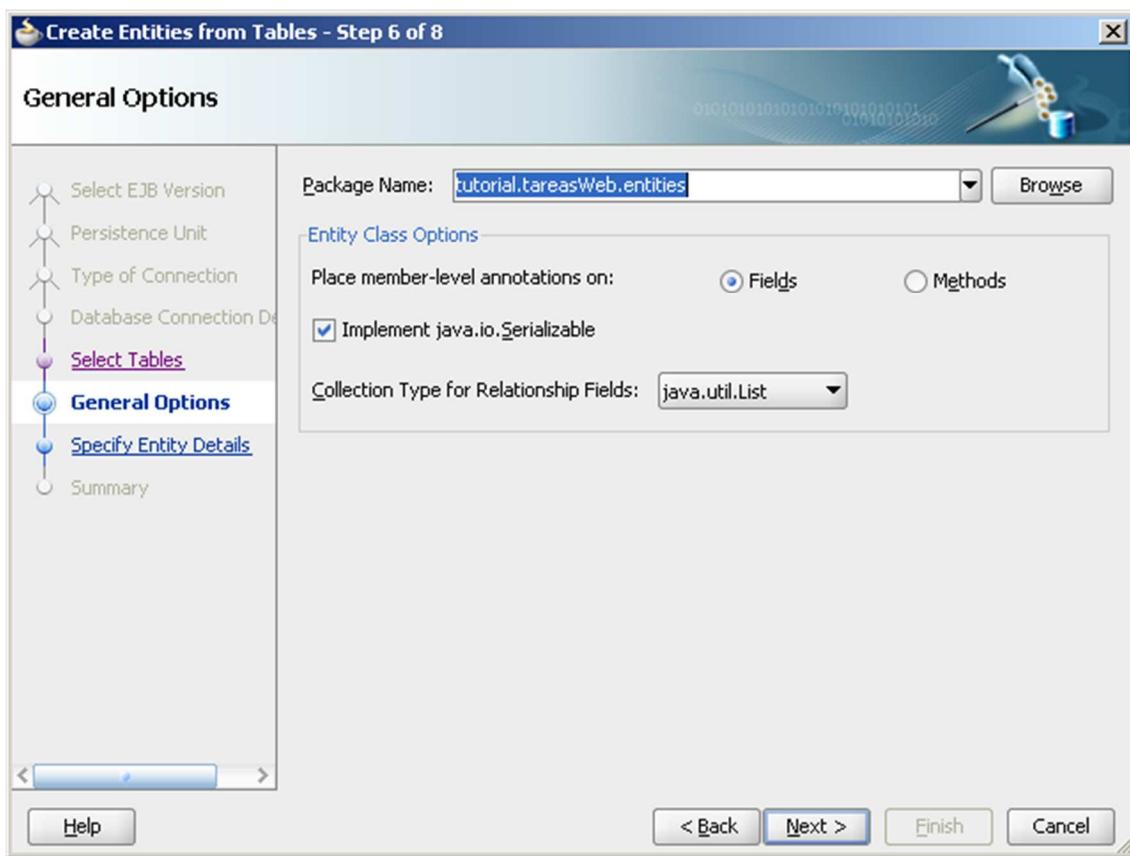


Ilustración 27: Crear entidades a partir de tablas 4

Llegamos al último paso, en el que podemos indicar el nombre de las clases de cada entidad, así como cambiar el paquete de alguna de ellas. Los nombres por defecto siguen la norma habitual en java, así que los aceptamos y pasamos al resumen final.

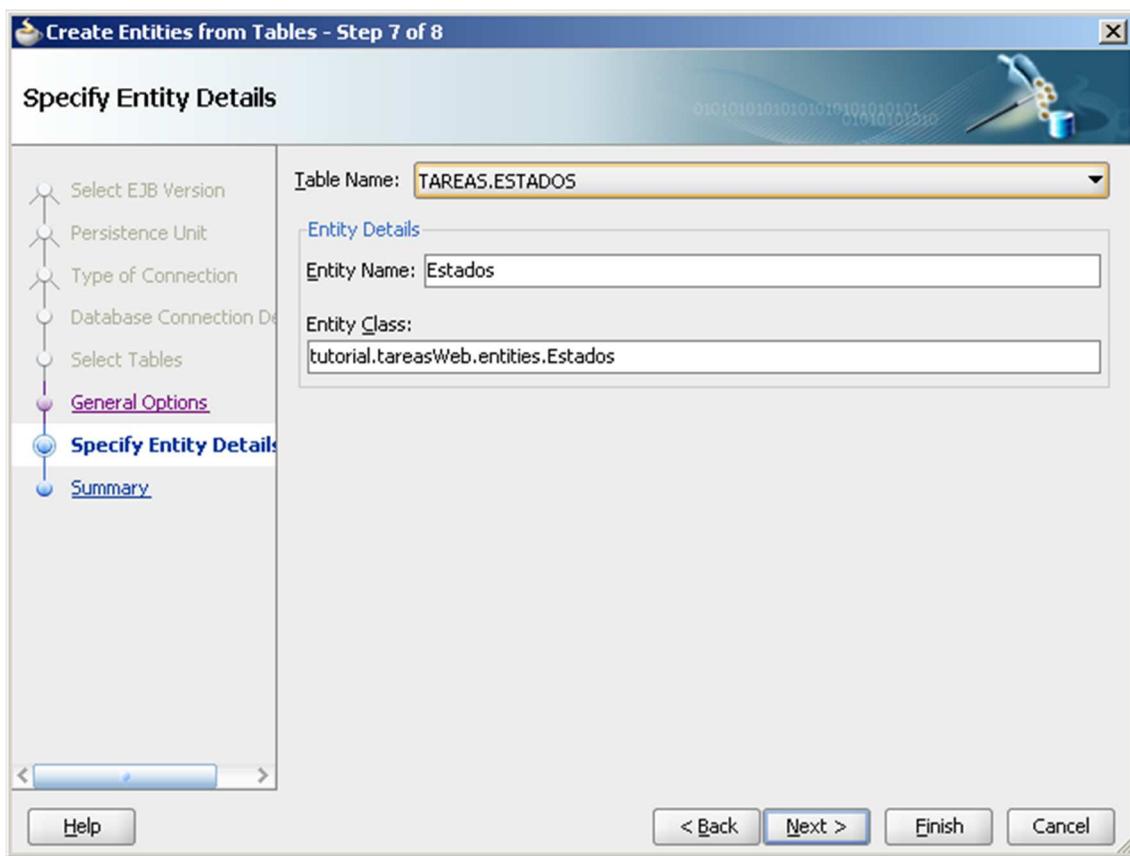


Ilustración 28: Crear entidades a partir de tablas 5

Repasamos la información por si se nos ha pasado algo, y finalizamos.

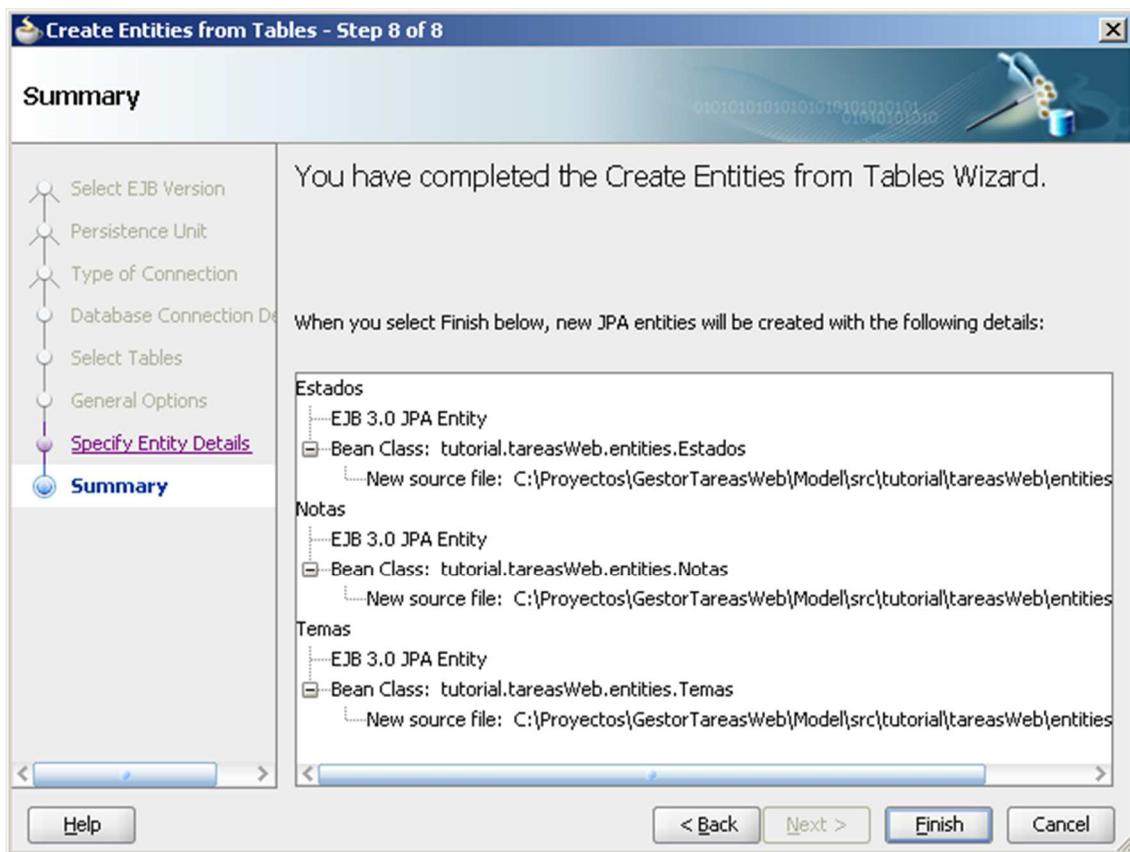


Ilustración 29: Crear entidades a partir de tablas 6

jDeveloper se pone entonces a trabajar durante unos instantes, tras los que podemos ver en el navegador de aplicaciones que se han creado nuestras tres entidades (con el icono de la judía, o bean). Podemos observar también que el *persistence.xml* aparece como modificado (nombre en cursiva).

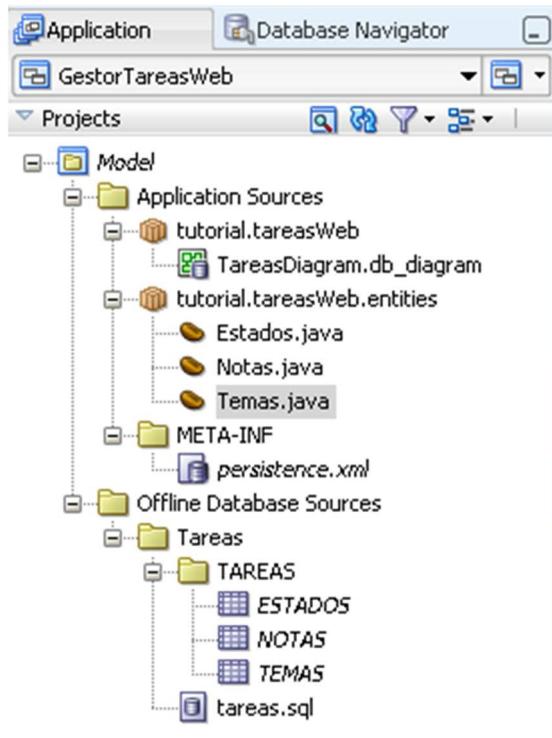


Ilustración 30: Navegador de aplicaciones

Configurando la unidad de persistencia

El fichero *persistence.xml* podemos consultarla directamente desde el IDE, y en él se definen las *persistence units*.

jDeveloper ofrece dos formas de consultar este tipo de ficheros. Directamente por código, a través de la pestaña *source*, o de un modo visual mediante la pestaña *overview*. En él podemos ver la unidad de persistencia, que es la que creamos con el nombre *persistenceUnit*.

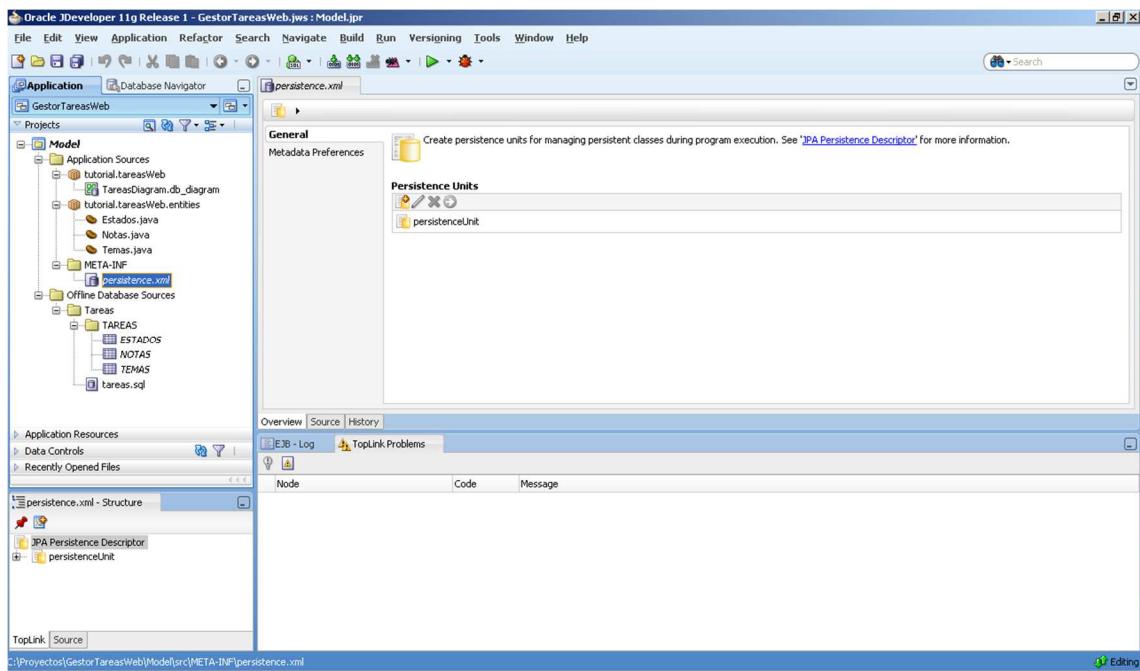


Ilustración 31: Edición de persistence.xml

Si seleccionamos nuestra unidad de persistencia, y pulsamos sobre el icono de flecha verde, se abrirá un cuadro desde el que podemos ver la configuración del fichero.

Lo primero que podemos ver es el proveedor de persistencia (*persistence provider*) que utilizaremos. Aquí podemos indicar qué implementación de JPA queremos. Es posible que la implementación más conocida y utilizada de JPA en la actualidad sea *Hibernate*; sin embargo, en este tutorial vamos a utilizar *Toplink*, en su versión 11.1.1.3.0. Esta es la implementación provista por Oracle, y además es la implementación de referencia de JPA. Si quisiésemos utilizar *Hibernate*, tendríamos que indicar a jDeveloper dónde encontrar las librerías necesarias.

Lo siguiente que podemos ir configurando en la misma pestaña es la conexión a la base de datos. En la sección *Development Database*, vemos que podemos seleccionar directamente aquellas bases de datos que nuestro jDeveloper tenga definidas a través de la pestaña *Database Navigator*. Como hemos configurado la nuestra, aparecerá en el desplegable.

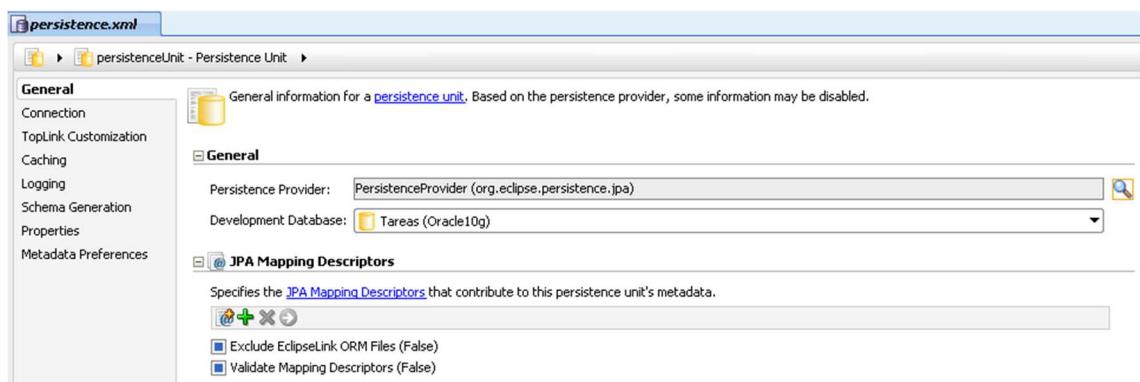


Ilustración 32: Edición de persistence.xml 2

En esta misma sección aparecen las tres entidades que hemos creado en nuestro proyecto. Además, en la opción *Connection*, sección *General*, vemos que se el entorno

ha dado un valor al nombre del *datasource* que utilizará JPA. Podemos guardar estos cambios.

Revisando las anotaciones

Vamos ahora a echar un vistazo a las entidades que el entorno ha creado para nosotros, y a explicar algunas cosas sobre ellas. Abrimos la clase *Temas.java*.

```
■ @Entity
  ■ @NamedQueries(
    ■ @NamedQuery(name = "Temas.findAll", query = "select o from Temas o")
  )
■ public class Temas implements Serializable {
  ■ private String descripcion;
  ■ private String incidencias;
■ ■ @Id
  ■ @Column(nullable = false)
  ■ private Long sid;
  ■ private String tema;
■ ■ @OneToMany(mappedBy = "temas")
  ■ private List<Notas> notasList;
■ ■ @ManyToOne
  ■ @JoinColumn(name = "FK_ESTADO")
  ■ private Estados estados;
```

Ilustración 33: Clase Temas

Lo primero que se puede observar, es que estamos ante una clase java normal y corriente, salvo por el hecho de que aparecen una serie de anotaciones. Esta palabra ya ha aparecido varias veces en el tutorial, pero ahora vamos a detenernos un poco más.

Una anotación no es más que una línea que se introduce en la clase java, y que se identifica porque comienza por el carácter `@`. Las anotaciones tienen un nombre, y pueden tener una serie de atributos. Así, la forma de incluir una anotación en el código sería:

```
@NombreAnotacion (nombreattr = valorattr, nombreattr = valorattr...)
```

Al trabajar con JPA podemos usar multitud de anotaciones. Aquí veremos algunas de ellas, pero para una descripción completa lo mejor es irse a la documentación. Por ejemplo en:

<http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html#Id>

La primera de ellas es la anotación `@Entity`, que lógicamente lo que hace es marcar a la clase como entidad. Por tanto sus instancias serán lo que se denomina como *Entity Objects*.

A continuación encontramos la etiqueta `NamedQueries`, que contiene una colección de consultas para el acceso a datos. Una `Named Query` se define como un par nombre-consulta. El nombre se usará para hacer referencia a la consulta, y poder utilizarla más adelante.

Las consultas se definen haciendo uso del lenguaje JPA *Query Language*, que es similar en cuanto a sintaxis a SQL, pero, por así decirlo, orientado a objetos. Es decir, que en lugar de hacer consultas sobre una tabla de una base de datos, se hacen como si se estuviese consultando un objeto. Si observamos las *Named Queries* creadas por el entorno será más fácil entender esta diferencia. Por ejemplo, la consulta para obtener un listado con todos los alumnos sería:

```
select o from Alumno o
```

Mientras que su equivalente en SQL hubiese sido:

```
select * from Alumno
```

Continuamos examinando las etiquetas de nuestras clases de entidad. Podríamos decir que las etiquetas que hemos visto hasta el momento tienen un nivel de clase, y están situadas antes de la cabecera de definición de la misma. Las siguientes, por el contrario, están ubicadas dentro de la definición, y afectan a los campos de la clase. El IDE ha generado las anotaciones junto a la definición de los atributos, porque así lo indicamos en el wizard anterior pero, como ya se comentó, también podrían haberse situado junto a la definición de los métodos de acceso a los mismos.

Por defecto, JPA toma todos los campos de una clase como persistentes, y entenderá que el nombre de estos se corresponde con el nombre de las correspondientes columnas de la base de datos. En nuestro caso, tenemos el atributo *tema*, que no tiene ninguna etiqueta; por tanto, implícitamente estamos indicando que dicho campo se persistirá en la columna *tema*, de la tabla *temas*.

En caso de que algún atributo tuviese un nombre distinto de la columna en que se persiste, se indicaría mediante la anotación *@Column*.

Además del nombre del campo, la etiqueta *@Column* permite indicar otras cosas como si el atributo puede ser nulo, si tiene que ser único, o cuál es su longitud máxima.

A la hora de mapear un atributo con la base de datos nos quedaría una tercera opción (tras el mapeo a una columna con el mismo nombre, y a una con distinto). Esta última opción es el no mapeo del atributo. Puede darse el caso de que tengamos un campo en nuestra clase que no tenga que ser persistente. En la clase Alumno no se da el caso, pero bastaría con marcar el atributo con la etiqueta *@Transient*.

La etiqueta *@Id*, que también podemos ver en el atributo *sid*, sirve para indicar cuál es la clave primaria de la entidad. Esta etiqueta no tiene ningún atributo, y puede ser usada sobre más de un campo en combinación con la etiqueta *@IdClass*, pero en nuestro caso la usaremos para claves con un único campo.

Para las claves primarias también podemos indicarle a JPA cómo queremos que sean generadas. En nuestro caso, hemos creado una secuencia Oracle para cada una de las tablas, por lo que vamos a añadir las siguientes anotaciones al atributo *sid*.

```
@GeneratedValue(strategy=GenerationType.SEQUENCE ,  
generator="SEQ_TEMAS")  
@SequenceGenerator(name="seqTemas" ,sequenceName="SEQ_TEMAS" ,  
allocationSize=1)
```

Al hacerlo el IDE marcará una serie de errores que solucionamos importando las clases que nos recomienda. Como puede verse, mediante estas anotaciones estamos indicado que queremos utilizar la estrategia de generación de claves basada en secuencias, y el nombre de la misma. Tenemos que copiar la anotación en nuestras tres entidades, modificando en cada una de ellas los atributos *name* y *sequenceName*.

Las últimas etiquetas que tenemos en nuestro ejemplo son aquellas que definen la relación entre entidades, o claves ajenas del modelo de datos. La tabla *estados* tiene una única relación, de uno a muchos, con la tabla *temas*. Esta relación se traslada al modelo de clases añadiendo una colección de *temas* a los *estados*, con el atributo *temasList*, y un objeto de tipo *Estado* a los *temas*, con el atributo *estados*. De esta forma ambos extremos de la relación contienen una referencia al otro.

Obviamente para que JPA gestione esta relación nos falta añadir las correspondientes anotaciones. Vamos a empezar fijándonos en el atributo *estados* de la clase *Temas*, que corresponde al extremo “muchos” de la relación. Este atributo se anota con las etiquetas *@ManyToOne* y *@JoinColumn*. Con la primera indicamos el tipo de relación, mientras que con la segunda, a través del atributo *name*, especificamos qué columna del modelo de datos es la clave ajena para esta relación.

```
@ManyToOne
@JoinColumn(name = "FK_ESTADO")
private Estados estados;
```

Si nos vamos ahora al otro extremo, a la entidad *Estados*, vemos que en este caso la etiqueta utilizada es *@OneToMany*, como es lógico, y que se especifica el atributo *mappedBy*, para indicar qué campo de la clase destino es utilizado para referenciar a nuestra clase.

Es posible que todo esto resulte difícil de comprender explicado verbalmente, así que en el siguiente resumen gráfico se puede ver la relación entre los valores que definen la relación.

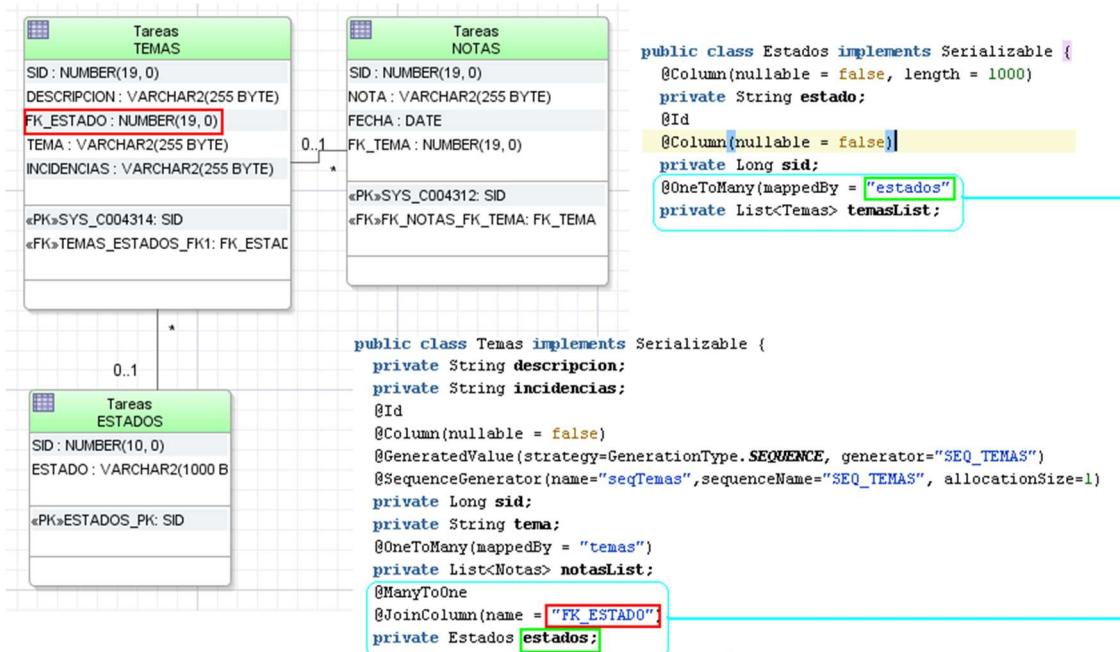


Ilustración 34: Relación uno a muchos en JPA

Con esto podemos entender todo lo relativo a cómo ha creado jDeveloper nuestras entidades.

Creando el EJB de Sesión

Para terminar con nuestra capa de modelo, vamos a crear un EJB de sesión. Como ya dijimos, este objeto nos servirá de fachada, y será con el que trabajen nuestros clientes. El uso de la fachada reduce el número de llamadas necesarias para acceder a los EJB, y por tanto el tráfico de red y el tiempo de respuesta, además de independizar al cliente de cambios en la implementación.

Para crear el EJB, accedemos a la opción para crear nuevos elementos en nuestro proyecto, y dentro de la sección *Business Tier – EJB*, seleccionamos *Session Bean*.

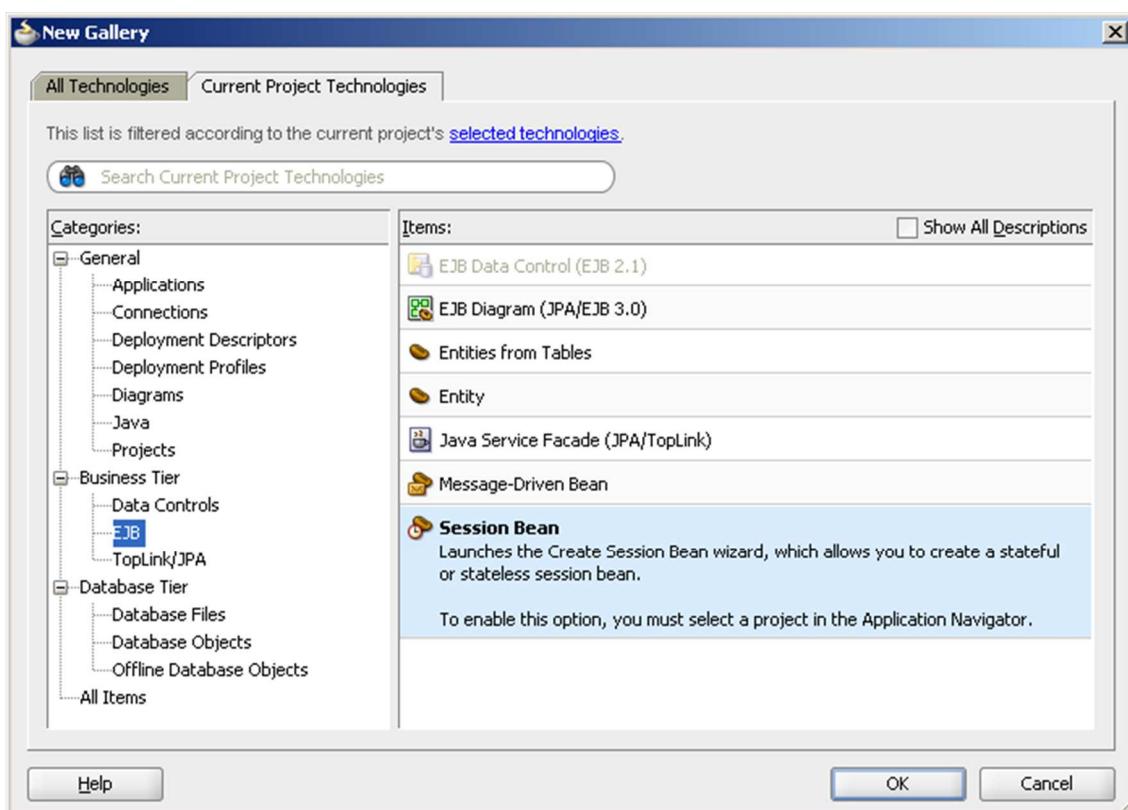


Ilustración 35: Nuevo EJB de Sesión

Lo primero que nos pide el wizard es que demos un nombre a nuestro bean. Por ejemplo, *TareasSessionEJB*. El tipo será *Stateless* (ya se explicó someramente qué quería decir esto), y las transacciones serán manejadas por el contenedor. Marcaremos también la opción para generar los métodos de la fachada. Además se nos da la opción de seleccionar la unidad de persistencia con la que trabajará el EJB; en nuestro caso sólo tenemos una que aparecerá seleccionada por defecto.

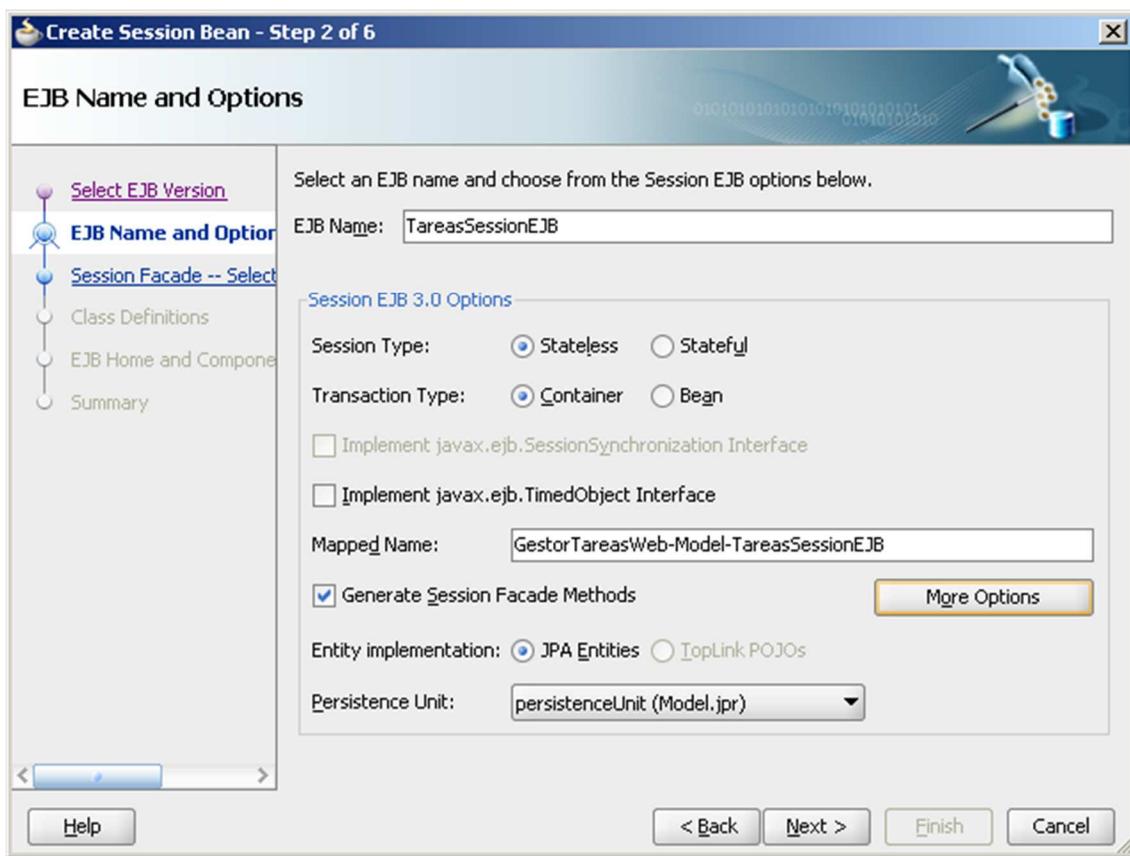


Ilustración 36: Crear session bean

En el siguiente paso el wizard nos ofrecerá una serie de métodos para incluir directamente en nuestra interfaz. Por defecto estarán seleccionados todos los disponibles.

En el siguiente paso tenemos que indicar la ruta de la clase en la que se implementará el EJB, y el directorio donde se ubica el código fuente (el src de nuestro proyecto). Vamos a utilizar el paquete `session` en lugar de `entities`, para separar el bean de sesión de los de entidad.

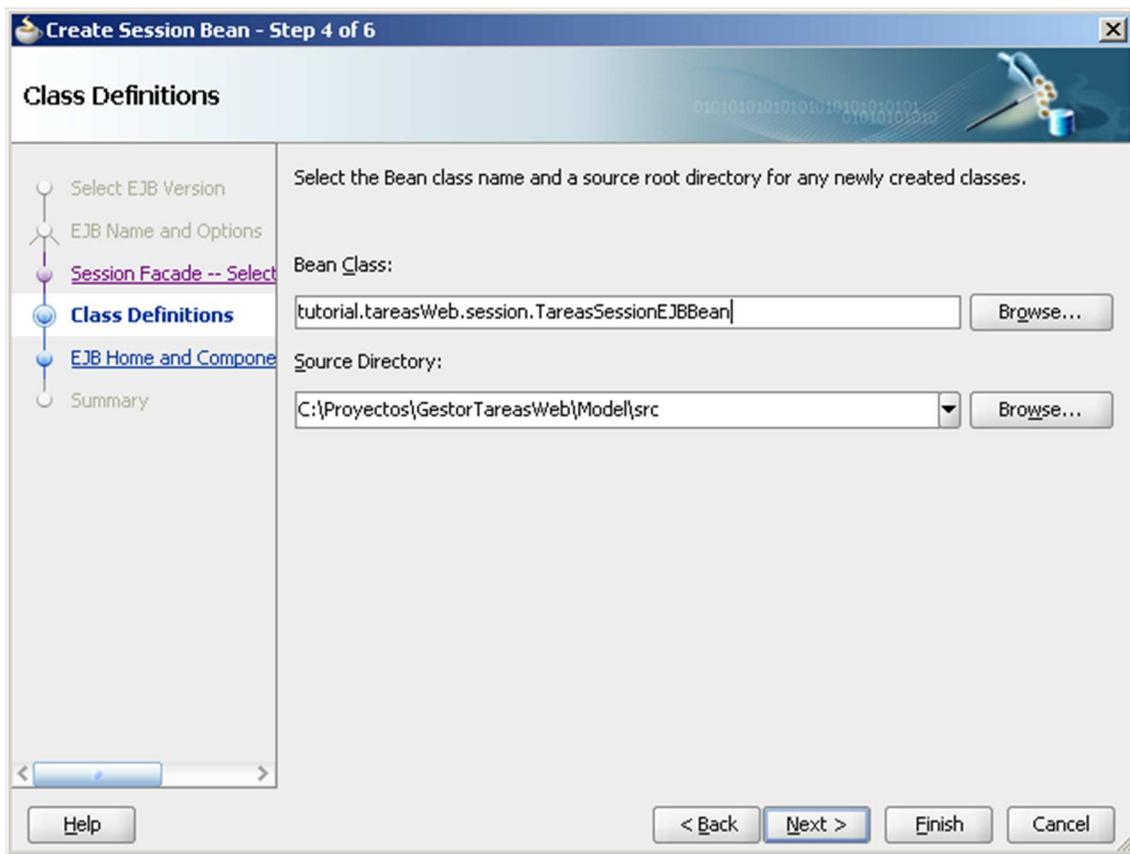


Ilustración 37: Crear session bean 2

Para terminar sólo queda indicar qué interfaces queremos generar. Las opciones que tenemos son la local y la remota. La interfaz remota es necesaria para clientes que están ejecutándose en una máquina virtual diferente a la del EJB, mientras que la local es para aquellos que se ejecutan en la misma máquina virtual.

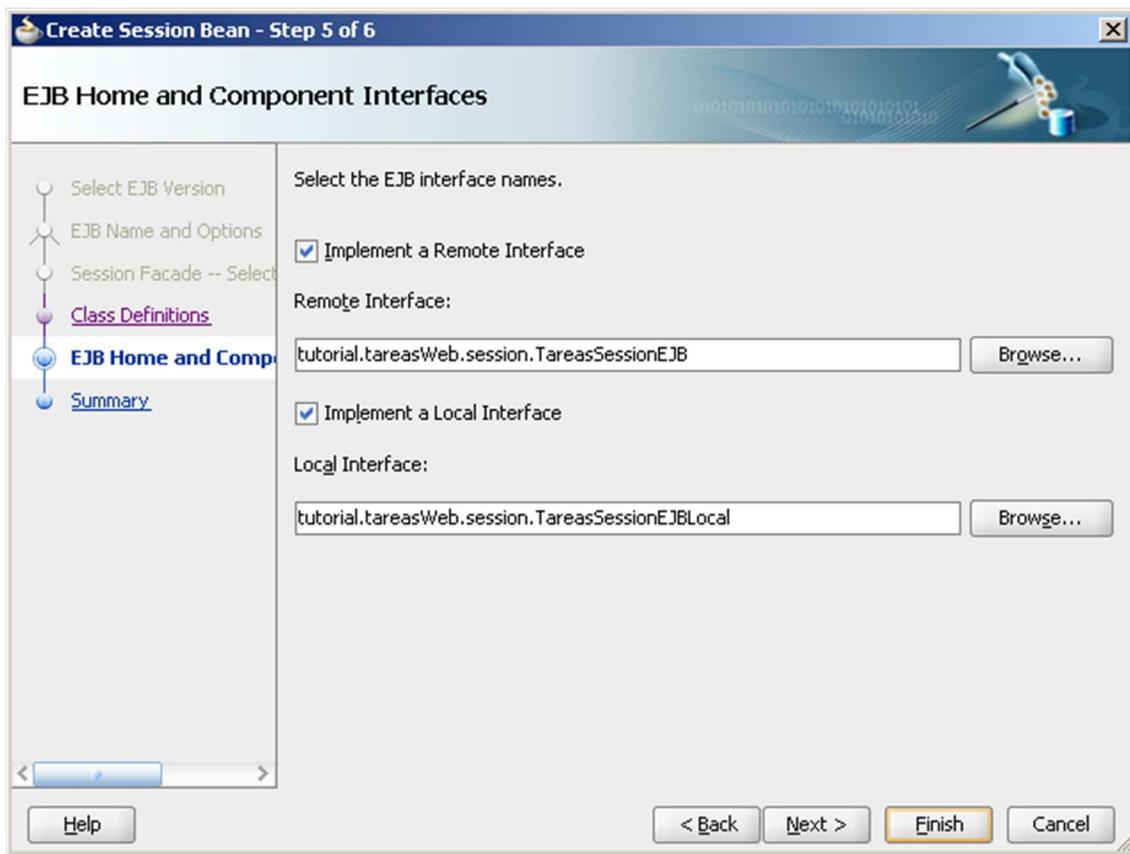


Ilustración 38: Crear session bean 3

Tras el paso de resumen, terminamos el wizad, y echamos un vistazo a lo que el IDE ha generado por nosotros. Se ha creado el paquete `tutorial.tareasWeb.session`, en el que le hemos dicho que meta el EJB, y dentro tenemos tres clases java, como se puede ver en la siguiente imagen.

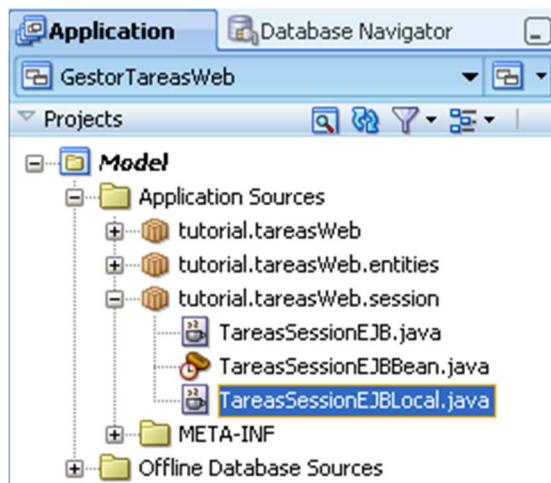


Ilustración 39: Navegador de aplicaciones

Estos tres ficheros .java son las dos interfaces del EJB (local y remota), y su implementación.

`TareasSessionEJB.java` es la interfaz remota. Si vemos el código, es una interfaz java normal, con la anotación `@Remote`.

```
@Remote  
public interface TareasSessionEJB {  
    Ilustración 40: Anotación Remote
```

TareasSessionEJBLocal.java es exactamente igual, pero en este caso con la etiqueta @Local

```
@Local  
public interface TareasSessionEJBLocal {  
    Ilustración 41: Anotación Local
```

Mientras que *TareasSessionEJBBean.java* es la implementación de las dos interfaces anteriores. En este caso tenemos una clase que está anotada tanto con @Remote como con @Local, además de como @Stateless, indicando que la clase es un EJB de sesión sin estado.

Además, encontramos la anotación @PersistenceContext. Gracias a esta anotación podemos ahorrarnos el código necesario para obtener una referencia a nuestra unidad de persistencia, ya que esta será inyectada por el contenedor de EJBs en el campo etiquetado.

```
@Stateless(name = "TareasSessionEJB", mappedName = "GestorTareasWeb-Model-TareasSessionEJB")  
@Remote  
@Local  
public class TareasSessionEJBBean implements TareasSessionEJB,  
                                         TareasSessionEJBLocal {  
    @PersistenceContext(unitName="persistenceUnit")  
    private EntityManager em;  
  
    public TareasSessionEJBBean() {  
    }
```

Ilustración 42: Anotaciones del session bean

Por tanto, ya tenemos la estructura montada, con nuestras dos interfaces, y la implementación. Si en algún momento queremos cambiar el listado de métodos publicados a través de este EJB podemos hacerlo pulsando con el botón derecho sobre él, y accediendo a la opción *Edit Session Facade....*. Veremos un diálogo con opciones similares a las que se nos ofrecían en el paso 2 del wizard de creación del EJB. En él podemos ver las diferentes entidades de nuestro proyecto y, bajo cada una de ellas, un método para crear, actualizar y eliminar, así como uno por cada una de las *named queries* definidas para la entidad.

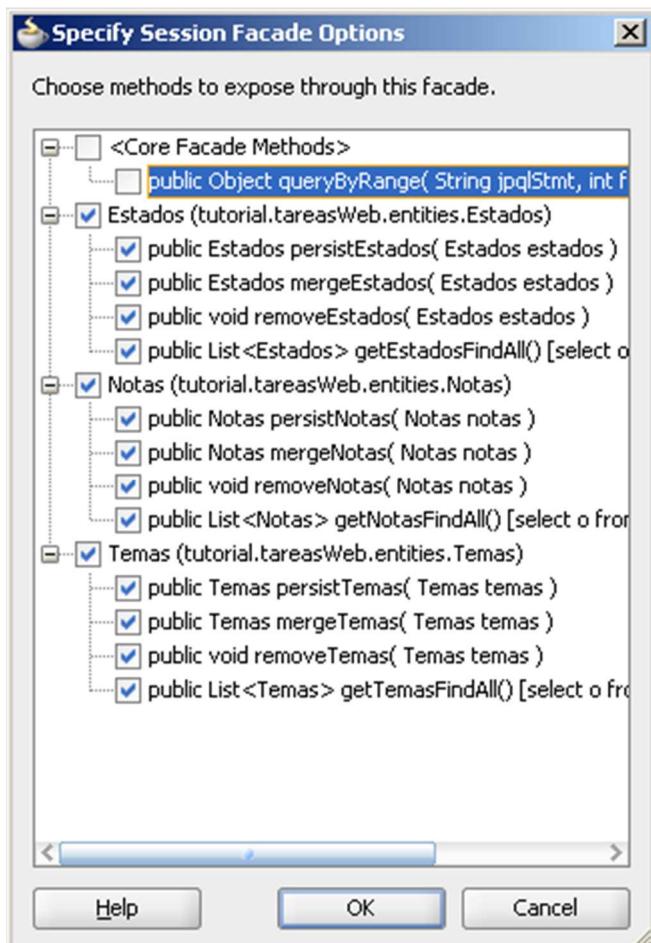


Ilustración 43: Especificar opciones del session facade

Para comprobar su funcionamiento podemos deseleccionar el método `queryByRange`, ya que no lo utilizaremos en nuestro desarrollo. Si vamos ahora al código del bean veremos que ha sido eliminado.

Hasta aquí llegamos por el momento con el desarrollo de nuestro modelo. Haciendo un poco de memoria, hemos diseñado nuestra base de datos, generando a partir del diseño tanto los propios objetos de base de datos como las entidades JPA de nuestro proyecto. Por último hemos generado un EJB de sesión que sirve como fachada para el acceso de nuestros clientes.

El proyecto ViewController

Vamos a pasar a generar otro proyecto dentro de nuestra aplicación en el que tendremos la vista y controlador haciendo uso de tecnología JSF. Para ello, además de acceder a través de la opción del menú `File`, como hicimos para crear el modelo, podemos usar el menú rápido que tenemos en el navegador de aplicaciones.

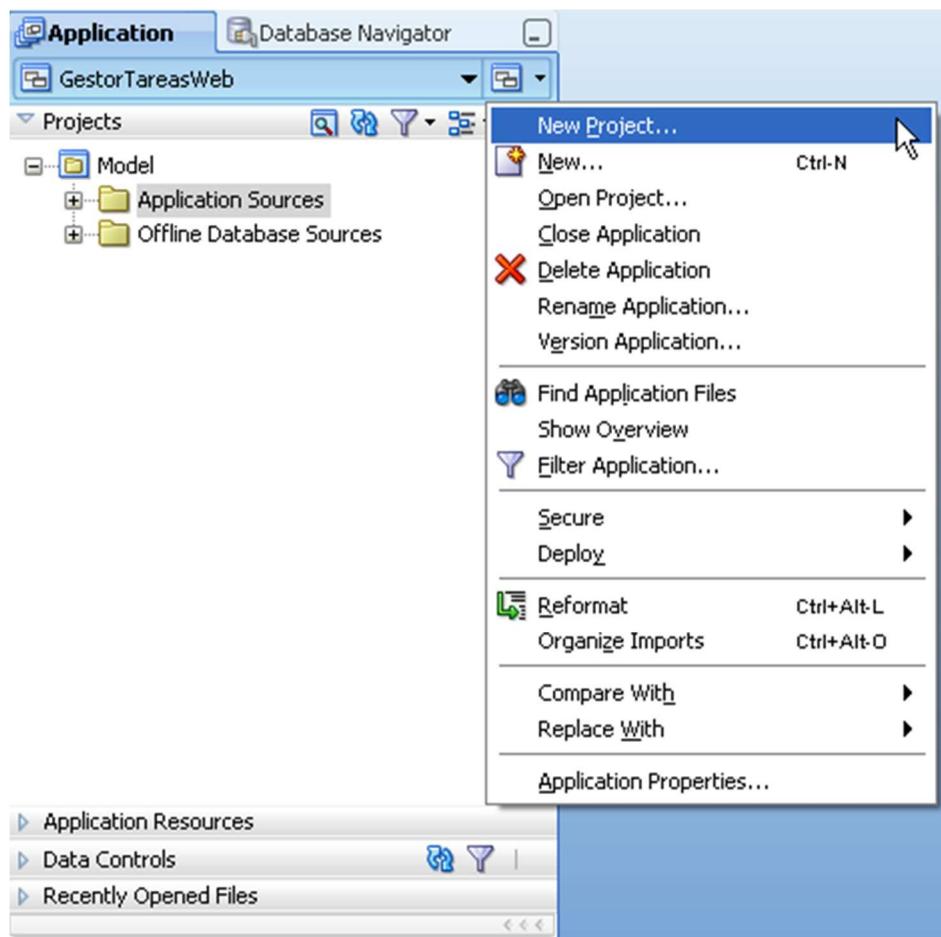


Ilustración 44: Menú nuevo proyecto

En este caso, en lugar de crear un proyecto genérico, vamos a aprovechar los proyectos predeterminados de jDeveloper para poder ver el ahorro de tiempo que supone. Crearemos un *ADF ViewController Project*. De esta forma accedemos a un diálogo en el que podemos elegir el nombre y ubicación del proyecto. Además vemos pestañas en las que se nos muestran las tecnologías y librerías del proyecto, y los componentes que se van a generar automáticamente. Podríamos añadir más tecnologías a nuestro proyecto, pero por defecto tenemos todas las que vamos a necesitar.

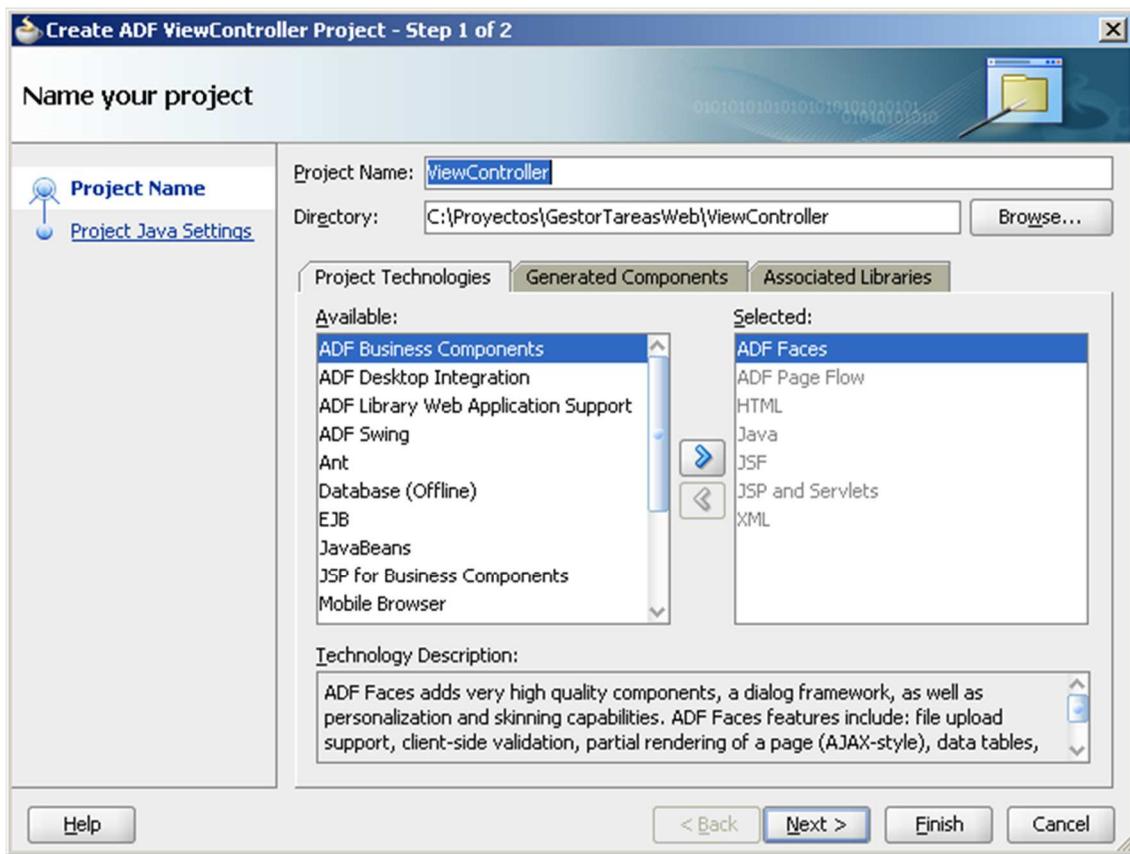


Ilustración 45: Crear proyecto ADF ViewController

A continuación indicamos el paquete por defecto, y las rutas para los fuentes y compilados. Podemos mantener los valores por defecto.

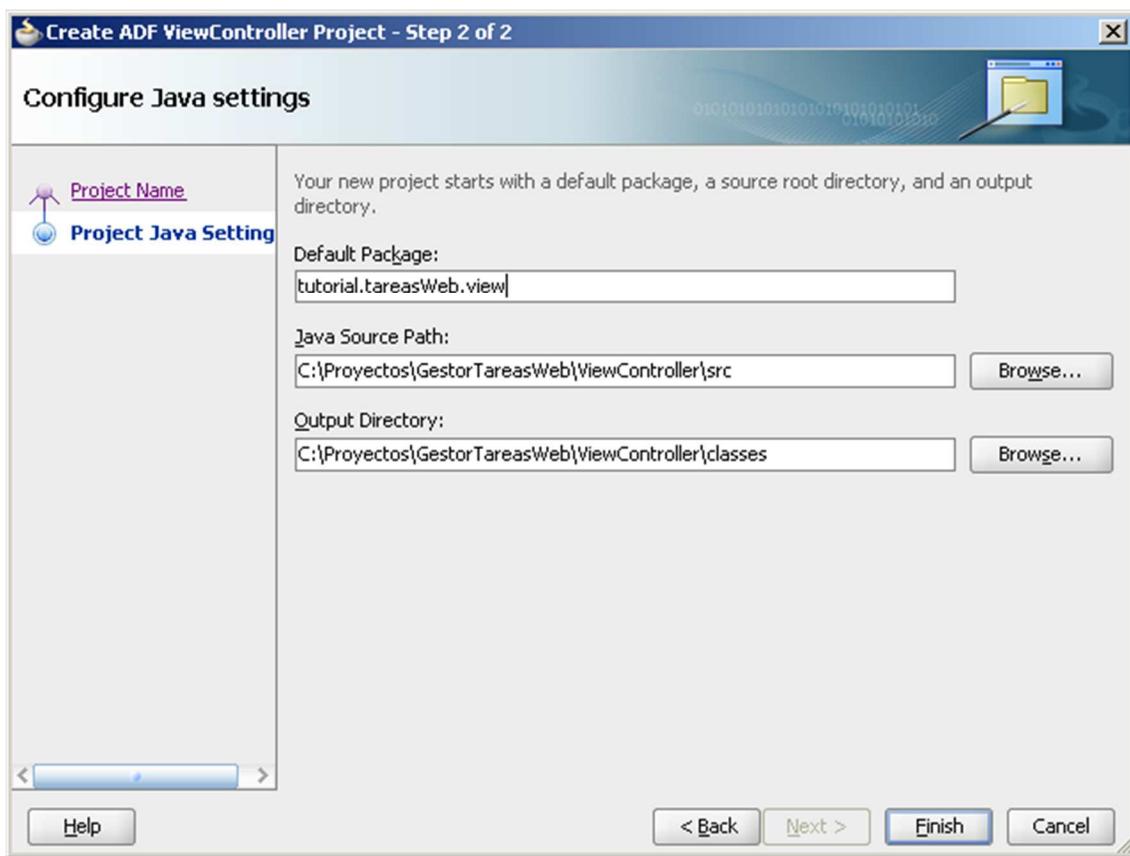


Ilustración 46: Crear proyecto ADF View Controller 2

Al pulsar sobre finalizar se generará el proyecto con varios ficheros necesarios.

Uno de los ficheros más importantes de cualquier proyecto JSF es el *faces-config.xml*. Este fichero contiene información tal como qué páginas vamos a tener, qué reglas de navegación entre ellas van a existir, o con qué Beans vamos a trabajar. En jDeveloper podemos editarla de forma visual, y utilizarla para crear desde ella nuestras páginas.

Creando páginas jspx

Para comenzar vamos a crear una página de consulta, desde la que podremos ver el listado de temas. Para cada uno de ellos podremos acceder a las anotaciones que tienen asociadas.

Podemos crear la página fácilmente desde el editor gráfico del faces-config. En la paleta de componentes seleccionamos *JSF Page*, y luego pulsamos sobre el lugar donde queramos poner la nueva página. Le ponemos el nombre *consulta.jspx*.

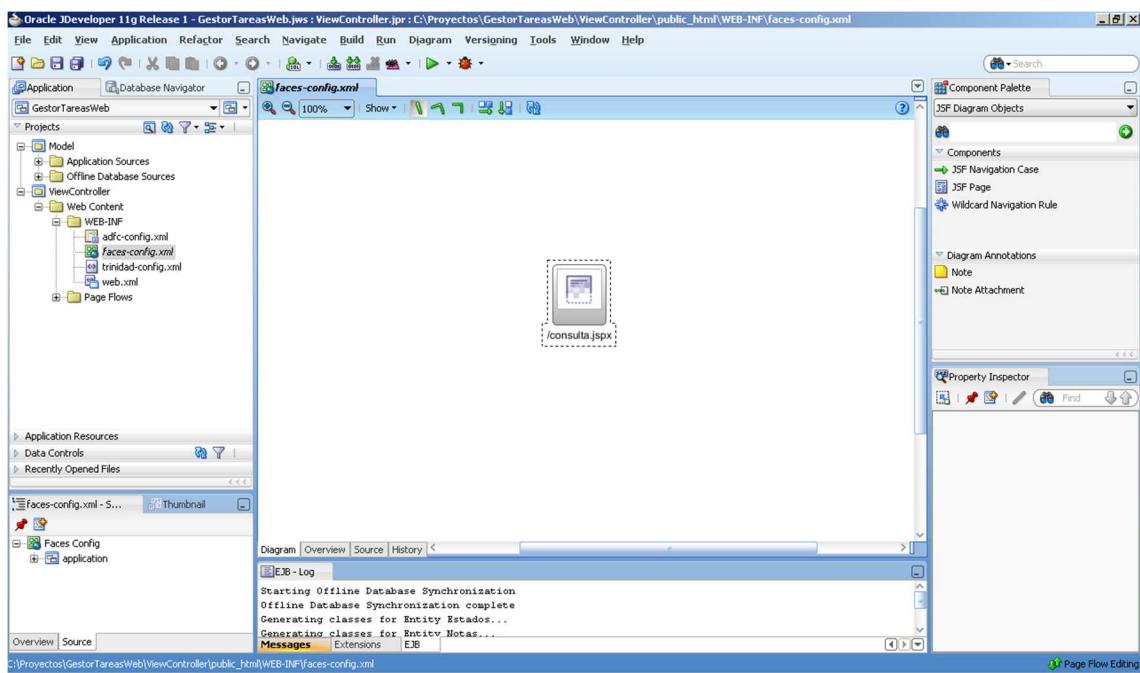


Ilustración 47: Nueva página en el faces-config.xml

A continuación podemos hacer doble clic sobre la página, y se abrirá el wizard de creación, desde el que podremos configurar los datos básicos, como el nombre y la ubicación. En general las páginas siempre se ubicarán en la carpeta public_html del proyecto, ya sea directamente, u organizadas en subcarpetas.

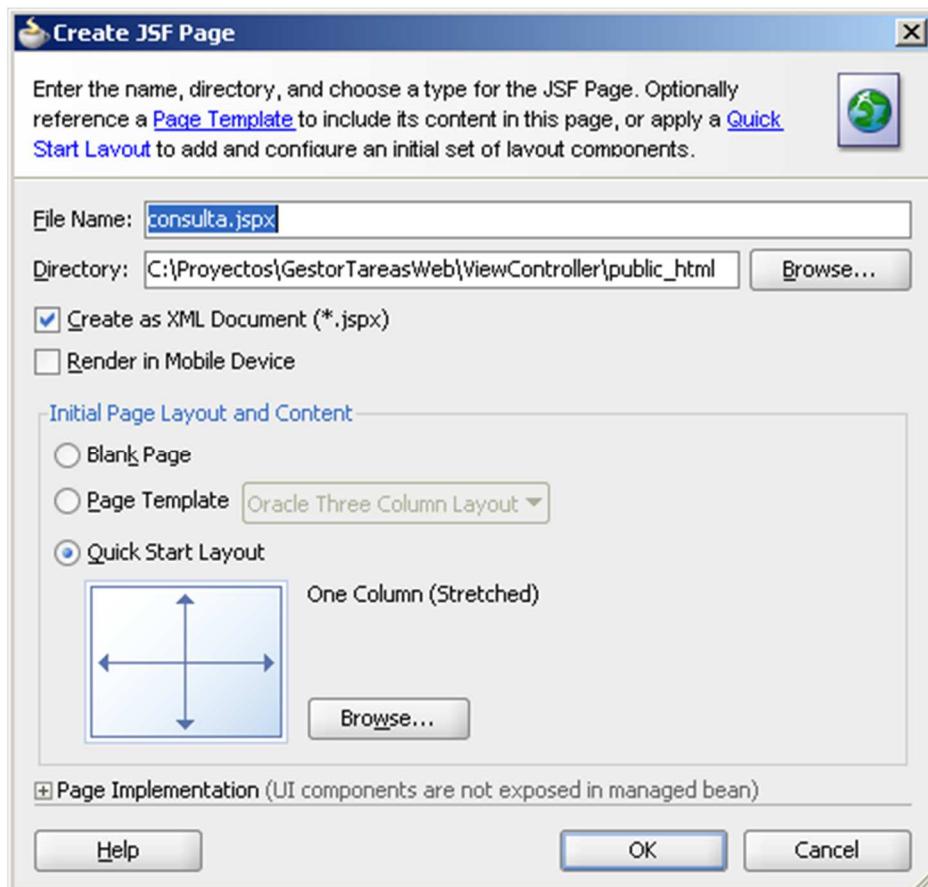


Ilustración 48: Crear página JSF

Además tenemos otras opciones. La primera de ella es la de crear la página como un documento XML. Trabajando con JSF podemos crear tanto página jsp como jspx. Las primeras son de sobra conocidas, y permiten el uso tanto de etiquetas, como de *scriptlets*. Por contra, las páginas jspx son documentos XML bien formados.

La siguiente opción nos indica si queremos que la página se muestre en terminales móviles. No es el objetivo de este tutorial, así que no marcaremos esta opción.

Podemos decidir también si queremos crear una página en blanco, o definir directamente la distribución que tendrá. Seleccionaremos la opción Quick Start Layout y accedemos a las opciones con el botón Browse. En la nueva pantalla dejaremos los valores por defecto, salvo para la opción Apply Themes, que marcaremos para obtener una Web con una apariencia mejorada.

Por último tenemos opciones relativas a la implementación de la página, que nos permiten generar o seleccionar un *Backing Bean* asociado. No seleccionaremos esta opción.

Tras estos pasos podemos ver que jDeveloper ha creado la página en la ruta y con las propiedades indicadas.

Dando contenido a las páginas

Ahora vamos a empezar a añadir componentes JSF a la página, para ir componiéndola. Para este trabajo tenemos que conocer algunas de las pestañas que muestra jDeveloper. La primera, por supuesto, es la paleta de componentes, en la que podemos ver todos los componentes disponibles para usar en nuestra aplicación.

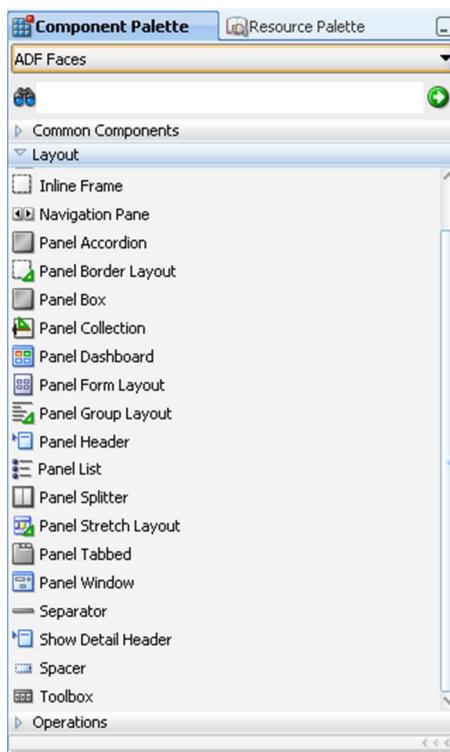


Ilustración 49: Paleta de componentes

En la parte superior tenemos un desplegable para elegir el conjunto de componentes, que luego aparecen agrupados por su tipo. Podemos ver que tenemos

disponibles tanto los componentes estándar de JSF, como los componentes de la implementación ADF Faces (además de HTML, JSP, etc).

Otra vista interesante es la de estructura, en la que podemos ver de forma gráfica, a través de un árbol, los componentes que tiene una página, y la relación entre ellos. Esta vista está disponible para casi cualquier elemento, (por ejemplo, para las clases java podemos ver en ella los atributos y métodos que la componen) y es muy interesante para localizar de forma rápida un fragmento específico de código. Además, podemos trabajar directamente con ella para ir añadiendo elementos, usando los menús que se acceden clicando con el botón derecho, o arrastrando componentes.

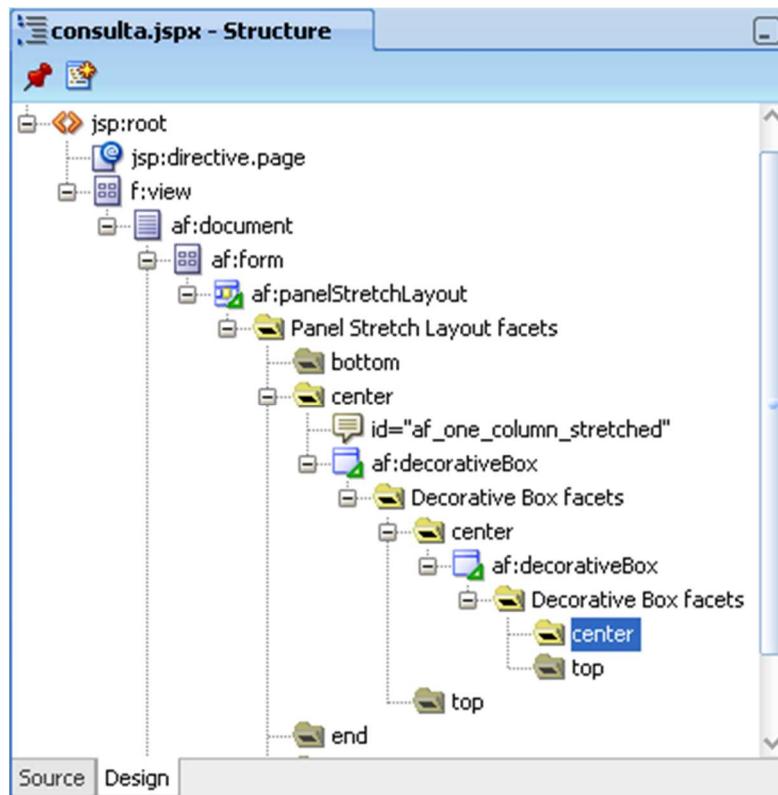


Ilustración 50: Vista estructura

Lo primero que vamos a añadir a nuestra página es un *Panel Stretch Layout*, que encontramos en la categoría *Layout*, y con el que la dividiremos en 5 secciones (superior, inferior, izquierda, derecha y central). Lo seleccionamos en la paleta, y lo arrastramos, bien directamente sobre la vista diseño, o bien sobre el elemento *center* de la vista de estructura. En el editor podemos ver las 5 secciones. A continuación añadimos al centro de nuestra página un panel dividido (componente *Panel Splitter* de ADF Faces), que nos permitirá tener a su vez dos secciones en el centro de la página.

En lugar de andar buscando los componentes por la paleta, podemos escribir alguna palabra en la sección de búsqueda que está al comienzo de la misma, y obtendremos el que nos interesa directamente.

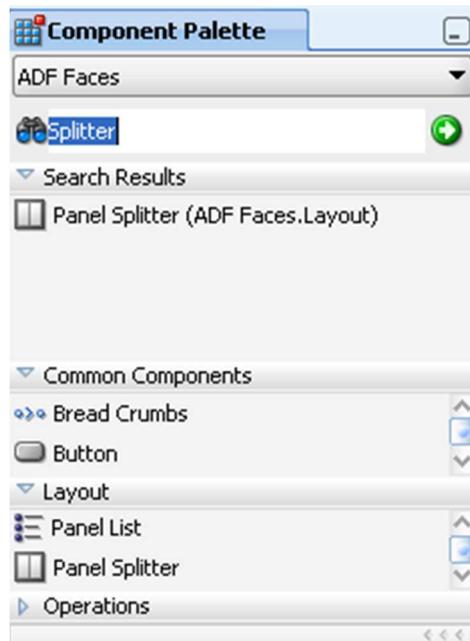


Ilustración 51: Paleta de componentes

El resultado en el editor sería el siguiente.

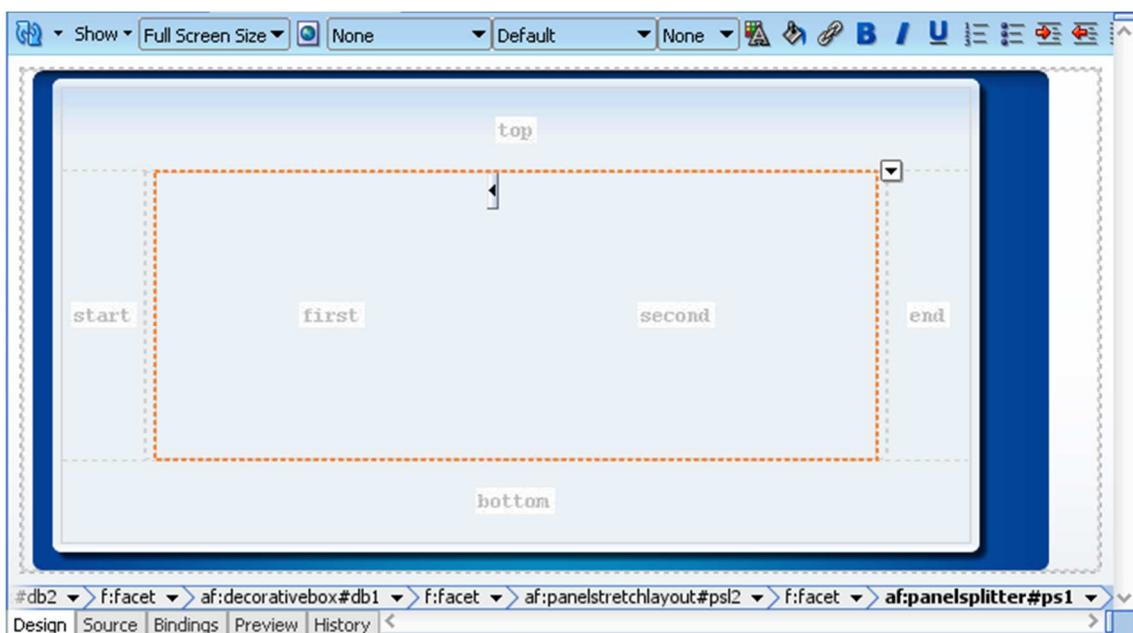


Ilustración 52: Editor jspx de diseño

Ahora vamos a agregar un menú en la sección superior de la página. Para ello usamos el componente *Panel Menu Bar*, que está dentro del grupo *Common Components* de *ADF Faces*. A su vez, dentro de la barra vamos a añadir un menú, pero en lugar de hacerlo arrastrando el componente desde la paleta, vamos a irnos a la vista de estructura, vamos a localizar la barra, y vamos a utilizar la opción de menú.

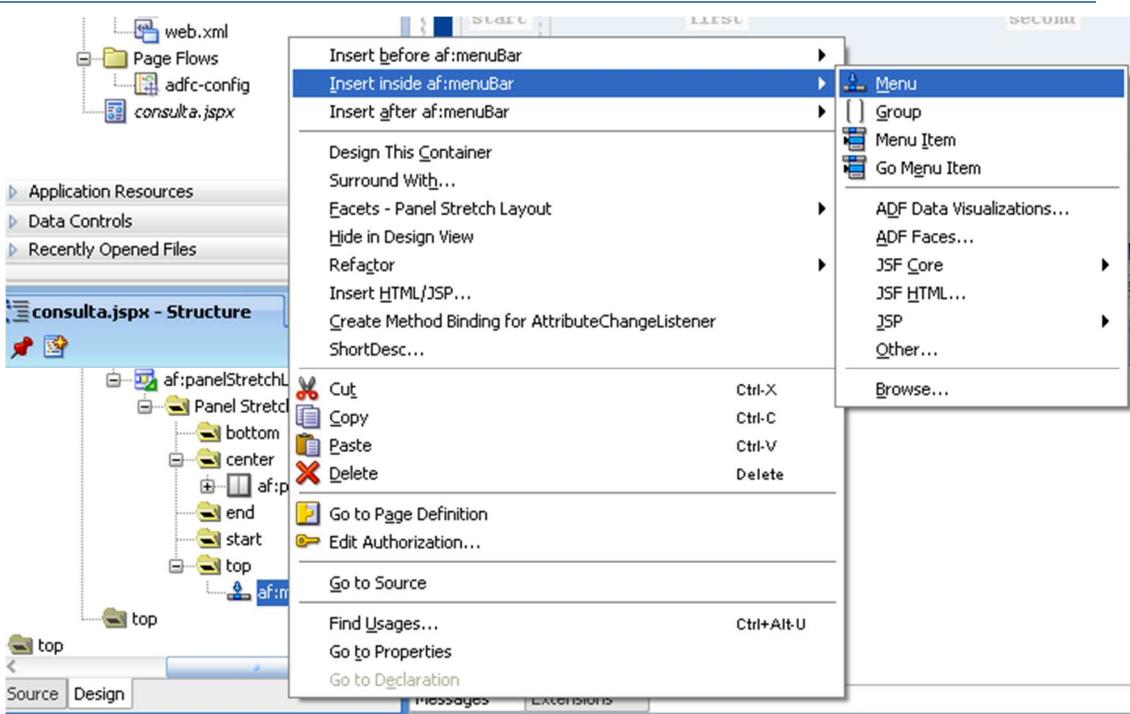


Ilustración 53: Crear nuevo menú

De esta forma tenemos el primer elemento de nuestra barra de menú, al que a su vez vamos a añadir un hijo (*Menu Item*) de la misma forma. De momento lo dejamos así, y más adelante lo completaremos con las etiquetas y reglas de navegación adecuadas.

Ya podemos dar por buena la estructura de la página, por lo que ahora tenemos que ver cómo acceder al modelo de nuestra aplicación para obtener los datos que mostrar.

ADF Bindings

Ya vimos en la introducción a ADF cómo este framework añade una capa de enlace de datos para independizar el modelo y la vista de las aplicaciones. En la nuestra, hasta el momento, hemos creado el modelo, y tenemos también ya una primera parte de la vista, por lo que necesitamos ponerlas en contacto.

Concretamente, lo que queremos para empezar es poder obtener una lista de los temas almacenados en nuestro sistema, para mostrarla en nuestra Web. Si recordamos, cuando implementamos el modelo creamos un EJB de sesión llamado *TareasSessionEJB*, en cuya interfaz publicamos un método *getTemasFindAll*, que nos proporciona la información que necesitamos. Pero, ¿cómo tenemos que acceder a este método? A través de los *Data Controls*, y *Data Bindings*.

Ya vimos que un *Data Control* pretendía lograr la abstracción respecto a la implementación de los servicios de negocio, que en nuestro caso se ha hecho con EJBs. La creación del Data Control es muy simple gracias a nuestro IDE. Basta con seleccionar un componente de nuestra implementación, y decirle que cree un Data Control a partir de éste, mediante el menú accesible con el botón derecho del ratón. Vamos a hacerlo para la implementación del EJB de sesión, *TareasSessionEJBBean*.

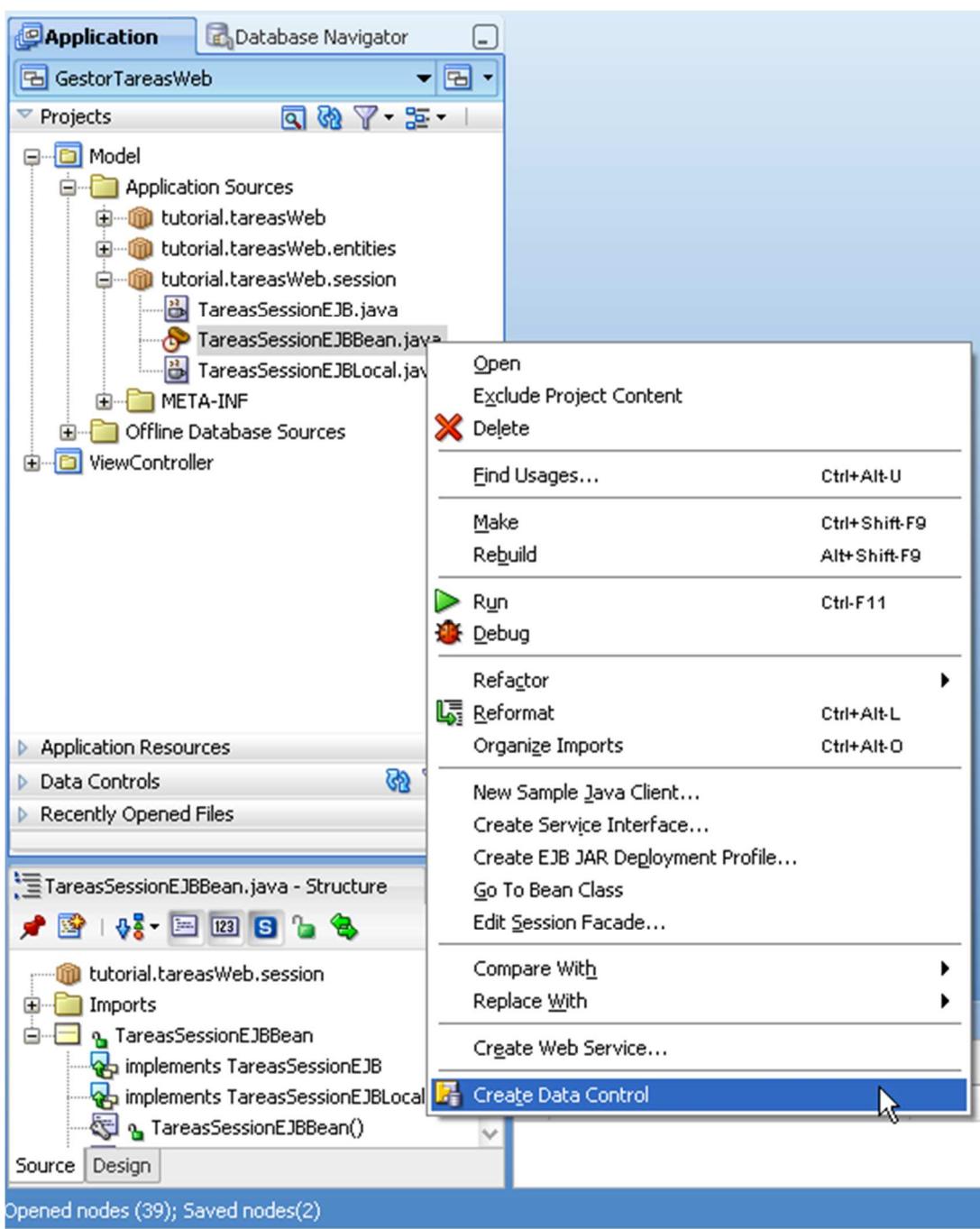


Ilustración 54: Crear Data Control

jDeveloper nos pide entonces que elijamos cuál de las interfaces del EJB, la local o la remota, queremos utilizar para crear el Data Control. El propio entorno nos indica que para trabajar con clientes Web, como JSF, es recomendable utilizar la interfaz local, así que la seleccionamos.

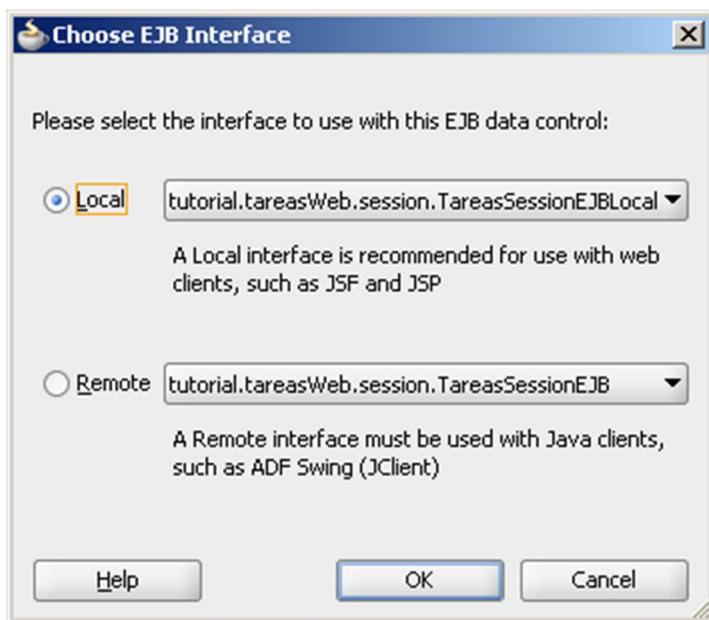


Ilustración 55: Seleccionar la interfaz del EJB

En pocos segundos tendremos el Data Control creado, sin mucho esfuerzo. Pero vamos a detenernos a ver qué es lo que el IDE ha creado para nosotros. Si revisamos el navegador de aplicaciones podemos ver que hay muchos elementos nuevos.

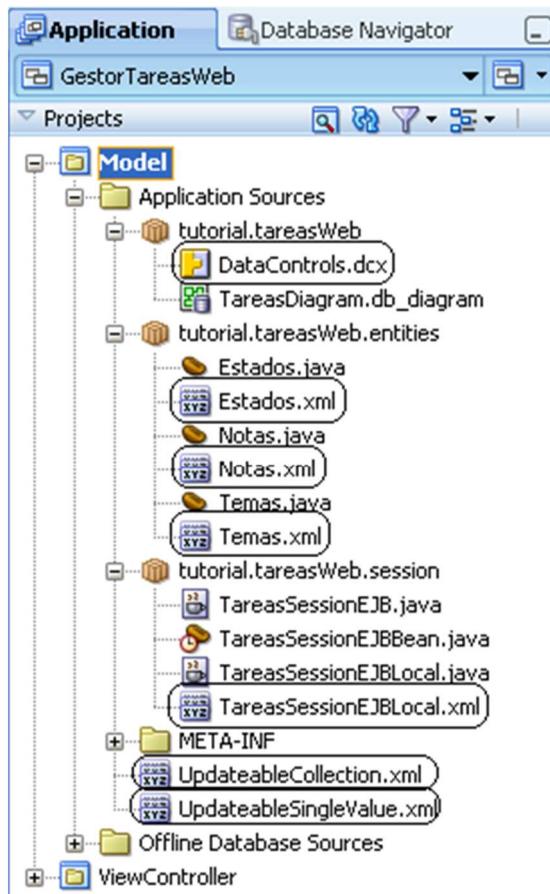


Ilustración 56: Nuevos elementos del proyecto Model

Se ha creado un fichero con extensión xml para cada uno de los beans, tanto de entidad como de sesión, del proyecto, (entidades Temas, Notas y Estados, y bean de sesión TareasSessionEJB), un fichero *DataControls.dcx*, y otros dos xml *UpdateableCollection* y *UpdateableSingleValue*. Vamos a ir viéndolos.

Comenzamos por los xml asociados a los beans de entidad. Estos xml contienen simplemente una descripción de las entidades, con sus atributos y métodos. En ellos hay un único elemento *JavaBean*, con atributos y métodos como hijos.

La etiqueta *JavaBean* sirve para definir el nombre del bean, y la localización de su implementación.

```
<JavaBean xmlns="http://xmlns.oracle.com/adfm/beanmodel" version="11.1.1.56.60"
           id="Temas" Package="tutorial.tareasWeb.entities"
           BeanClass="tutorial.tareasWeb.entities.Temas" isJavaBased="true">
```

Ilustración 57: Cabecera JavaBean

Para cada uno de los atributos de la clase tenemos una entrada; en ellas se indican los nombres y tipos, y se marca también cuál es la clave primaria.

```
<Attribute Name="descripcion" Type="java.lang.String"/>
<Attribute Name="incidencias" Type="java.lang.String"/>
<Attribute Name="sid" PrimaryKey="true" Type="java.lang.Long"/>
<Attribute Name="tema" Type="java.lang.String"/>
```

Ilustración 58: Atributos

Para aquellos atributos que no son de tipos básicos, tendremos una etiqueta *AccessorAttribute*. Los atributos indican su nombre, si es una colección o un único objeto, los métodos para agregar y eliminar elementos si se trata de una colección, y el tipo del objeto (u objetos, en caso de ser una colección).

```
<AccessorAttribute id="notasList" IsCollection="true"
                     RemoveMethod="removeNotas" AddMethod="addNotas"
                     BeanClass="tutorial.tareasWeb.entities.Notas"
                     CollectionBeanClass="UpdateableCollection">
    <Properties>
        <Property Name="RemoveMethod" Value="removeNotas"/>
        <Property Name="AddMethod" Value="addNotas"/>
    </Properties>
</AccessorAttribute>
```

Ilustración 59: Atributo objeto

Esta etiqueta se usa tanto para los atributos como para aquellos métodos que devuelvan un objeto o colección, pero no reciban parámetros.

Por último, tenemos una etiqueta para cada uno de los métodos de la clase, incluidos los constructores. En este caso se usarán las etiquetas *MethodAccessor* o *ConstructorMethod*, con atributos que indicarán si el valor devuelto es una colección, el tipo o tipos de objetos devueltos, los parámetros que reciben con su nombre y tipo, etc.

```
<MethodAccessor IsCollection="false" Type="tutorial.tareasWeb.entities.Notas"
    BeanClass="tutorial.tareasWeb.entities.Notas" id="removeNotas"
    ReturnNodeName="Notas">
    <ParameterInfo id="notas" Type="tutorial.tareasWeb.entities.Notas"
        isStructured="true"/>
</MethodAccessor>
```

Ilustración 60: Métodos

Si pasamos al xml generado para el bean de sesión, veremos que es exactamente igual. Tenemos un único elemento *JavaBean*, que se corresponde con nuestro EJB, y se listan los métodos de la clase. Por tanto, no vamos a detenernos más.

Respecto a los ficheros *UpdateableCollection* y *UpdateableSingleValue*, son usados por jDeveloper en tiempo de diseño para especificar la lista de operaciones que la paleta de *Data Controls*, que veremos ahora, debe mostrar. Podemos ver que son referidas en las etiquetas *AccessorAttributes* de la definición de los *JavaBeans*.

Nos queda el fichero *DataControls.dcx*, en el que se van a definir los controles de datos que estarán accesibles en el proyecto.

En este fichero tendremos una entrada de tipo *AdapterDataControl* para cada uno de los servicios de negocio sobre los que hayamos definido un *DataControl*, cuyo contenido variará según el tipo de servicio de negocio. Dentro puede haber una etiqueta *CreatableTypes*, en la que se definen aquellos tipos para los que se puede invocar un constructor. En nuestro caso, las entidades con las que trabajamos. Por último tendremos la etiqueta *Source*, en la que se define el servicio sobre el que se ha creado el *Data Control*. Al tratarse de un *Data Control* creado sobre un EJB de sesión tenemos dentro el elemento *ejb-definition*, en el que se especifican los datos del EJB como nombre, versión, tipo, o interfaz.

```
<?xml version="1.0" encoding="UTF-8" ?>
<DataControlConfigs xmlns="http://xmlns.oracle.com/adfm/configuration"
    version="11.1.1.56.60" id="DataControls"
    Package="tutorial.tareasWeb">
    <AdapterDataControl id="TareasSessionEJBLocal"
        FactoryClass="oracle.adf.model.adapter.DataControlFactoryImpl"
        ImplDef="oracle.adfinternal.model.adapter.ejb.EjbDefinition"
        SupportsTransactions="false" SupportsSortCollection="true"
        SupportsResetState="false" SupportsRangeSize="false"
        SupportsFindMode="false" SupportsUpdates="true"
        Definition="tutorial.tareasWeb.session.TareasSessionEJBLocal"
        BeanClass="tutorial.tareasWeb.session.TareasSessionEJBLocal"
        xmlns="http://xmlns.oracle.com/adfm/datacontrol">
        <CreatableTypes>
            <TypeInfo FullName="tutorial.tareasWeb.entities.Notas"/>
            <TypeInfo FullName="tutorial.tareasWeb.entities.Temas"/>
            <TypeInfo FullName="tutorial.tareasWeb.entities.Estados"/>
        </CreatableTypes>
        <Source>
            <ejb-definition ejb-version="3.0" ejb-name="TareasSessionEJB"
                ejb-type="Session"
                ejb-business-interface="tutorial.tareasWeb.session.TareasSessionEJBLocal"
                ejb-interface-type="local"
                initial-context-factory="weblogic.jndi.WLInitialContextFactory"
                DataControlHandler="oracle.adf.model.adapter.bean.jpa.JPQLDataFilterHandler"
                xmlns="http://xmlns.oracle.com/adfm/adapter/ejb"/>
        </Source>
    </AdapterDataControl>
</DataControlConfigs>
```

Ilustración 61: Fichero DataControls

Otro fichero en el que podemos fijarnos ahora es el *adfm.xml*, dentro del *meta-inf* del proyecto. En principio no tiene ninguna importancia para nuestro desarrollo, y podemos obviarlo, pero como siempre es bueno saber por qué están ahí las cosas, el adfm no es más que un registro que lleva jDeveloper de los *Data Controls* que se han ido creando. Gracias a él, el IDE puede localizarlos con facilidad para incluirlos en la paleta de *Data Controls*.

Retomando nuestro desarrollo, acabamos de crear el Data Control para nuestro EJB de sesión, y llega el momento de utilizarlo desde nuestra página Web. De nuevo el uso puede resultar muy sencillo gracias al IDE, pero entraremos en detalle para tratar de entender cómo funciona todo por detrás.

Ahora podemos acceder a una vista que no habíamos utilizado, aunque sí mencionado, hasta el momento: la paleta de *Data Controls*. Esta vista se encuentra dentro del navegador de la aplicación, y estará replegada al no haber sido usada todavía. Para agrandarla no tenemos más que hacer clic sobre ella.

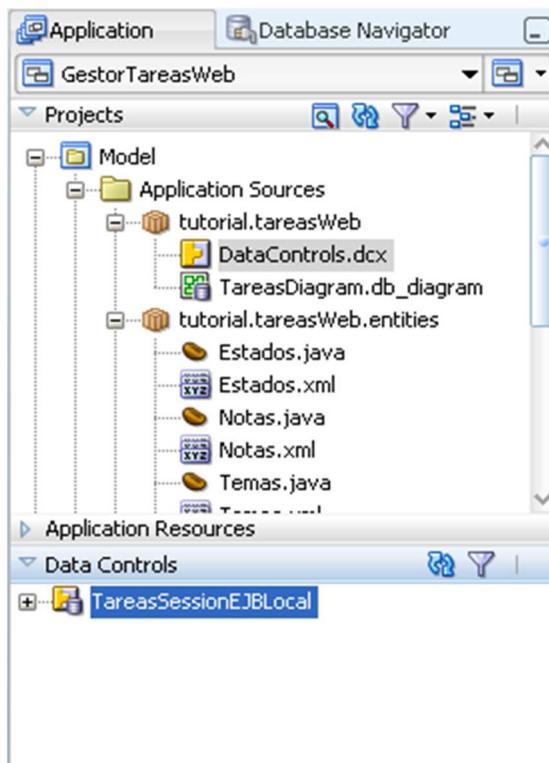


Ilustración 62: Paleta DataControls

Como era de esperar, nos encontramos con un único *DataControl*, que es el que acabamos de crear, *TareasSessionEJBLocal* (el id que tiene en su definición dentro del *DataControls.dcx*). Si desplegamos a su vez este nodo, vemos que tenemos un nodo para cada uno de los métodos definidos en el fichero *TareasSessionEJBLocal.xml*, así como otro adicional para los constructores, que se especificaron con los *CreatableTypes* del *DataControls.dcx*. A continuación podemos ver la paleta de *Data Controls*, frente a la estructura del *TareasSessionEJBLocal.xml*.

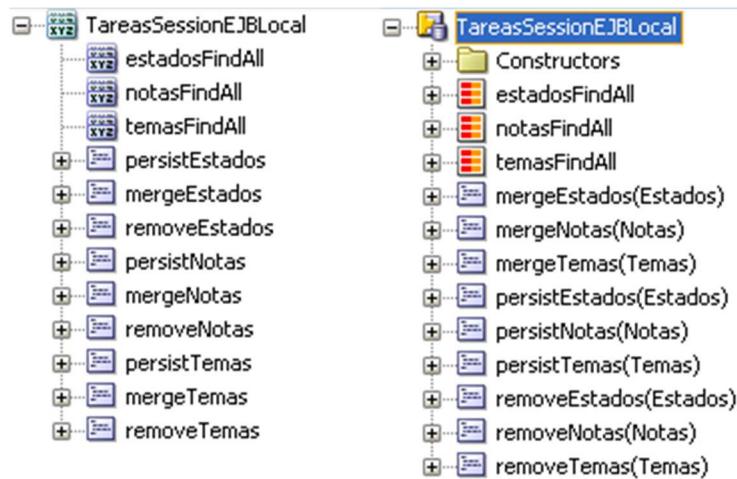


Ilustración 63: Estructura EJB frente a DataControls

Si recordamos, nuestro objetivo era mostrar en la página que estábamos creando, `consulta.jspx`, la lista de temas registrados en el Sistema. Una vez que tenemos el *Data Control*, es tan sencillo como arrastrar la operación adecuada (`temasFindAll`, en nuestro caso), al lugar en que queramos que se muestre de la página. Por tanto, seleccionamos `temasFindAll` en la paleta de *Data Controls*, y arrastramos hacia el lado izquierdo del panel dividido de nuestra página, marcado como *first*. Al hacerlo se mostrará un menú para seleccionar el componente que queremos crear, y elegiremos, dentro del menú *Table*, la opción *ADF read only table*. Tras eso se presentará un wizard para que podamos personalizar la tabla, indicando las columnas a mostrar, si queremos que se pueda seleccionar, etc. Eliminamos las columnas `sid`, tanto del tema como de su estado, y reordenamos el resto, obteniendo el siguiente resultado.

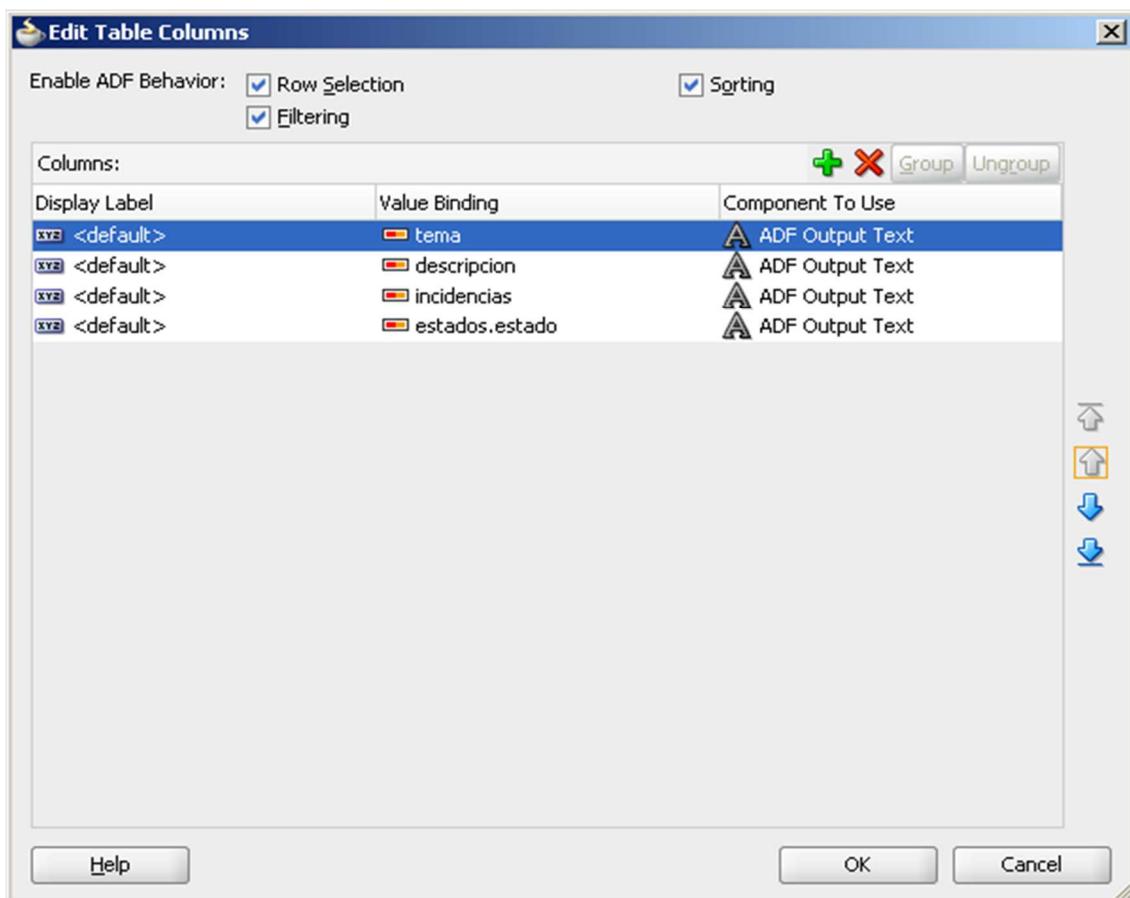


Ilustración 64: Editar columnas de la tabla

Como se ve, podemos pedirle al IDE que cree una tabla que incluya automáticamente las capacidades de selección, ordenación y filtrado, sin necesidad de ningún tipo de esfuerzo por nuestra parte.

En este punto hemos utilizado por primera vez un *Data Control* en nuestro *ViewController*, y eso implica que, como podemos ver en el navegador de aplicaciones, se han creado una serie de ficheros que pasamos a detallar.

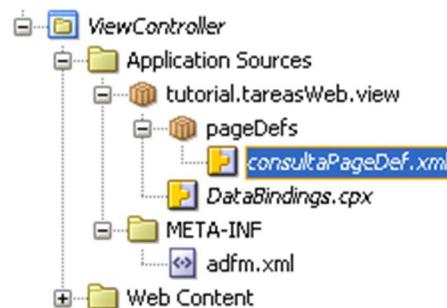


Ilustración 65: Navegador del proyecto ViewController

El primero es el *DataBindings.cpx*. Este fichero define el *Binding Context* de la aplicación, y proporciona metadatos a partir de los cuales los *bindings* de ADF son creados en tiempo de ejecución.

El *Binding Context* es un objeto que define un espacio de nombres común para ser usado por el cliente de la aplicación, y permite exponer los objetos del modelo a través de un nombre identificativo.

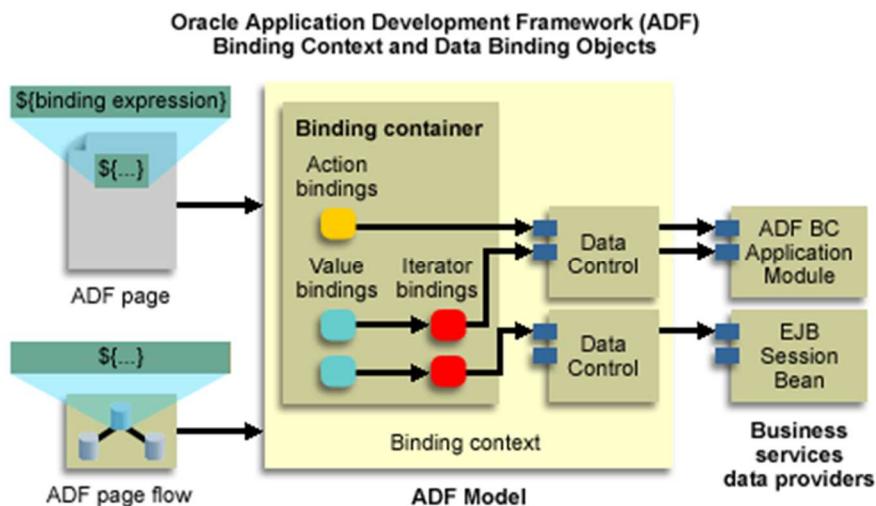


Ilustración 66: BindingContext y DataBindings

Por su parte, el *Binding Container* contendrá enlaces de datos (*Data Bindings*), que pueden ser *Action*, *Value* e *Iterator Binding*. Es decir, enlaces hacia un método, valor o colección de objetos del modelo. Para cada página Web de nuestra aplicación se creará un *Binding Container*.

Para entenderlo, lo mejor es verlo. El fichero se divide en tres secciones:

- *PageMap*.- en esta sección se asocian las páginas con sus ficheros de definición (*PageDefinition*), a través de los *PageDefinitionUsage*. Los ficheros de definición los vamos a explicar a continuación.
- *PageDefinitionUsages*.- indican la ruta de los *PageDefinitionUsage*.
- *DataControlUsages*.- contiene la lista de *Data Controls* con la ruta en la que se localiza cada uno de ellos.

Por otro lado tenemos los ficheros de definición de página, a los que acabamos de hacer referencia, y que también han sido generados por el IDE. En nuestro caso podemos ver el fichero *consultaPageDef.xml*, que sería el fichero de definición de la página *consulta.jspx*. Esta relación la podemos ver en el *DataBindings*.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Application xmlns="http://xmlns.oracle.com/adfm/application"
    version="11.1.1.56.60" id="DataBindings" SeparateXMLFiles="false"
    Package="tutorial.tareasWeb.view" ClientType="Generic">
    <pageMap>
        <page path="/consulta.jspx"
            usageId="tutorial_tareasWeb_view_consultaPageDef"/>
    </pageMap>
    <pageDefinitionUsages>
        <page id="tutorial_tareasWeb_view_consultaPageDef"
            path="tutorial.tareasWeb.view.pageDefs.consultaPageDef"/>
    </pageDefinitionUsages>
    <dataControlUsages>
        <dc id="TareasSessionEJBLocal"
            path="tutorial.tareasWeb.TareasSessionEJBLocal"/>
    </dataControlUsages>
</Application>
```

Ilustración 67: Fichero DataBindings

Los ficheros de definición de página también son generados automáticamente por el IDE, aunque tenemos la posibilidad de modificarlos siempre que queramos. Se componen de tres secciones.

- Parameters.- define parámetros que son accesibles a nivel de página.
- Executables.- define los elementos que deben ser ejecutados durante la fase de preparación del modelo de la página. No vamos a entrar en el ciclo de vida de ADF, pero básicamente estos métodos se ejecutarán durante la carga de la página.
- Bindings.- en esta sección se hace referencia a los ejecutables definidos, para indicar cómo se accederá a nivel de atributos.

Si vemos nuestro caso concreto, tenemos:

```
<executables>
    <variableIterator id="variables"/>
    <iterator Binds="root" RangeSize="25" DataControl="TareasSessionEJBLocal"
               id="TareasSessionEJBLocalIterator"/>
    <accessorIterator MasterBinding="TareasSessionEJBLocalIterator"
                      Binds="temasFindAll" RangeSize="25"
                      DataControl="TareasSessionEJBLocal"
                      BeanClass="tutorial.tareasWeb.entities.Temas"
                      id="temasFindAllIterator"/>
    <searchRegion Binds="temasFindAllIterator" Criteria=""
                   Customizer="oracle.jbo.uicli.binding.JUSearchBindingCustomizer"
                   id="temasFindAllQuery"/>
</executables>
```

Ilustración 68: Sección de ejecutables

Vemos que se define en primer lugar un *variableIterator*, que no es más que un iterador que contendrá todas las variables declaradas para el *binding container*.

A continuación tenemos un iterador, definido sobre nuestro único *Data Control*, al que se ha llamado *TareasSessionEJBLocalIterator*. Este elemento será el padre del siguiente, un *accessorIterator* de nombre *temasFindAllIterator*, que enlaza con el método *temasFindAll*, del *DataControl TareasSessionEJBLocal*, y que contiene objetos de tipo *Temas*. Resumiendo, acabamos de declarar un iterador que en ejecución contendrá el resultado de la llamada al método que obtiene todas las asignaturas registradas.

En la sección de bindings encontramos:

```
<bindings>
    <tree IterBinding="temasFindAllIterator" id="temasFindAll">
        <nodeDefinition DefName="tutorial.tareasWeb.entities.Temas">
            <AttrNames>
                <Item Value="tema"/>
                <Item Value="descripcion"/>
                <Item Value="incidencias"/>
            </AttrNames>
            <Accessors>
                <Item Value="estados"/>
            </Accessors>
        </nodeDefinition>
        <nodeDefinition DefName="tutorial.tareasWeb.entities.Estados">
            <AttrNames>
                <Item Value="estado"/>
                <Item Value="sid"/>
            </AttrNames>
        </nodeDefinition>
    </tree>
</bindings>
```

Ilustración 69: Sección de Bindings

Como vemos, tenemos un único *binding*, bajo la etiqueta *tree*. Esta etiqueta es equivalente a *table*, y especifica que vamos a tener una tabla de nombre *temasFindAll*,

definida sobre el iterador *temasFindAllIterator*. Esto significa que nuestra tabla mostrará en cada momento los elementos de este iterador, que a su vez contendrá el resultado que se obtenga de llamar al método *temasFindAll*.

Los nodos son de tipo *Temas* y *Estados*, y los atributos son los de estas clases (se muestran los datos del *tema*, y el campo *estado* que corresponde a la entidad *estados*).

Una vez revisados estos ficheros nos falta ver cómo se usan los elementos que hemos definido desde la página. Si accedemos al código de *consulta.jspx*, podemos ver el siguiente código correspondiente a la tabla que hemos agregado (para acceder a él de forma sencilla podemos seleccionar la tabla en la vista de diseño o de estructura, y seleccionar la opción *Go to source* haciendo clic derecho).

```
<af:table value="#{bindings.temasFindAll.collectionModel}"
    var="row"
    rows="#{bindings.temasFindAll.rangeSize}"
    emptyText="#{bindings.temasFindAll.viewable ? 'No data to display.' : 'Access Denied.'}"
    fetchSize="#{bindings.temasFindAll.rangeSize}"
    rowBandingInterval="0" rowSelection="single" id="tl"
    filterModel="#{bindings.temasFindAllQuery.queryDescriptor}"
    queryListener="#{bindings.temasFindAllQuery.processQuery}"
    filterVisible="true" varStatus="vs"
    selectedRowKeys="#{bindings.temasFindAll.collectionModel.selectedRow}"
    selectionListener="#{bindings.temasFindAll.collectionModel.makeCurrent}">
<af:column sortProperty="tema" filterable="true"
    sortable="true" id="c1"
    headerText="#{bindings.temasFindAll.hints.tema.label}">
    <af:outputText value="#{row.tema}" id="ot1"/>
</af:column>
<af:column sortProperty="descripcion"
    filterable="true" sortable="true" id="c3"
    headerText="#{bindings.temasFindAll.hints.descripcion.label}">
    <af:outputText value="#{row.descripcion}" id="ot2"/>
</af:column>
<af:column sortProperty="incidencias"
    filterable="true" sortable="true" id="c2"
    headerText="#{bindings.temasFindAll.hints.incidencias.label}">
    <af:outputText value="#{row.incidencias}" id="ot4"/>
</af:column>
<af:column sortProperty="estado" filterable="true"
    sortable="true" id="c4"
    headerText="#{bindings.temasFindAll.hints.estados.estado.label}">
    <af:outputText value="#{row.estados.bindings.estado.inputValue}"
        id="ot3"/>
</af:column>
</af:table>
```

Ilustración 70: Código fuente de un tabla

Lo que se ha añadido a la página es el componente *table* de *ADF Faces*. En el atributo *value* vemos que se está accediendo al elemento *tree* que acabamos de ver definido en el *pageDef*, con nombre *temasFindAll*.

En este punto, para aclarar conceptos, sería interesante ir siguiendo la pista desde esta definición, para ver cómo se acaba llegando hasta el método de nuestro EJB que devuelve la información deseada.

Respecto a la definición de la tabla, podemos ver que hay un texto para mostrar en caso de que no haya asignaturas (*emptyText*), que se especifica el número de filas que se va recuperando cada vez (*fetchSize*), etc., pero no vamos a detenernos más.

A partir de aquí vamos a ir trabajando de forma más ágil, y dejaremos que el IDE vaya modificando los ficheros que hemos visto para que todo funcione correctamente. Todo lo que iremos haciendo está basado en la misma idea que acabamos de explicar.

Ya tenemos el listado de temas, y queremos que cada vez que el usuario seleccione uno se muestre el listado de notas asociadas a mismo. Esta información la vamos a mostrar en el lado derecho de nuestro panel y, para obtenerla, recurrimos de nuevo a nuestro *Data Control*.

Si desplegamos el elemento *temasFindAll*, veremos los atributos del tipo *Temas*, entre los que se encuentra su lista de *notas*. Si desplegamos también este elemento, veremos los atributos de una *Nota*.

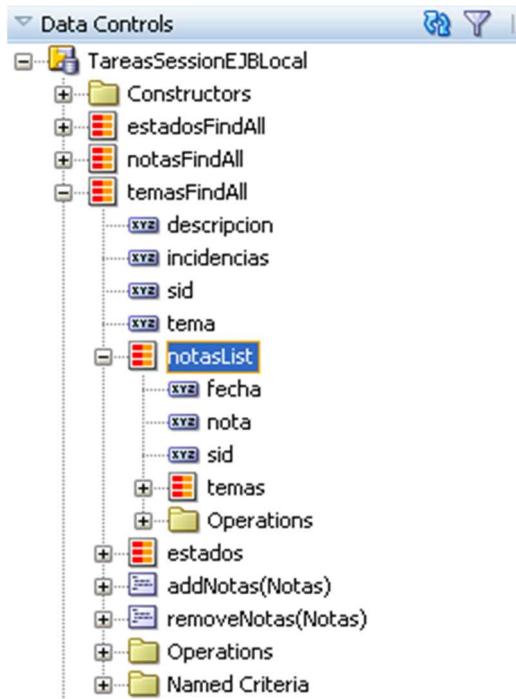


Ilustración 71: DataControls

Arrastraremos el elemento *notasList* hacia la sección *second* de la página *consulta.jspx*. Nuevamente veremos una menú contextual en el que elegiremos la opción *Table – ADF Read-Only Table*, abriéndose el diálogo para crear la tabla. Esta vez tendremos disponibles todos los datos de la clase *Notas* (*sid*, *nota* y *fecha*), y *Temas* (*sid*, *tema*, *descripcion* e *incidencias*). Eliminaremos de la lista todos los atributos excepto los de la clase *Notas*, y seleccionaremos los checks para que la tabla tenga las opciones de selección, búsqueda y ordenación.

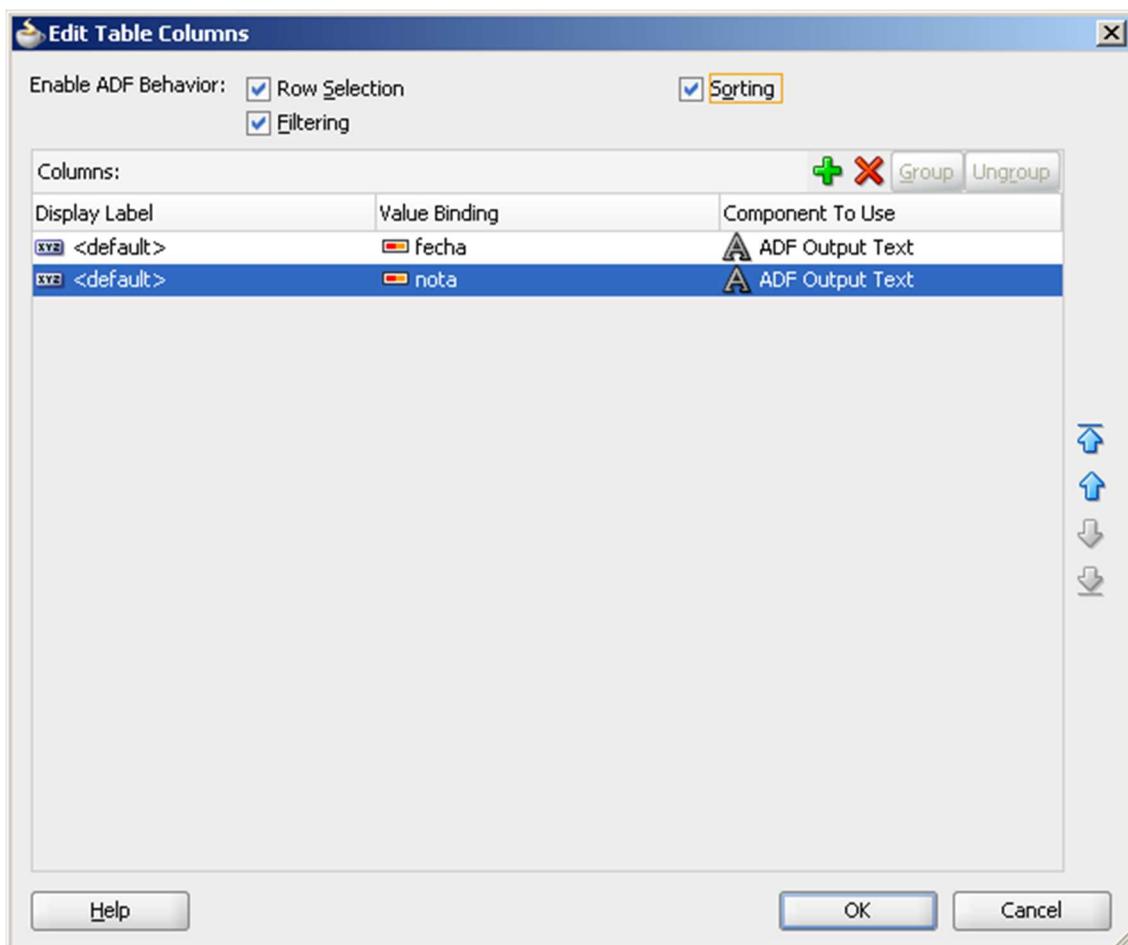


Ilustración 72: Editar columnas de la tabla

Al haber hecho esto seleccionando la lista de notas debajo de la de asignaturas, conseguimos que nuestra página muestre el listado de notas relacionadas con el tema seleccionado actualmente.

Sin embargo, para que todo funcione correctamente, tenemos que hacer algo más. Queremos que cada vez que se cambie la fila seleccionada en la tabla padre, se repinte la tabla hija, mostrando los registros relacionados. Este comportamiento lo obtenemos utilizando el atributo *partialTriggers* de la tabla, mediante el que podemos decirle a un componente que debe actualizarse cada vez que se produzca un cambio en otros. Para nuestro caso, le decimos a la tabla *t2* que se actualice cuando haya cambios en la tabla *t1*. Lo indicamos dando el valor *::t1* a la propiedad *partialTriggers* de *t2* (lo podemos encontrar en la categoría *Behavior*).

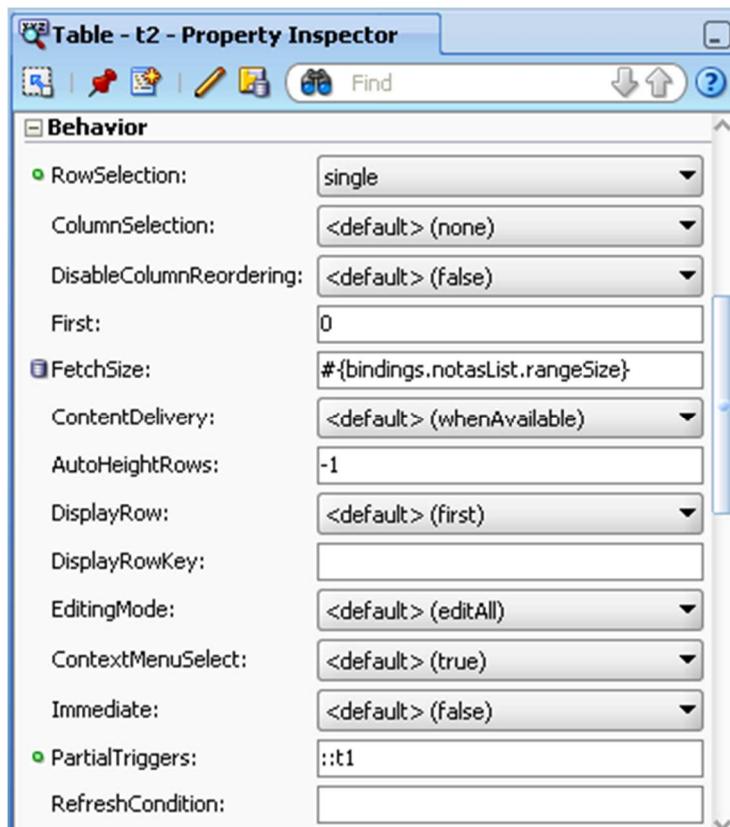


Ilustración 73: Propiedades de la tabla

Por el momento hemos terminado con la página de consulta. Vamos a pasar ahora a desarrollar una nueva interfaz para crear temas. Volvemos al fichero *faces-config.xml*, y arrastramos una página *jspx* al diagrama. La iniciamos siguiendo exactamente los mismos pasos que al crear *consulta.jspx*, cambiando en este caso el nombre por *nuevoTema.jspx*, hasta llegar al editor.

Para esta página no vamos a usar ningún panel de layout, como hicimos en la de consulta con el *panelStretchLayout*, sino que agregaremos directamente los controles que necesitamos.

Queremos tener un simple formulario para poder introducir los datos de un nuevo tema. Accedemos al Data Control y arrastramos el iterador *temasfindAll*. Se muestran todos los componentes que podemos crear, y elegimos, dentro del grupo *Form*, la opción *ADF Form*. Así pasamos a un nuevo cuadro de diálogo en el que podemos configurar el formulario.

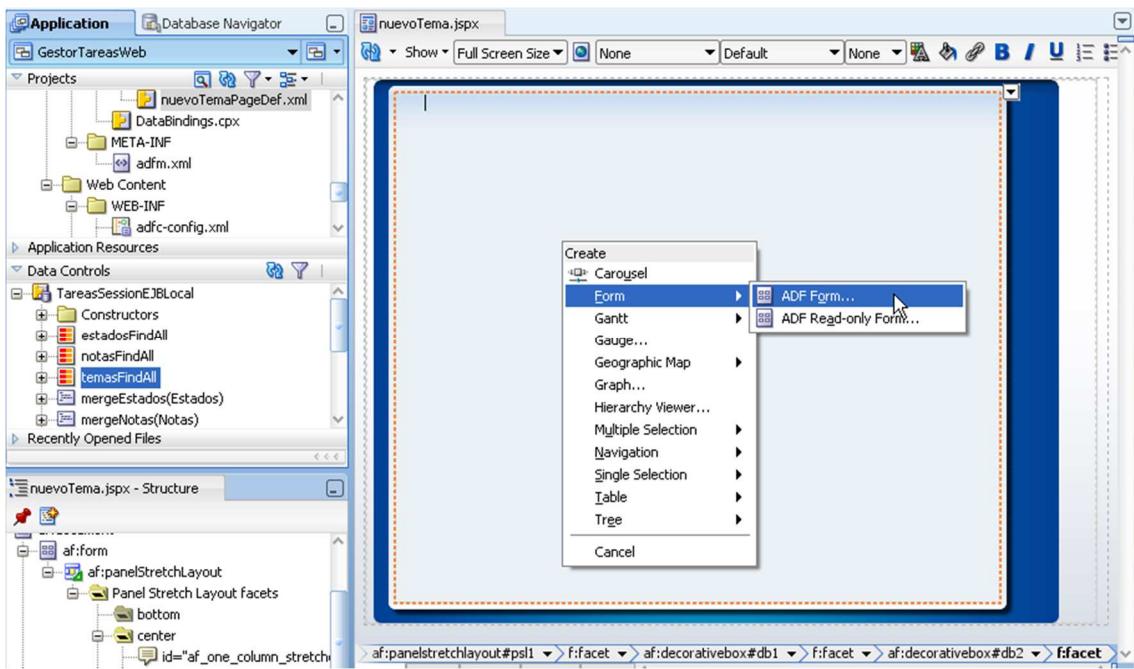


Ilustración 74: Crear formulario ADF

En el diálogo *Edit form fields*, tenemos que seleccionar los campos que queremos que se muestren en el formulario. Vamos a dejar la descripción, las incidencias, el tema, y el campo sid del estado. En la lista aparecen dos campos nombrados sid, uno correspondiente a los temas, y otro a los estados. No hay forma de diferenciarlos por la información que se muestra, pero el de estado será siempre el que aparezca en segundo lugar.

Abajo se nos da la posibilidad de crear botones de navegación y envío del formulario. Los de navegación no nos interesan ya que queremos que en esta página se muestre sólo el tema que estamos creando. El de envío sí lo seleccionamos, y aceptamos el diálogo.

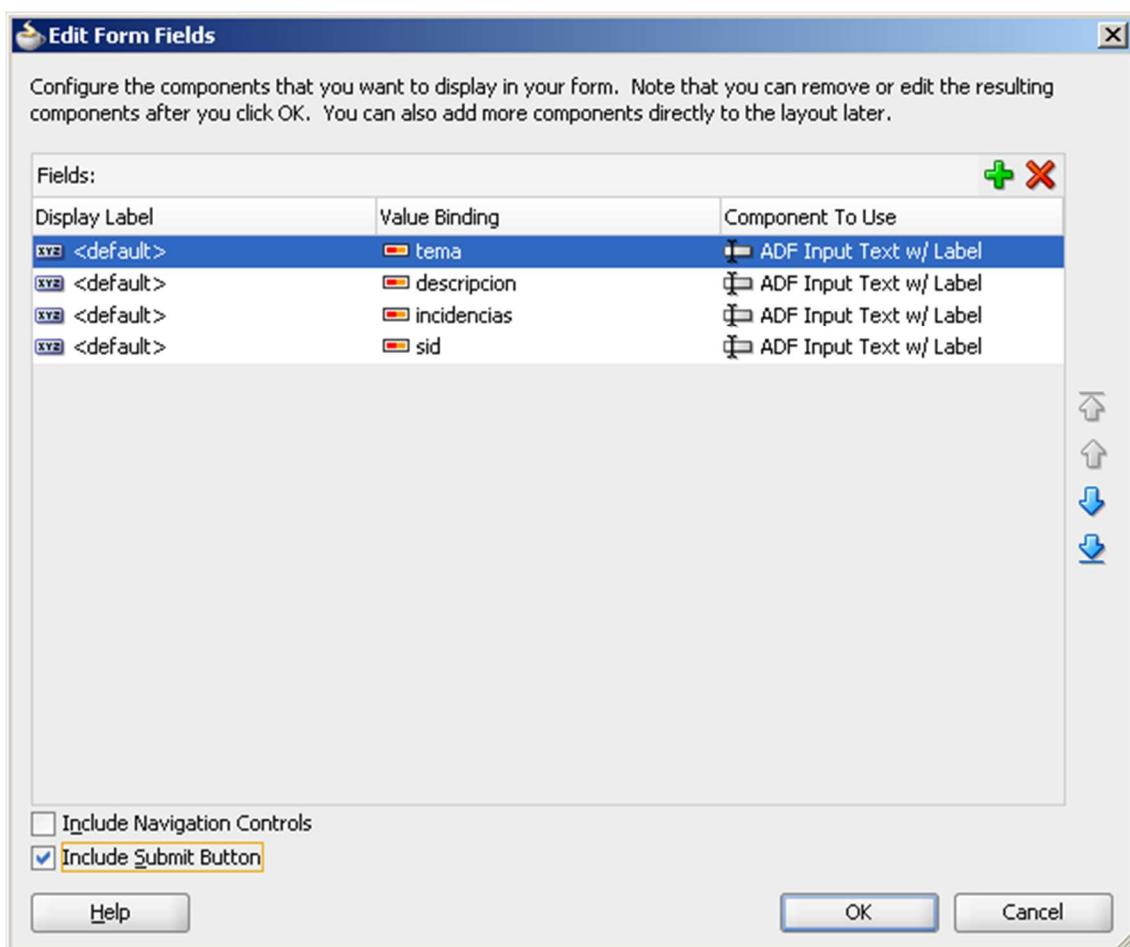


Ilustración 75: Editar campos de formulario

Vemos que se han creado los controles que hemos seleccionado como cuadros de texto. Si analizamos alguno de ellos, podemos ver que la mayoría de sus propiedades han sido enlazadas mediante un *binding*. Podemos hacer clic derecho sobre alguno y elegir *Go to binding*; se mostrará de forma gráfica el enlace de datos creado. Para cada propiedad añadida se ha creado un enlace de tipo *attributeValues*.

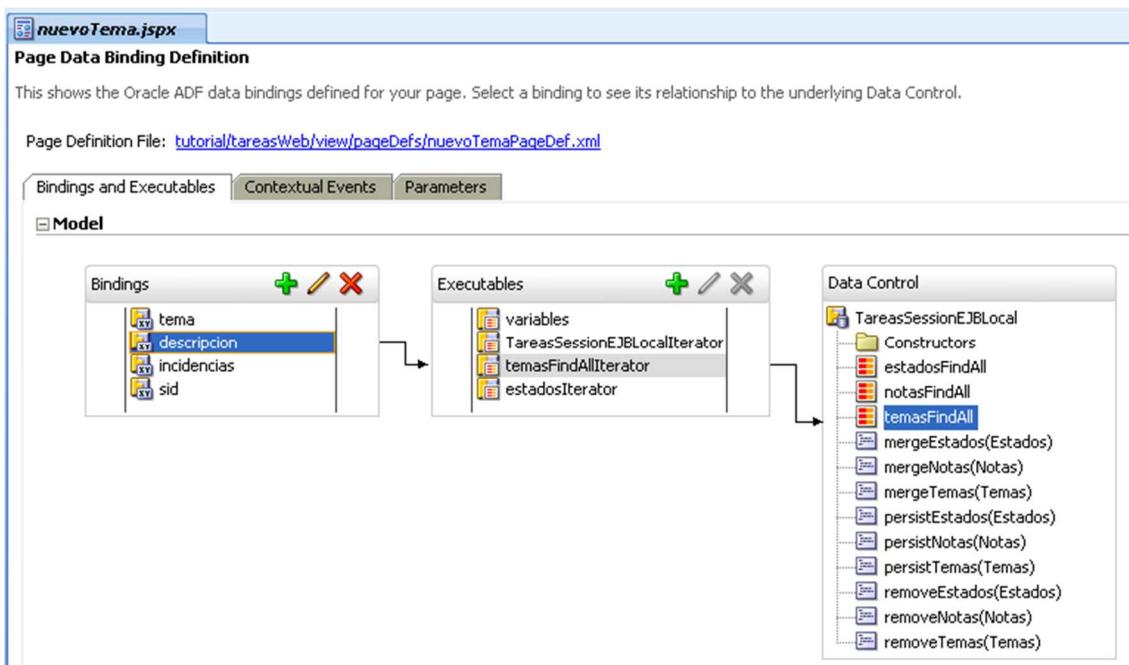


Ilustración 76: Enlace de datos de página

Para ver el código fuente del binding tenemos que abrir el fichero *nuevoTemaPageDef.xml* (podemos hacerlo con botón derecho sobre la página jspx, y eligiendo *Go to Page Definition*).

En este fichero tenemos la definición de los cuatro atributos, y de un elemento *accessorIterator*, que crea un iterador sobre la colección devuelta por la invocación al método *temasFindAll* de nuestro EJB. Al estar trabajando con una colección, los atributos (descripcion, incidencias, tema y sid) obtienen los valores del elemento actual de la misma. Esto implica que si accediésemos directamente a esta página nada más arrancar la aplicación, los valores que se mostrarían serían los del primer tema de la colección.

Sin embargo, nosotros queremos que los campos estén vacíos, para crear un nuevo tema desde cero. Para ello, crearemos un nuevo tema en la colección, y lo marcaremos como el elemento seleccionado. De esta forma los controles mostrarán los datos del tema, que al estar recién creado, estarán vacíos.

Esta operación la realizaremos al pulsar el botón que nos envíe a la página *nuevoTema*. Nos vamos a *consulta.jspx*, y vamos a crear una opción de menú que nos dirija a ella. Para este primer caso, aprovechamos el elemento que creamos anteriormente pero dejamos sin información.

Si seleccionamos el menú, en la vista de propiedades podemos cambiar tanto su *Id*, como el campo *Text*, al que pondremos *Temas*. Después, desde la vista de estructura, podemos desplegar el menú, y seleccionar el *MenuItem* que creamos bajo él. En las propiedades modificamos de nuevo el *Id* y el *Text*, al que pondremos *Nuevo Tema*.



Ilustración 77: Menú en vista estructura

Ahora tenemos que hacer que al pulsar esta opción del menú nuestra aplicación navegue hacia la página *nuevoTema*. Las reglas de navegación se definen en el fichero *faces-config.xml*. Se pueden crear en la vista diagrama, uniendo páginas jspx, o a través de la vista Overview, en la sección *Navigation Rules*.

En primer lugar tenemos que añadir un elemento a la tabla *From Views*. Aquí iremos metiendo cada una de las páginas de la aplicación para crear reglas de navegación que parten de ellas. Añadimos un elemento, y elegimos la vista *consulta.jspx*.

Ahora podemos crear elementos también en la tabla *Navigation Cases*, que tendrá cada una de las reglas de navegación. Tenemos que indicar la vista hacia la que se navega (*To View Id*), y el nombre de la regla (*From Outcome*). La llamaremos *nuevoTema*.

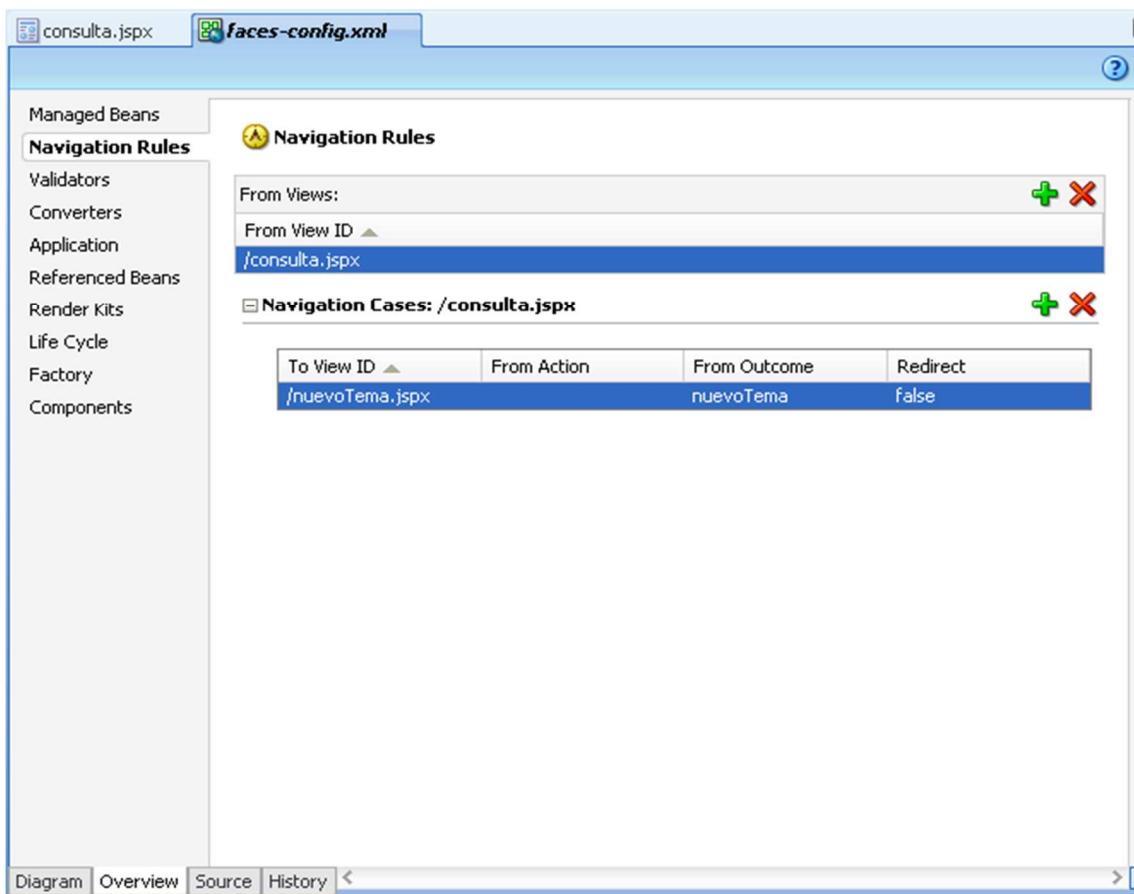


Ilustración 78: Crear regla de navegación

Para crear reglas de navegación globales añadimos un elemento a la tabla de vistas sin seleccionar una página concreta. Esto lo haremos más adelante.

Una vez definida la regla, volvemos a las propiedades del elemento de menú *nuevoTema*. En la propiedad *Action* tenemos un desplegable en el que se mostrarán todas las reglas disponibles desde la página. Seleccionamos la que hemos creado.

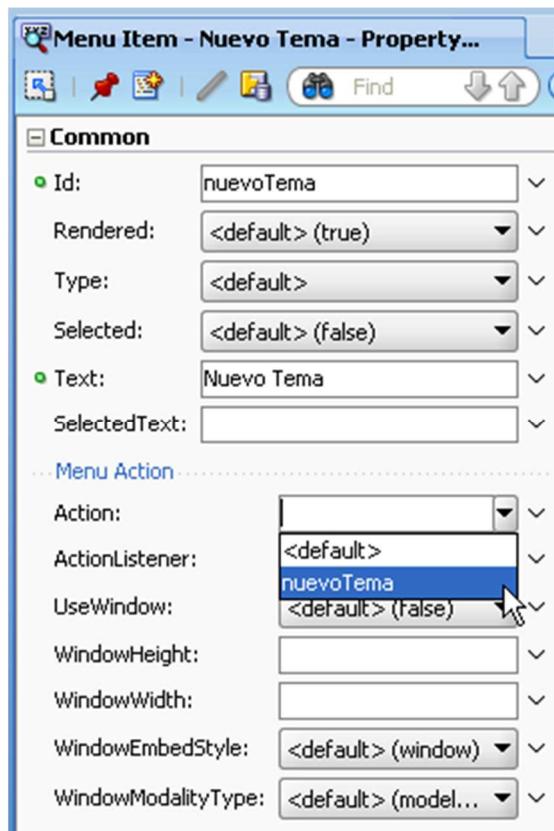


Ilustración 79: Asignar regla de navegación

Por último tenemos que hacer que al seleccionar este menú se cree un nuevo elemento en la colección de temas, como explicamos antes. Si expandimos el nodo `temasFindAll` del DataControl, y dentro de él el grupo de operaciones, veremos la acción `Create`. Este tipo de elementos del DataControl podemos arrastrarlos sobre los botones de nuestras páginas (o de la vista estructura) para enlazarlos. Arrastramos sobre el elemento de menú, y se abrirá un cuadro de diálogo en el que se informa de los cambios que se van a hacer en las propiedades del botón, para darnos la opción de mantener las que nos interesen. En este caso queremos mantener todos los valores salvo `actionListener` y `disabled`.

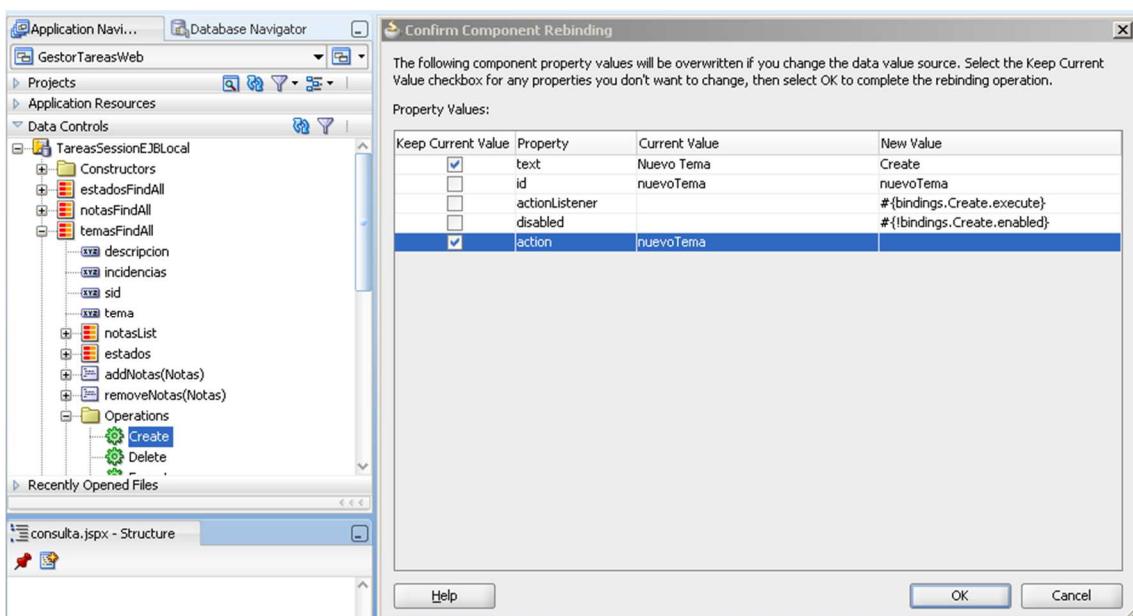


Ilustración 80: Confirmar enlace de componentes

Con esto hemos completado el elemento de menú, que nos permitirá crear un nuevo elemento para editarlo en la página *nuevoTema*, y navegar hacia ella.

En *nuevoTema.jspx* todavía nos queda bastante trabajo que hacer. En primer lugar, vamos a añadir un botón para poder cancelar la creación del tema. En este caso crearemos el componente desde la paleta, en lugar de usar el DataControl. Arrastramos un *Button* junto al botón Submit, que ya existe.

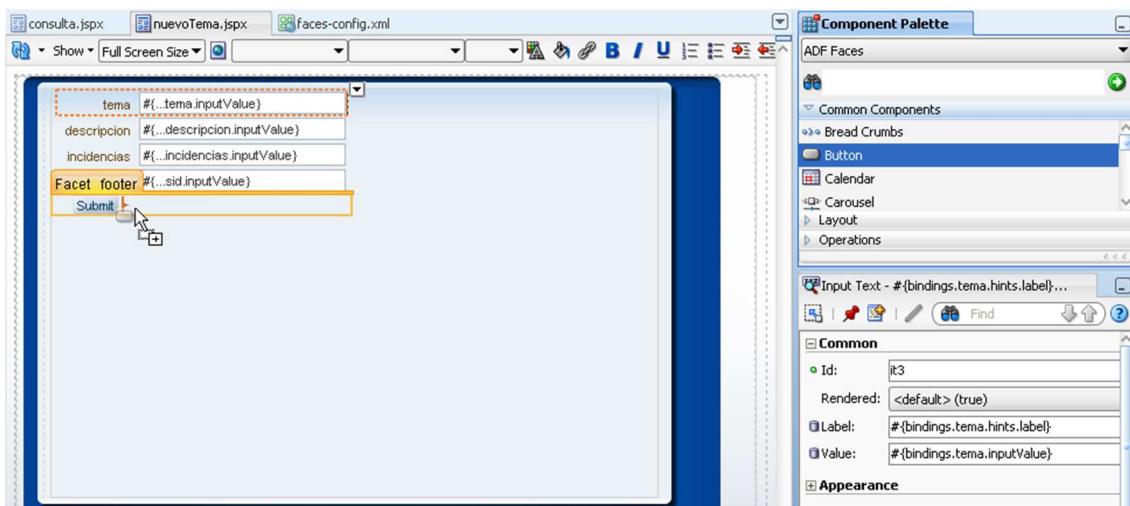


Ilustración 81: Arrastrar botón

Cuando se pulse este botón querremos volver a la pantalla de consulta, por lo que vamos a crear una nueva regla de navegación. En este caso haremos una regla global, dado que más adelante crearemos otras pantallas desde las que se navegará hacia la de consulta, que es el punto central de la aplicación. Seguimos los pasos equivalentes a los que hicimos antes para obtener este resultado.

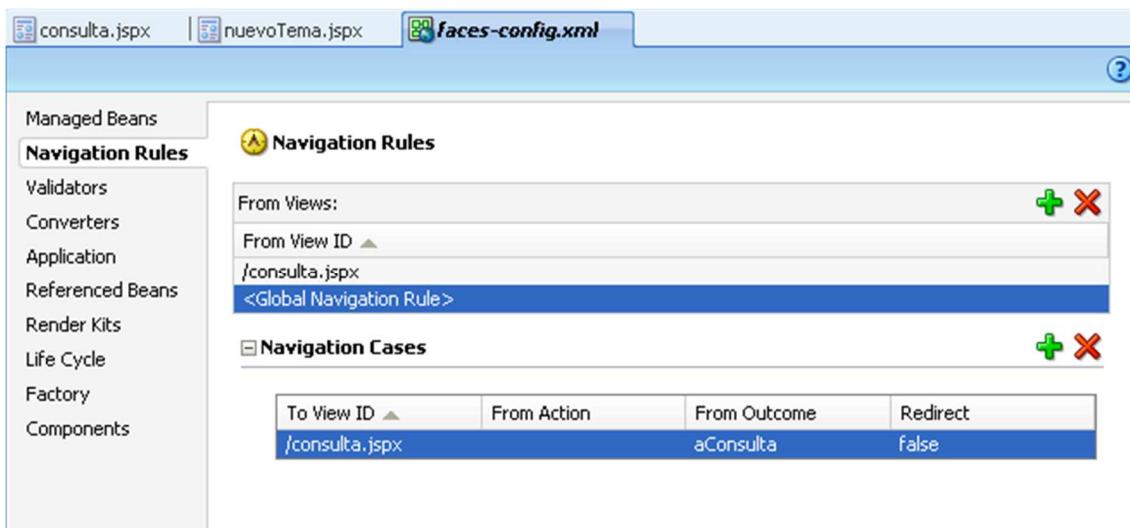


Ilustración 82: Regla de navegación global

Ahora podemos ir a las propiedades del botón y seleccionar la acción, además de cambiar su texto.

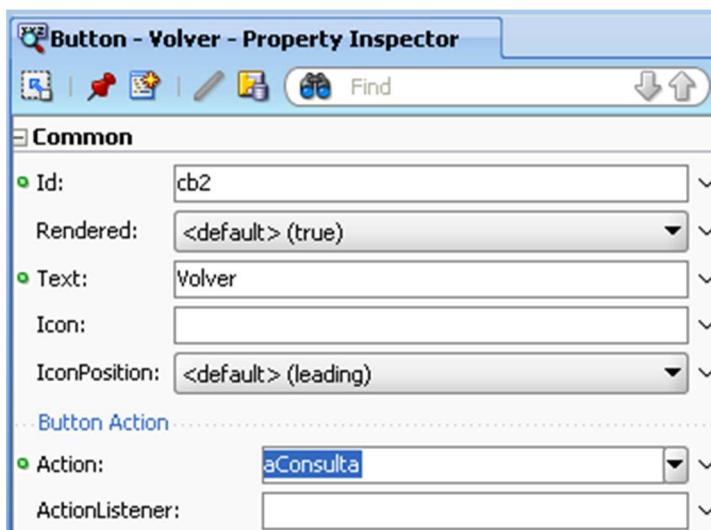


Ilustración 83: Asignar regla de navegación

A la hora de navegar hacia la página *nuevoTema*, creamos previamente un nuevo elemento en la colección de temas. Si el usuario pulsa el botón cancelar y regresa a la página de consulta en lugar de guardar los datos, ese elemento debería volver a quitarse de la colección; en caso contrario aparecerá en el listado de temas que se muestra en la página de consulta. Como este nuevo elemento no ha sido persistido, lo que haremos para eliminarlo es decirle a la aplicación que vuelva a cargar la lista de temas de la base de datos, ejecutando la acción *Execute* sobre la colección. Por tanto, en el *DataControl* desplegamos las operaciones de *TemasFindAll*, y arrastramos *Execute* sobre el botón. En el diálogo de cambio de propiedades seleccionamos las que queremos mantener, que son todas menos *disabled* y *ActionListener*.

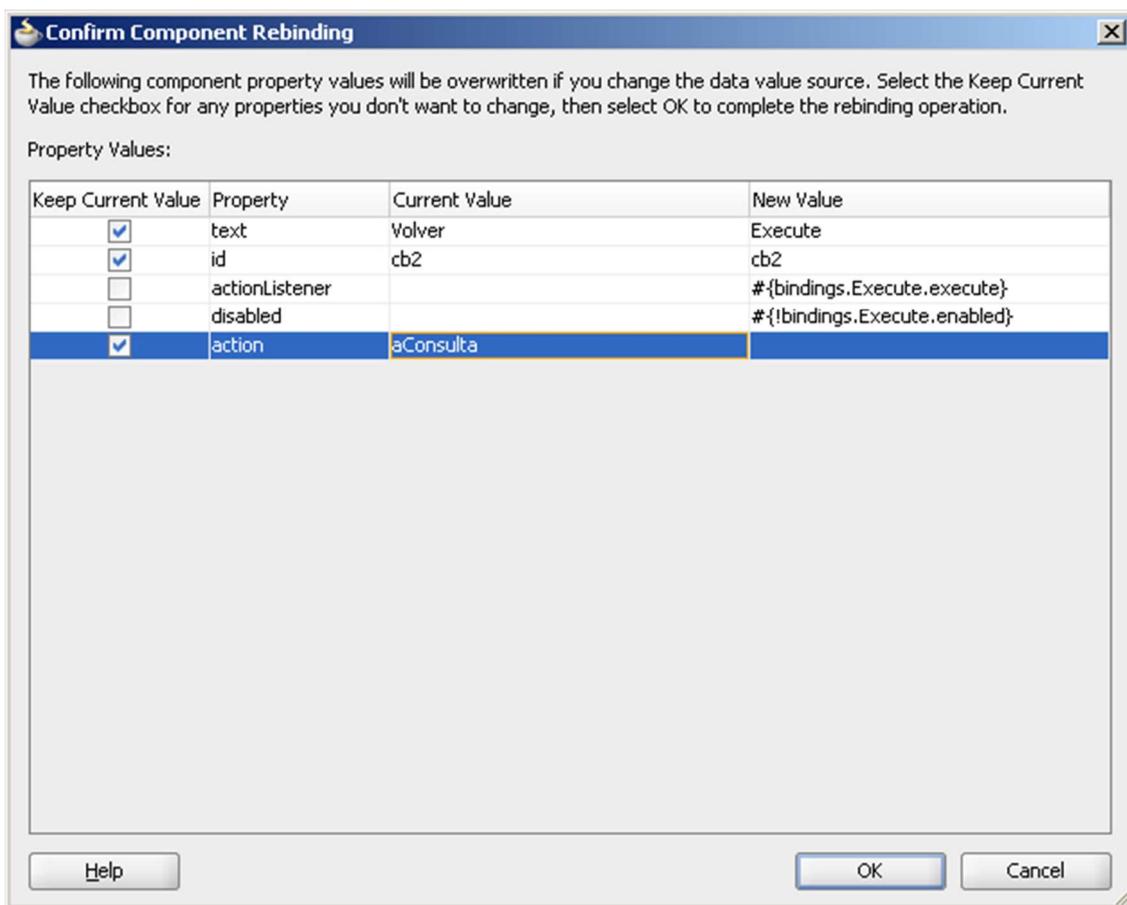


Ilustración 84: Confirmar enlace de componentes

Ahora vamos a pasar a completar el formulario de edición de datos. Hasta el momento hemos añadido cuadros de texto para las distintas propiedades, pero lógicamente, para el estado, lo adecuado sería presentar una lista desplegable que permitiese al usuario elegir el estado de entre los disponibles.

Lo primero que vamos a hacer es añadir a la página esta lista desplegable. Como simplemente queremos que muestre todos los estados, no hay más que arrastrar desde el DataControl la colección `estadosFindAll` como un *ADF Select One Choice*, dentro del grupo *Single Selection*. En el diálogo elegimos el atributo que se mostrará en el selector, que será *estado*, y aceptamos.

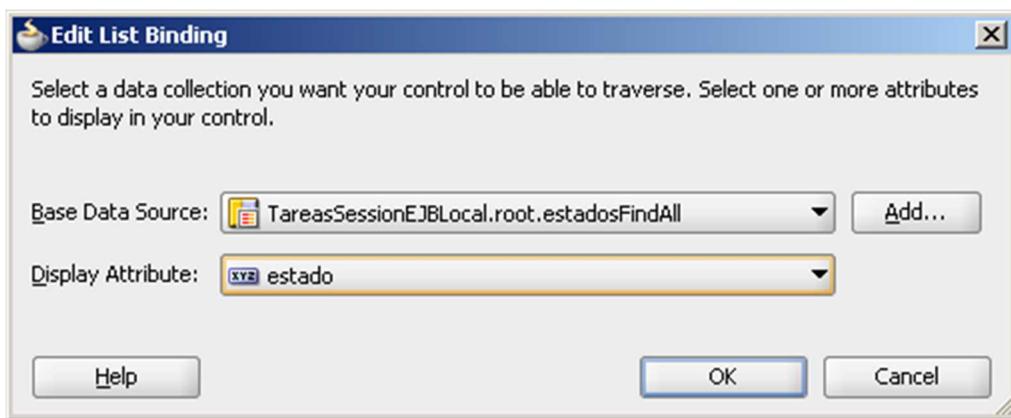


Ilustración 85: Editar enlace de lista

Como vamos a usar este selector, podemos eliminar el cuadro de texto *sid*, que creamos anteriormente, haciendo clic derecho y eligiendo *Delete*.



Ilustración 86: Diseño de nuevoTema

Lo único que nos queda es dar funcionalidad al botón *Submit* para guardar los datos. Su función será la de insertar en la base de datos el nuevo tema creado, para lo que utilizaremos la función *persistTemas* de nuestro EJB. Este método lo tenemos disponible directamente en nuestro DataControl.

Sin embargo, antes de utilizarlo vamos a hacer un pequeño cambio. Tal como hemos definido la página, los valores de los campos *incidencias*, *descripción* y *tema* de la interfaz están enlazados directamente con las propiedades correspondientes de la entidad que creamos al acceder a esta página. Por tanto, si persistimos esa entidad, esos valores se guardarán en la base de datos.

No ocurre lo mismo con el valor del campo *estado*. Lo que hemos hecho ha sido mostrar una lista con los estados registrados en el sistema, pero no hemos enlazado el valor seleccionado en la lista con el campo *estado* del tema. Esto lo haremos directamente en código en lugar de a través de los bindings.

Vamos a sobrecargar el método *persistTemas* añadiendo una versión que recibirá tanto un objeto *Temas* como uno *Estados*. Abrimos la implementación del EJB, y añadimos el siguiente código.

```
public Temas persistTemas(Temas temas, Estados estados) {
    estados = em.find(Estados.class, estados.getSid());
    temas.setEstados(estados);
    estados.addTemas(temas);
    em.persist(temas);
    return temas;
}
```

Ilustración 87: Código de persistTemas

Además de en la implementación, tenemos que añadir el método a las interfaces local y remota. Podemos acceder a ellas desde el navegador de aplicaciones, o yendo a la cabecera de la clase del EJB, *TareasSessionEJBBean*, y clicando sobre el

nombre de las interfaces en la cláusula *implements* mientras mantenemos pulsado el botón control.

Una vez publicado el método, tenemos que volver a crear nuestro *DataControl* para que lo incluya. Accedemos al menú contextual del elemento *TareasSessionEJBBean*, y elegimos la opción *Create Data Control*. En el diálogo volvemos a seleccionar la interfaz local. Una vez terminado el proceso, podemos abrir el fichero *TareasSessionEJBLocal.xml* y veremos que hay dos *methodAccessor* con identificador *persistTemas*.

Si volvemos a la página *nuevoTema* y exploramos el *DataControl* ahora, veremos que el nuevo método aparece entre los disponibles. Lo seleccionamos y arrastramos sobre el botón *Submit*.

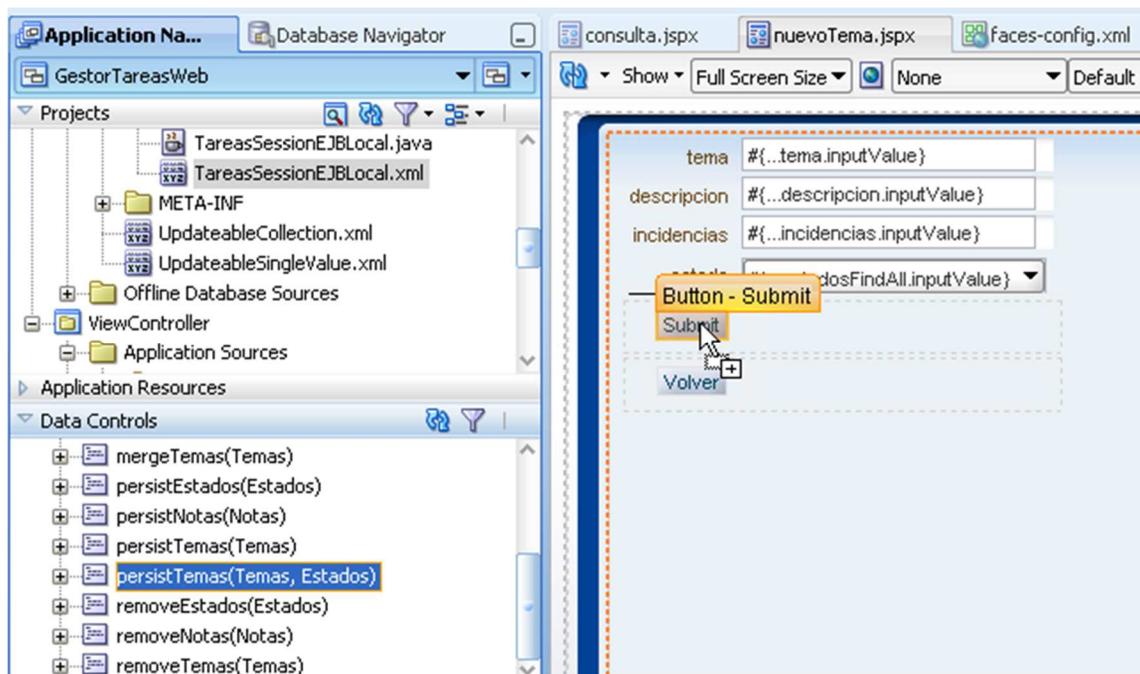


Ilustración 88: Arrastrar persistTemas

Lo primero que veremos será el diálogo de cambio de propiedades del botón, en el que podemos dejar que se cambie todo. Posteriormente aparecerá otro diálogo en el que tenemos que seleccionar los parámetros que vamos a pasar en la llamada al método. Para ello disponemos de un editor en el que podemos navegar sobre la estructura de objetos creados en nuestro enlace de datos, y al que accedemos seleccionando en el desplegable *Show EL Expression Builder*. Elegimos *ADF Bindings – bindings – temasFindAllIterator – currentRow – dataProvider* y *ADF Bindings – bindings – estadosFindAllIterator – currentRow – dataProvider*, respectivamente para el tema y el estado. De esta forma le estamos diciendo que debe pasar como parámetros al método el tema actual (el que se está editando), y el estado seleccionado en el desplegable.

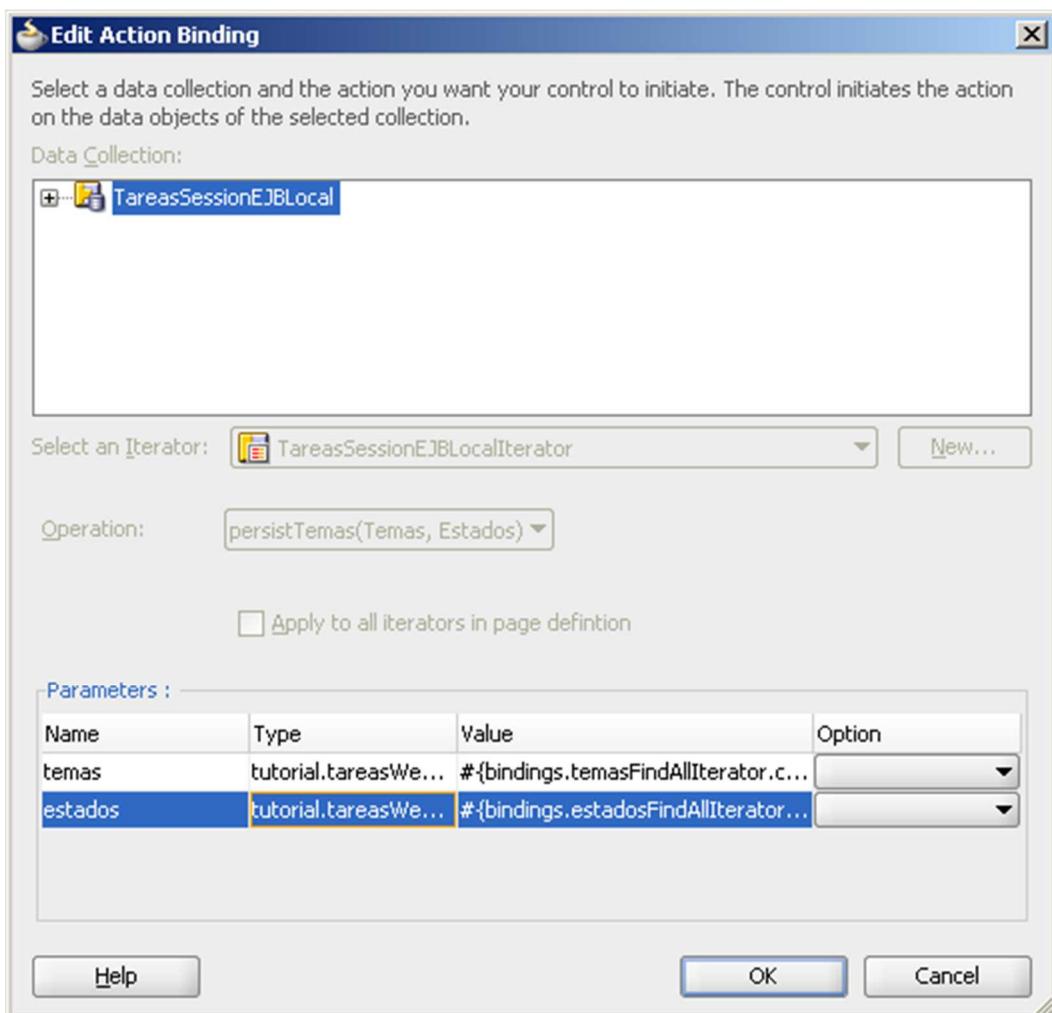


Ilustración 89: Editar enlace de acción

Ahora podemos ir al pageDef para ver el binding, en el que se ha creado un nuevo elemento *methodAction*.

```
<methodAction id="persistTemas" RequiresUpdateModel="true"
    Action="invokeMethod" MethodName="persistTemas"
    IsViewObjectMethod="false" DataControl="TareasSessionEJBLocal"
    InstanceName="TareasSessionEJBLocal.dataProvider"
    ReturnName="TareasSessionEJBLocal.methodResults.persistTemas_TareasSessionEJBLocal_dataProvider_persistTemas_result">
    <NamedData NDName="temas"
        NDValue="#{bindings.temasFindAllIterator.currentRow.dataProvider}"
        NDType="tutorial.tareasWeb.entities.Temas"/>
    <NamedData NDName="estados"
        NDValue="#{bindings.estadosFindAllIterator.currentRow.dataProvider}"
        NDType="tutorial.tareasWeb.entities.Estados"/>
</methodAction>
```

Ilustración 90: Código de methodAction

Para terminar con esta página sólo tenemos que dar valor a la propiedad *Action* del botón, de forma que tras persistir el tema se vuelva a la página de consulta, para lo que utilizamos la regla de navegación global que ya habíamos creado, *aConsulta*.

Vamos a añadir ahora una página para la edición de temas. Comenzamos creando una nueva página jspx que llamaremos *editarTemas.jspx*, una regla de navegación desde *consulta.jspx* hacia ella, y un elemento de menú con el texto *Editar Temas*, que permitirá navegar hacia la nueva interfaz.

En este caso no necesitamos realizar ninguna acción adicional a la hora de navegar. En la página de consulta seleccionaremos un tema, y ese será el que se muestre en la nueva página.

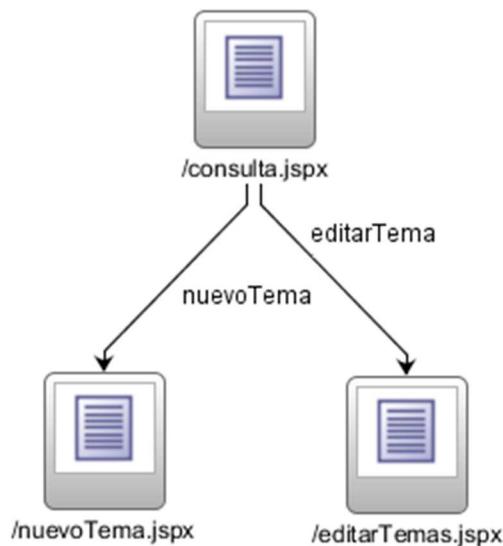


Ilustración 91: Faces-config.xml

Desde el editor gráfico, vamos a arrastrar sobre la página el elemento *temasFindAll* del DataControl. Igual que antes, seleccionamos la opción *ADF Form*, y quitamos los atributos que sobran, dejando el tema, la descripción, la incidencia, y el sid del estado. También incluimos un botón submit.

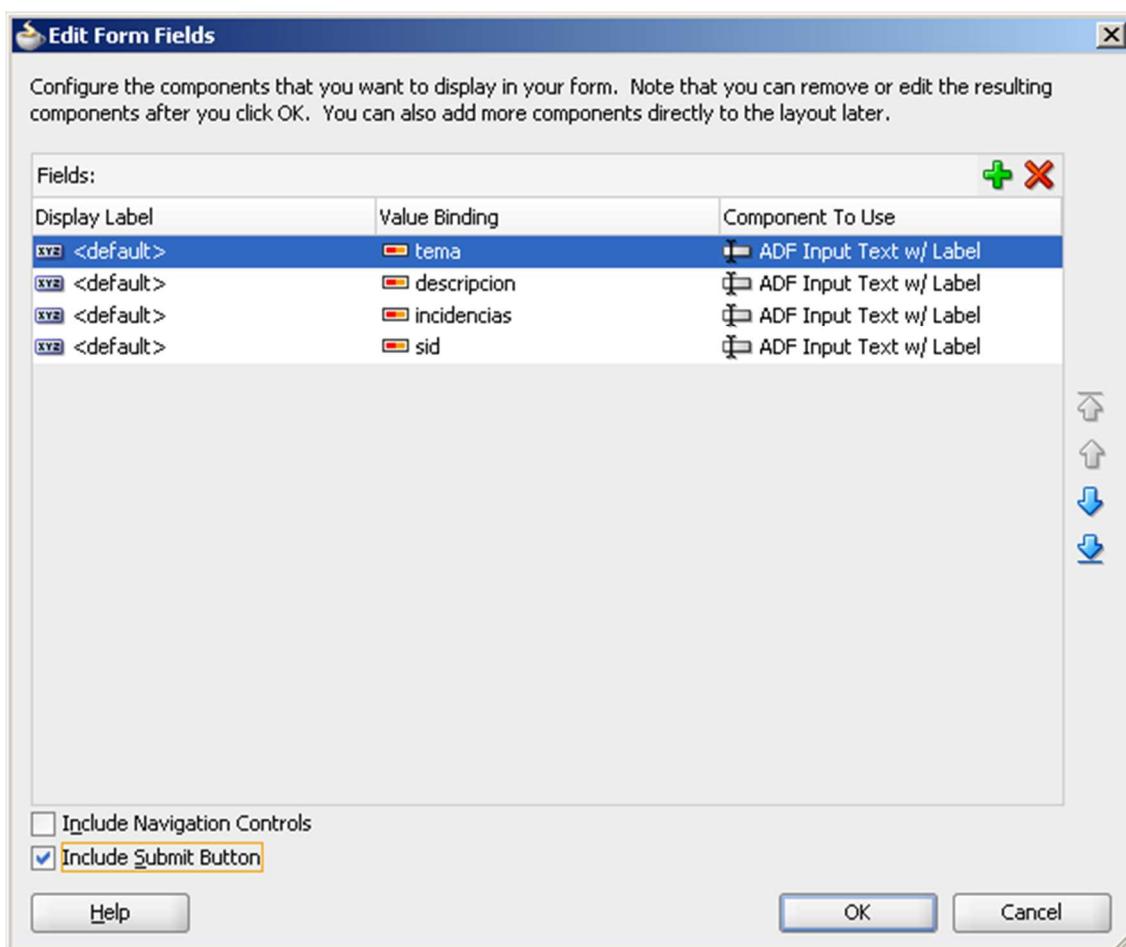


Ilustración 92: Editar campos de formulario

Para terminar con la parte del comportamiento que es idéntico a la página anterior, añadimos el botón para cancelar los cambios, que navegará hacia la página de consulta y volverá a ejecutar el iterador *temasFindAll* para desechar los cambios realizados.

A partir de aquí la página se comportará de forma distinta. Lo primero es incluir un selector para el campo estado. Lo creamos como siempre, arrastrando *estadosFindAll* como un componente *ADF Select One Choice*. En el diálogo seleccionamos el atributo *estado* para ser mostrado.

En este caso vamos a modificar el enlace de datos para que el valor seleccionado en el desplegable sea el que tome el campo estados del tema que se está editando sin necesidad de realizar ninguna acción por código. Así conseguiremos además que el valor seleccionado al acceder a esta página en el desplegable sea el que actualmente tenga asignado el tema.

Si vamos al *pageDef*, podemos ver que se han creado, entre otros elementos, los siguientes:

- *temasFindAllIterator*.- el iterador que contiene todos los temas registrados, y cuyo elemento actual es el que estamos editando.
- *estadosIterator*.- este iterador enlaza con la propiedad *estados* del *Tema* actual de *temasFindAllIterator*. Es decir, tenemos un iterador que contendrá un único estado, que es el que tiene asignado el tema que estamos editando.

- `estadosFindAllIterator`.- otro iterador que, en este caso, contiene todos los estados registrados en el sistema.
- `sid`.- un *binding* de tipo `attributeValue` que enlaza con la propiedad `sid` de `estadosIterator`. Es decir, el sid del estado asignado al tema que estamos editando.
- `estadosFindAll`.- una lista que está enlazada con `estadosFindAllIterator` y que, por tanto, mostrará todos los estados registrados.

Si nos fijamos en este último elemento, contiene dos atributos enlazados con el mismo iterador: *IterBinding* y *ListIter*. El primero de ellos indica cuál es el iterador al que está asociada la lista. El segundo indica cuál es el iterador al que está asociada la lista fuente del elemento *list*. Esto es, en *ListIter* vamos a indicar cuál es el iterador del que se tomarán los datos a mostrar, y en *IterBinding* cuál es el iterador en el que volcaremos la selección.

Modificamos el valor del atributo *IterBinding* a *estadosIterator*. De esta forma el valor actual de la lista se enlaza con el estado del tema que se está editando. Al ser distintos *IterBinding* y *ListIter* tenemos que indicar qué atributo de cada una de las colecciones vamos a utilizar en el enlace, y cuál es el que queremos mostrar al usuario. El enlace lo haremos con el *sid*, y en la página mostraremos el *estado*. También tenemos que modificar el *ListOperMode*, dándole el valor *setAttribute*.

```
<list IterBinding="estadosIterator" ListOperMode="setAttribute"
      ListIter="estadosFindAllIterator" id="estadosFindAll"
      DTSSupportsMRU="true">
    <AttrNames>
      <Item Value="sid"/>
    </AttrNames>
    <ListAttrNames>
      <Item Value="sid"/>
    </ListAttrNames>
    <ListDisplayAttrNames>
      <Item Value="estado"/>
    </ListDisplayAttrNames>
  </list>
```

Ilustración 93: Código de list

Lo único que nos queda es eliminar el cuadro de texto con el campo *sid* del *estado* que incluimos en la página. No podemos eliminarlo utilizando la opción *Delete* del menú contextual, ya que eso implicaría que se borrarían también los *bindings* que tiene asociados, y necesitamos mantenerlos para que la página funcione. Por tanto lo seleccionamos, y elegimos *Go to Source* en el menú. El IDE nos llevará a la pestaña *Source*, y el elemento *inputText* estará seleccionado, por lo que no tenemos más que suprimirlo.

Una vez terminado el *binding*, configuramos el botón *submit* arrastrando sobre él el método *mergeTemas*. En el diálogo elegimos como parámetro `#{}bindings.temasFindAllIterator.currentRow.dataProvider`. Después podemos ponerle como texto *Guardar*.

Como último cambio, modificamos el código del método *mergeTemas* en el EJB de sesión. Estas nuevas líneas son necesarias para que el estado seleccionado se actualice en la base de datos.

```
public Temas mergeTemas(Temas temas) {
    Estados estado = em.find(Estados.class, temas.getEstados().getSid());
    temas.setEstados(estado);
    return em.merge(temas);
}
```

Ilustración 94: Código de mergeTemas

En los siguientes pasos, vamos a añadir a nuestra aplicación todo lo necesario para trabajar con las anotaciones de los temas. Esto incluye la posibilidad de crear, editar y eliminar notas.

En primer lugar creamos en *consulta.jspx* un nuevo menú, *Notas*, con tres elementos: *Nueva nota*, *Editar nota*, y *Eliminar Nota*.

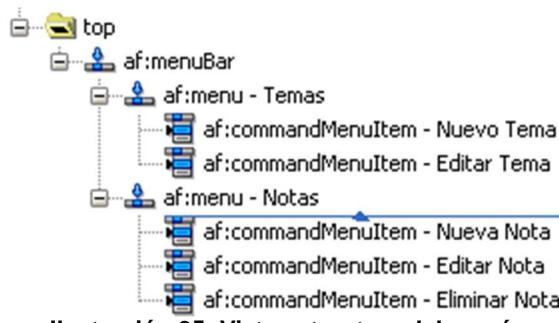


Ilustración 95: Vista estructura del menú

Accedemos al faces-config, y creamos dos nuevas páginas: *nuevaNota.jspx* y *editarNota.jspx*. Creamos reglas de navegación desde la página de consulta hacia ellas, obteniendo el resultado que se ve a continuación.



Ilustración 96: Faces-config

Empezamos con la página para crear nuevas notas, cuyo funcionamiento es bastante sencillo. En primer lugar, tenemos que asegurarnos de que cuando el usuario pulse la opción correspondiente del menú, se cree una nueva nota, por lo que, en *consulta.jspx*, arrastramos el método *Create* del iterador *notasList* (que se encuentra bajo *temasFindAll*) hacia el elemento correspondiente del menú. Lo más cómodo es arrastrar sobre la vista estructura. Al hacerlo se muestra la pantalla de confirmación de modificación de datos, en la que mantenemos lo que nos interesa.

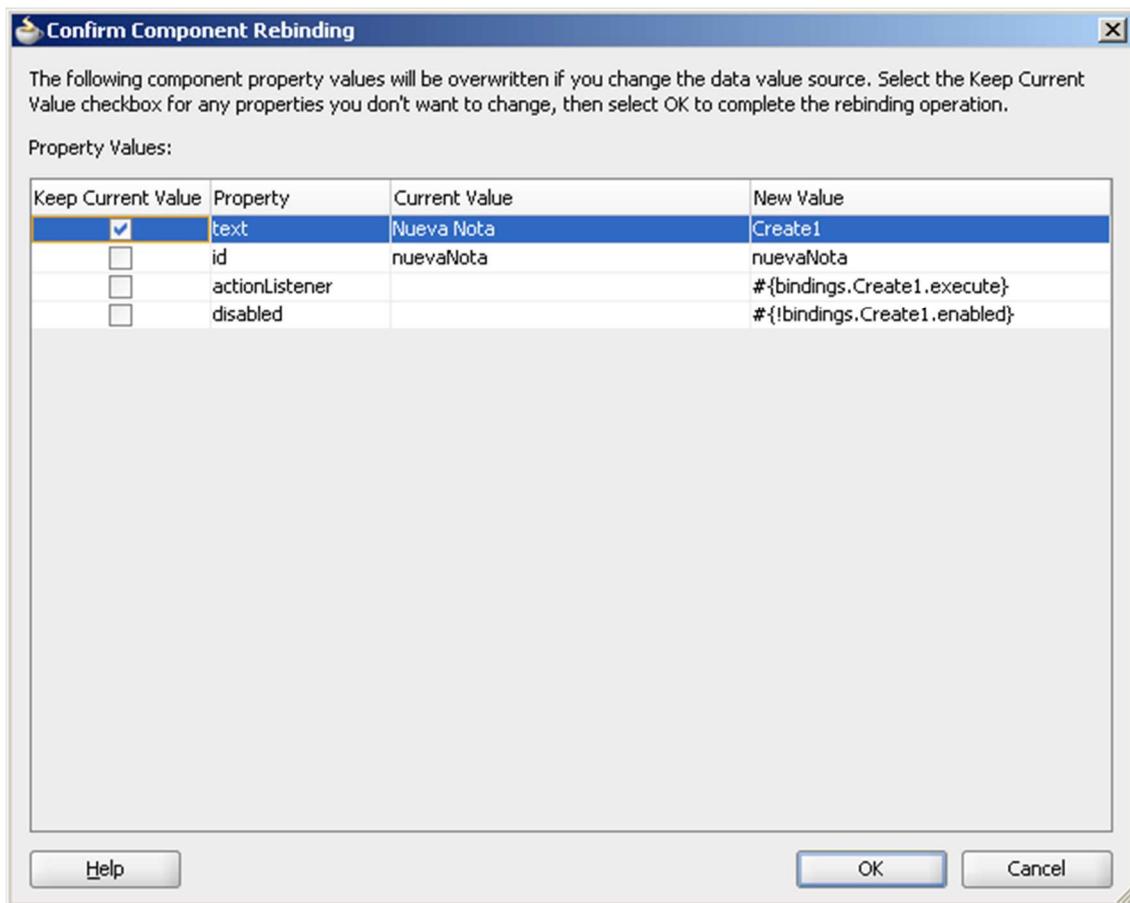


Ilustración 97: Confirmar enlace de componentes

Ahora sí nos centramos en la página *nuevaNota.jspx*. Arrastramos sobre ella el iterador *notasList* de nuestro DataControl, y en el menú elegimos *ADF Form*. En este caso vamos a dejar todos los campos que corresponden directamente a una nota, excepto el sid, que se generará automáticamente. Añadimos también el botón *submit*.

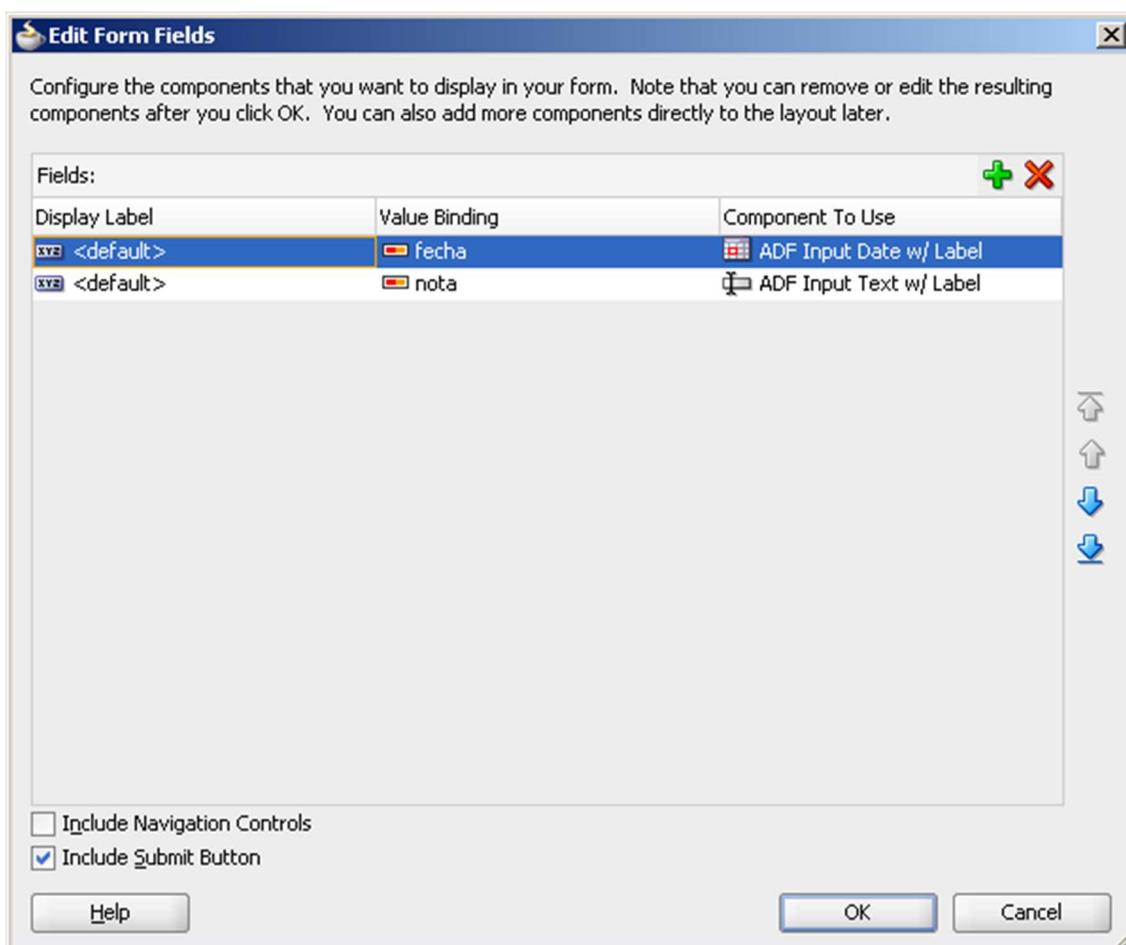


Ilustración 98: Editar campos de formulario

Obtenemos de forma automática un formulario para edición de notas. Sin embargo, para la edición de la nota en sí (campo *nota*), ADF elige por defecto un editor de texto simple, *InputText*. Vamos a cambiarlo por un editor enriquecido, que nos ofrecerá la posibilidad de crear las notas utilizando *html*, por lo que podremos aplicar formato a las mismas.

Si nos fijamos en la captura anterior, al crear el formulario el IDE nos preguntó qué componente queríamos usar para cada campo. Podríamos haberlo modificado en ese paso, pero no lo hemos hecho porque el componente que queremos no aparece en el desplegable.

Accedemos al código fuente de la página *nuevaNota*, seleccionando el campo *nota*, pulsando con el botón derecho, y eligiendo *Go to Source*. Así el IDE nos lleva directamente al código fuente del componente que queremos modificar. Vamos a utilizar el componente *richTextEditor*, por lo que cambiamos la etiqueta *af:inputText* por *af:richTextEditor*. Al hacerlo el editor nos lanza un error indicando que el tag *validator* no puede ser hijo de *richTextEditor*. El *validator* se utiliza para definir reglas de validación sobre los valores de entrada, pero como no las hemos definido podemos eliminar ese tag. Ahora nos indica que el atributo *maxLength* no está entre los atributos de *richTextEditor*, así que lo quitamos también.

El resultado que obtenemos es el siguiente.

```
<af:richTextEditor value="#{bindings.nota.inputValue}"  
    label="#{bindings.nota.hints.label}"  
    required="#{bindings.nota.hints.mandatory}"  
    columns="#{bindings.nota.hints.displayWidth}"  
    shortDesc="#{bindings.nota.hints.tooltip}"  
    id="it1">  
</af:richTextEditor>
```

Ilustración 99: Código de richTextEditor

Si volvemos a la vista de diseño, veremos que la apariencia ha cambiado mucho, y ahora tenemos un campo de texto amplio, con una barra de botones de formato.

Continuamos enlazando el botón de submit con el guardado de los datos. Para ello, seleccionamos del DataControl el método *persistNotas*, y lo arrastramos sobre el botón. En el diálogo seleccionamos las opciones que se muestran (en *value* elegimos `#{bindings.notasListIterator.currentRow.dataProvider}`).

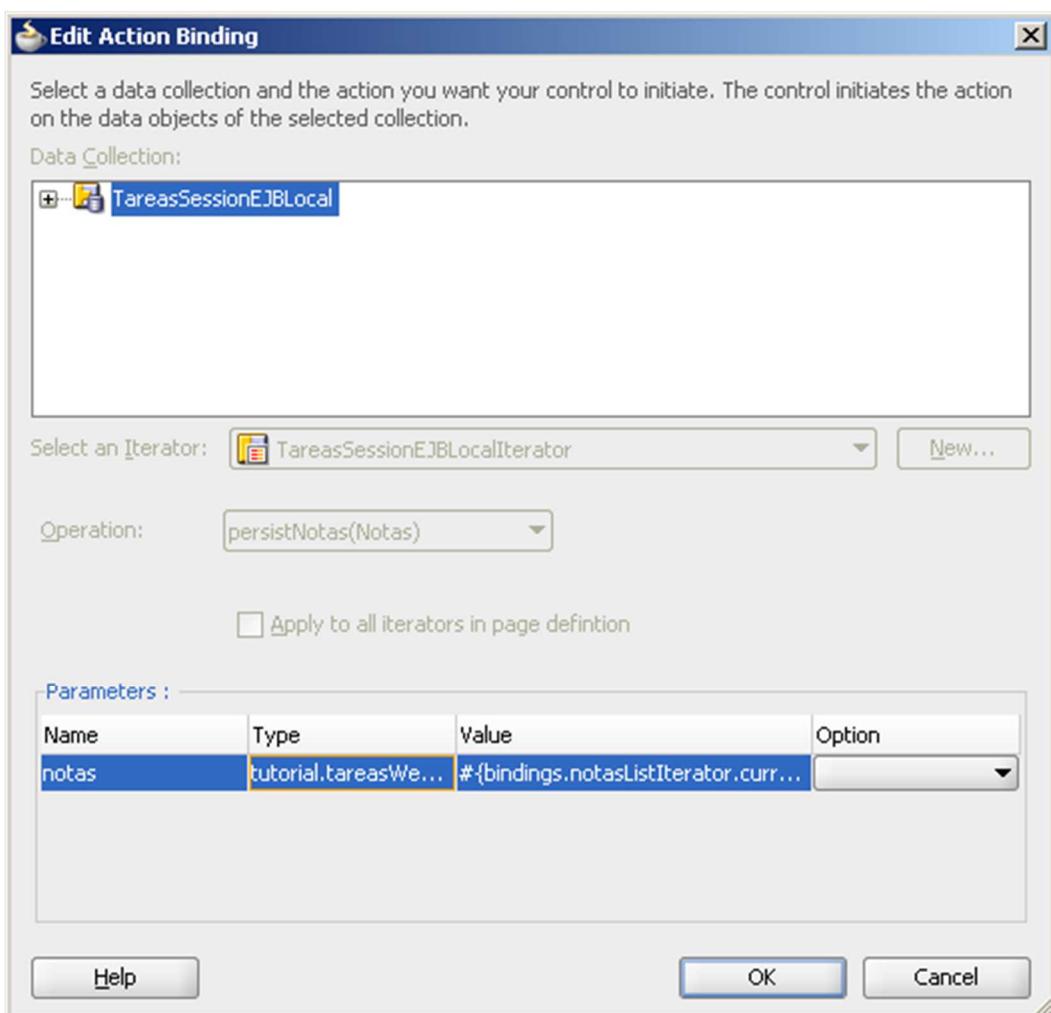


Ilustración 100: Editar enlace de acción

En el siguiente diálogo podemos mantener el valor del atributo *Text* del botón, y antes de continuar podemos indicar que cuando se pulse se debe volver a la página de consulta modificando la propiedad *action*.

Si echamos un vistazo al método *persistNotas* del EJB *TareasSessionEJBBean*, vemos que lo único que hace es decirle al *entityManager* que persista el objeto, y devolverlo.

De esta forma estamos haciendo que se inserte en la tabla de notas de la base de datos un nuevo registro con sus datos correctamente informados, incluida la referencia hacia el tema correspondiente. Por tanto el guardado de los datos se ha efectuado correctamente. Sin embargo tenemos que tener en cuenta que la responsabilidad de mantener los datos de nuestra aplicación, y las relaciones entre las entidades, recae sobre nosotros mismos. Esto implica que tenemos que asegurarnos de que la relación entre el objeto nota que estamos persistiendo y su tema asociado está completa en ambos sentidos: la nota referencia al tema, y el tema tiene a la nota en su lista de notas. Para ello dejamos el código como se muestra.

```
public Notas persistNotas(Notas notas) {  
    Temas temas = em.find(Temas.class, notas.getTemas().getSid());  
    em.persist(notas);  
    notas.setTemas(temas);  
    temas.addNotas(notas);  
    return notas;  
}
```

Ilustración 101: Códigos de persistNotas

Exactamente lo mismo ocurre cuando vamos a eliminar una nota, por lo que aprovechamos para modificar también el código del método *removeNotas*, dejándolo como sigue.

```
public void removeNotas(Notas notas) {  
    notas = em.find(Notas.class, notas.getSid());  
    Temas temas = em.find(Temas.class, notas.getTemas().getSid());  
    em.remove(notas);  
    temas.removeNotas(notas);  
}
```

Ilustración 102: Código de removeNotas

Para terminar con la página añadimos un botón con el texto *volver*, y lo enlazamos con la operación *Execute* del iterador *temasFindAll*, para refrescar los datos. Como action seleccionamos la regla de navegación *aConsulta*, para volver a la página principal.

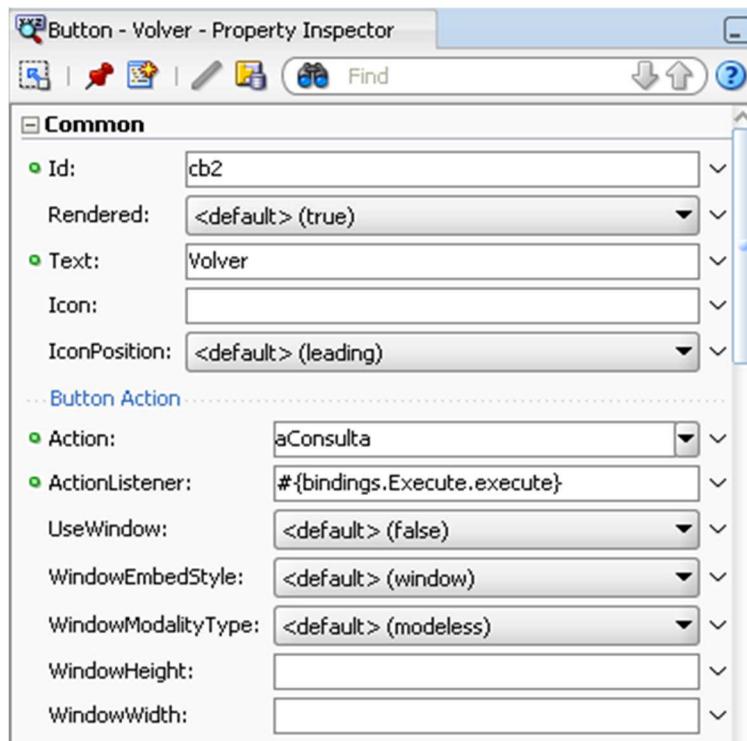


Ilustración 103: Propiedades de botón

De esta forma tenemos la página completamente operativa, y podemos crear notas. No olvidemos fijar la propiedad *Action* del menú *nuevaNota* para navegar hacia esta pantalla.

Edición de notas

Lo primero es configurar el elemento del menú que da acceso a ella. En este caso, además de indicar en el *action* la regla de navegación correspondiente (*editarNota*), vamos a modificar la propiedad *disabled*, de forma que esta opción del menú sólo esté disponible si hay una nota seleccionada en la interfaz. Esto lo conseguimos con la expresión que se muestra a continuación, y que se puede introducir gracias al editor, que está disponible pulsando en el ícono de flecha apuntando hacia abajo, situado a la derecha del campo.

Disabled: #{}{bindings.notasListIterator.currentRow ==null}

Ilustración 104: Propiedad disabled

Además, añadimos la propiedad *partialTriggers*, necesaria para que el menú se actualice cuando haya cambios en la tabla *t2*.

La página de edición es prácticamente igual que la de creación, por lo que arrastramos nuevamente el iterador *notasList* que está bajo *temasfindAll*, y elegimos las mismas opciones en el diálogo. Volvemos a modificar también el editor para el campo nota accediendo al código fuente.

El botón submit lo enlazamos ahora con el método *mergeNotas*, tal como se ve a continuación.

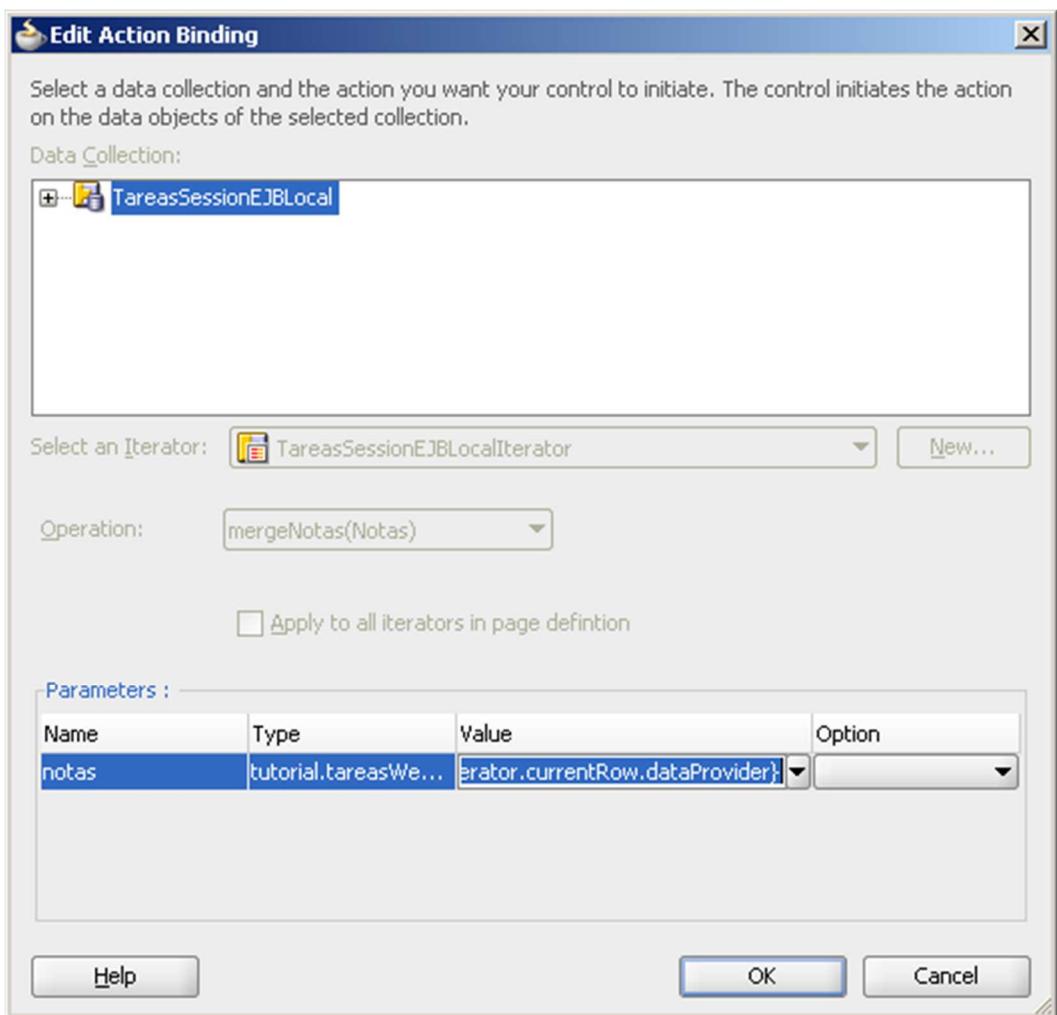


Ilustración 105: Editar enlace de acción

Además de indicar en el action que debe navegar de vuelta a la página de consulta, con la regla *aConsulta*.

Para terminar con esta página añadimos un botón para volver, que navegará hacia consulta, y ejecutará el iterador *temasFindAll* para refrescarlo.

Eliminar notas

Lo último que haremos en relación con las notas es habilitar la posibilidad de eliminarlas. En este caso no necesitaremos una página independiente, sino que llevaremos a cabo el borrado directamente cuando el usuario pulse la opción correspondiente del menú.

Para evitar que se pueda acceder a esta opción sin haber seleccionado previamente una nota, en la propiedad *disabled* utilizaremos la misma expresión que en el botón para editar notas (*#{bindings.notasListIterator.currentRow == null}*). También daremos a *partialTriggers* el valor *t2*.

Lo siguiente sería invocar al método que lleva a cabo el borrado de la nota. Sin embargo, vamos a añadir antes una petición de confirmación al usuario utilizando dos nuevos componentes de ADF Faces: *popup* y *dialog*.

Cuadros de diálogo y popups

El mensaje de confirmación se mostrará en una ventana emergente o popup. El componente necesario lo encontramos en la paleta, bajo la categoría *Common Components* de *ADF Faces*.

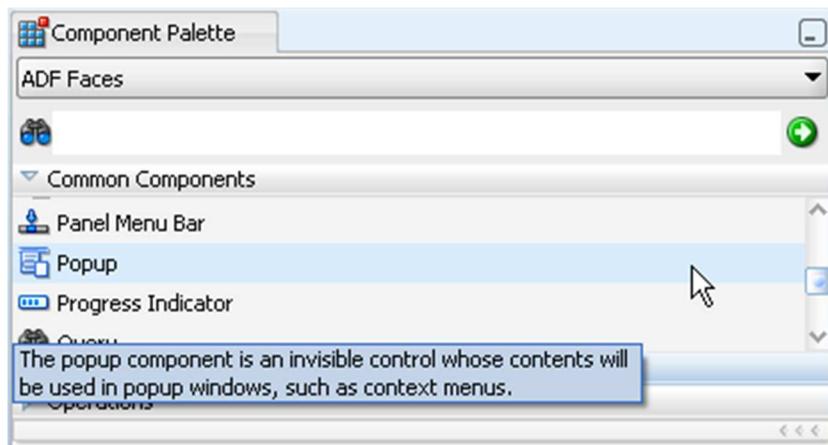


Ilustración 106: Componente popup

El *popup* podemos ubicarlo en cualquier lugar de la página, pero vamos a arrastrarlo a la sección *bottom* de la misma, que se encuentra vacía, para que sea más fácil localizarlo. Cada vez que queramos acceder a sus propiedades para modificarlas no tendremos más que pinchar sobre esta sección en la vista diseño, y podremos verlo en la vista estructura.

Dentro del *popup* vamos a colocar el componente *dialog*, que podemos encontrar también en la sección *Common Components* de la paleta. En la configuración del diálogo vamos a indicar el texto que se mostrará, en la propiedad *title*. Además tenemos que seleccionar el tipo de diálogo. En este caso se nos ofrece una serie de alternativas como *yesNo*, *cancel*, *okCancel*... Si seleccionamos alguna de ellas estamos indicando al diálogo los botones que queremos que muestre. En nuestro caso vamos a elegir la opción *none*, para crear nuestros propios botones. También vamos a deshabilitar la posibilidad de cerrar el diálogo (*CloseIconVisible*). En la vista de diseño podemos ver cómo va quedando nuestro diálogo.

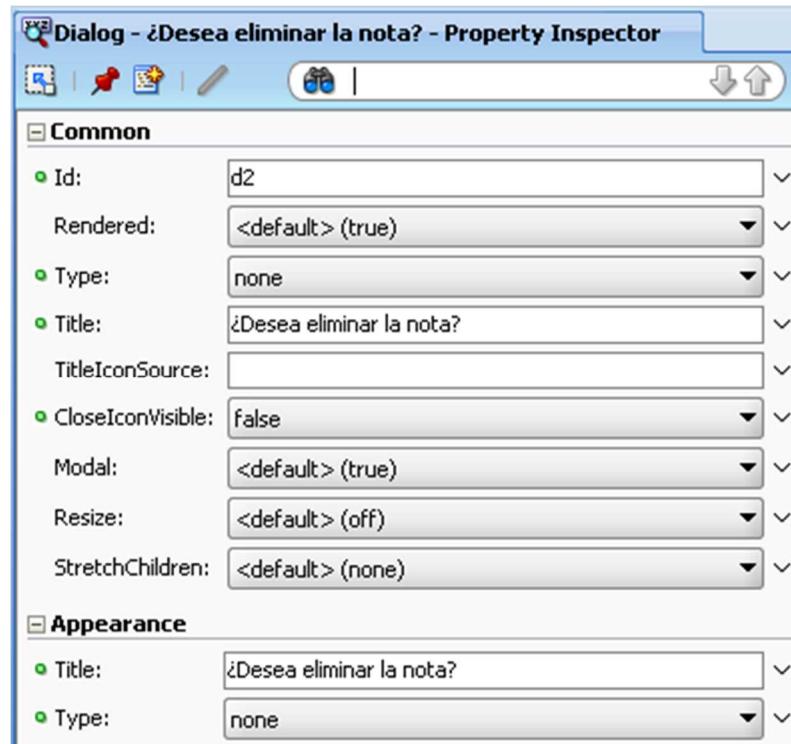


Ilustración 107: Propiedades de dialog

Vamos a añadir dos botones para confirmar y cancelar la eliminación, respectivamente. Para ello editaremos directamente el código del diálogo, por lo que accedemos al mismo en la pestaña *source* de la página.

Lo que hacemos es incluir dos nuevos elementos *commandButton* dentro de la etiqueta *af:dialog*.

```
<af:popup id="p1">
    <af:dialog id="d2" title="¿Desea eliminar la nota?"
        type="none" closeIconVisible="false">
        <af:commandButton id="cbSi" text="Sí"/>
        <af:commandButton id="cbNo" text="No"/>
    </af:dialog>
</af:popup>
```

Ilustración 108: Código de popup

Si volvemos a la vista de diseño podemos ver cómo queda el diálogo. Lo que nos falta ahora es dar funcionalidad a los botones, y para ello vamos a utilizar por primera vez los *backing beans*.

Un *backing bean* no es más que un *JavaBean* asociado con los componentes utilizados en una página JSF. Así, cada una de las propiedades del *backing bean* estará enlazada con un componente de una página. Por ejemplo, si en una página jspx tenemos una tabla (componente *af:table*), en el correspondiente *backing bean* tendremos una propiedad con el mismo identificador, cuyo tipo será *oracle.adf.view.rich.component.rich.data.RichTable*.

Uno de los principales usos que se le puede dar a una *backing bean* es el de invocar varias acciones desde un mismo componente de forma sencilla. Normalmente la acción a ejecutar, por ejemplo, al pulsar un botón, la indicamos mediante su atributo *actionListener*. El problema es que sólo podemos enlazar esta propiedad con una

acción. Si necesitamos invocar más de una, o realizar tratamientos previos, podemos enlazar el botón con un método del *backing bean*, y desde él invocar varias acciones en el orden deseado.

En nuestro caso, queremos que cuando se pulse el botón para aceptar el diálogo se ejecuten dos acciones. Por un lado, hay que eliminar la nota seleccionada, y por otro, hay que cerrar el diálogo.

Para crear el *backing bean* tenemos que acceder al fichero *faces-config.xml*, en la vista *overview*, y ahí a la sección *ManagedBeans*. Pulsamos el botón para agregar uno nuevo, y en el diálogo indicamos el nombre que le queremos dar al bean (el identificador mediante el que accederemos a él desde las páginas), el nombre de la clase en que lo queremos implementar, y el paquete. Marcamos la opción para que el IDE cree la clase en caso de que no exista.

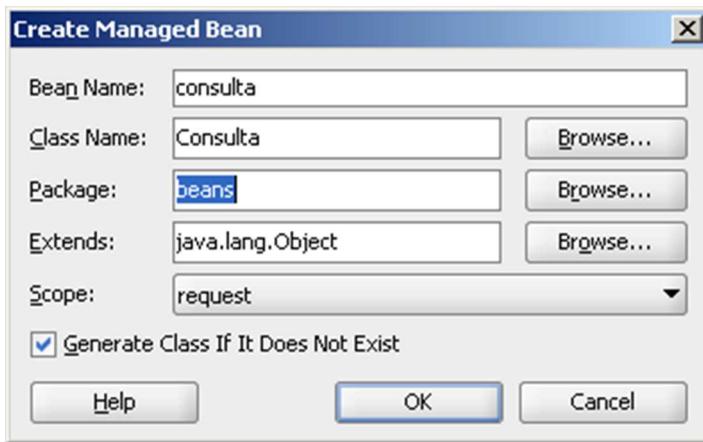


Ilustración 109: Crear Managed Bean

Vemos que en el proyecto *ViewController* se ha creado la clase que hemos indicado, y si la abrimos estará vacía. Lo primero que haremos será crear dos propiedades para enlazar con elementos de la interfaz de consulta: el *popup*, y la tabla de temas (la tabla no la necesitamos, pero la creamos para ver otro ejemplo). Las clases que corresponden a estos componentes son *oracle.adf.view.rich.component.rich.RichPopup* y *oracle.adf.view.rich.component.rich.data.RichTable*. Generamos también los métodos *get* y *set*, para lo que podemos hacer clic derecho sobre el código de la clase, y elegir la opción *Generate Accessors*.

```
package beans;

import oracle.adf.view.rich.component.rich.RichPopup;
import oracle.adf.view.rich.component.rich.data.RichTable;

public class Consulta {

    RichTable tabla;
    RichPopup popup;

    public Consulta() {
    }

    public void setPopup(RichPopup popup) {
        this.popup = popup;
    }

    public RichPopup getPopup() {
        return popup;
    }

    public void setTabla(RichTable tabla) {
        this.tabla = tabla;
    }

    public RichTable getTabla() {
        return tabla;
    }
}
```

Ilustración 110: Código de Consulta

Para completar el enlace entre los componentes de la interfaz y las propiedades del *backing bean* tenemos que informar el campo *bindings* de los primeros en la página jspx. Para el caso del popup, por ejemplo, ponemos `#{consulta.popup}`.

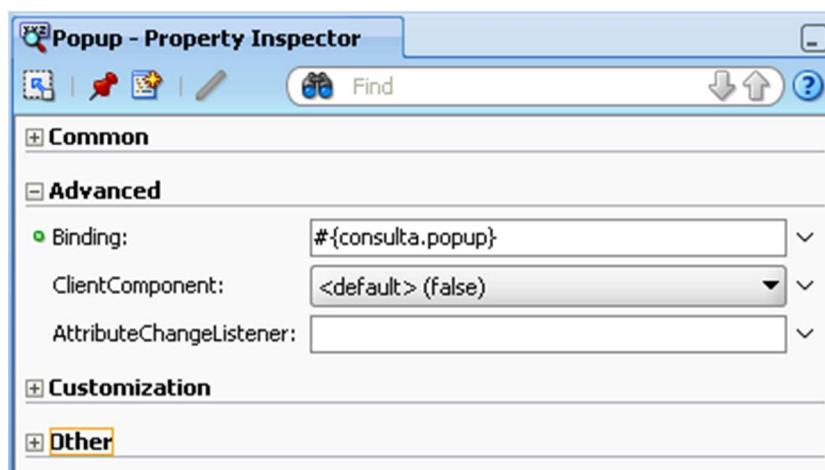


Ilustración 111: Propiedades de popup

Ahora vamos a incluir un método para cada uno de los botones del diálogo, a los que llamaremos *yesAction* y *noAction*. Lo primero que haremos en ambos casos es cerrar el diálogo. Si estamos trabajando con una versión de ADF 11.1.1.3 o superior

podemos utilizar el método *hide* de la clase *RichPopup*. En caso contrario, tendremos que utilizar el fragmento de código que se muestra a continuación.

```
FacesContext context = FacesContext.getCurrentInstance();
String popupId = popup.getClientId(context);
StringBuilder script = new StringBuilder();
script.append("var popup = AdfPage.PAGE.findComponent('');");
script.append("if (popup.isPopupVisible()) { ");
script.append("popup.hide();");
ExtendedRenderKitService erks =
    Service.getService(context.getRenderKit(), ExtendedRenderKitService.class);
erks.addScript(context, script.toString());
```

Ilustración 112: Código para cerrar popup

Además de lo explicado, en el método *yesAction* tendremos que llevar a cabo el borrado de la nota, y el refresco del iterador de temas. Para ello tenemos que invocar los métodos *removeNotas* y *Execute*, y por supuesto lo haremos a través del enlace de datos.

Para que el IDE cree por nosotros el *methodAction* necesario en el *pageDef*, accedemos al *DataControl*, y buscamos el método *removeNotas*. A continuación lo arrastramos sobre el botón Yes que creamos dentro del diálogo, y que podemos ver en la vista de estructura. Aparecerá el diálogo para editar el action binding. Elegimos el parámetro (como siempre, la nota actualmente seleccionada: `#{bindings.notasListIterator.currentRow.dataProvider}`), y aceptamos.

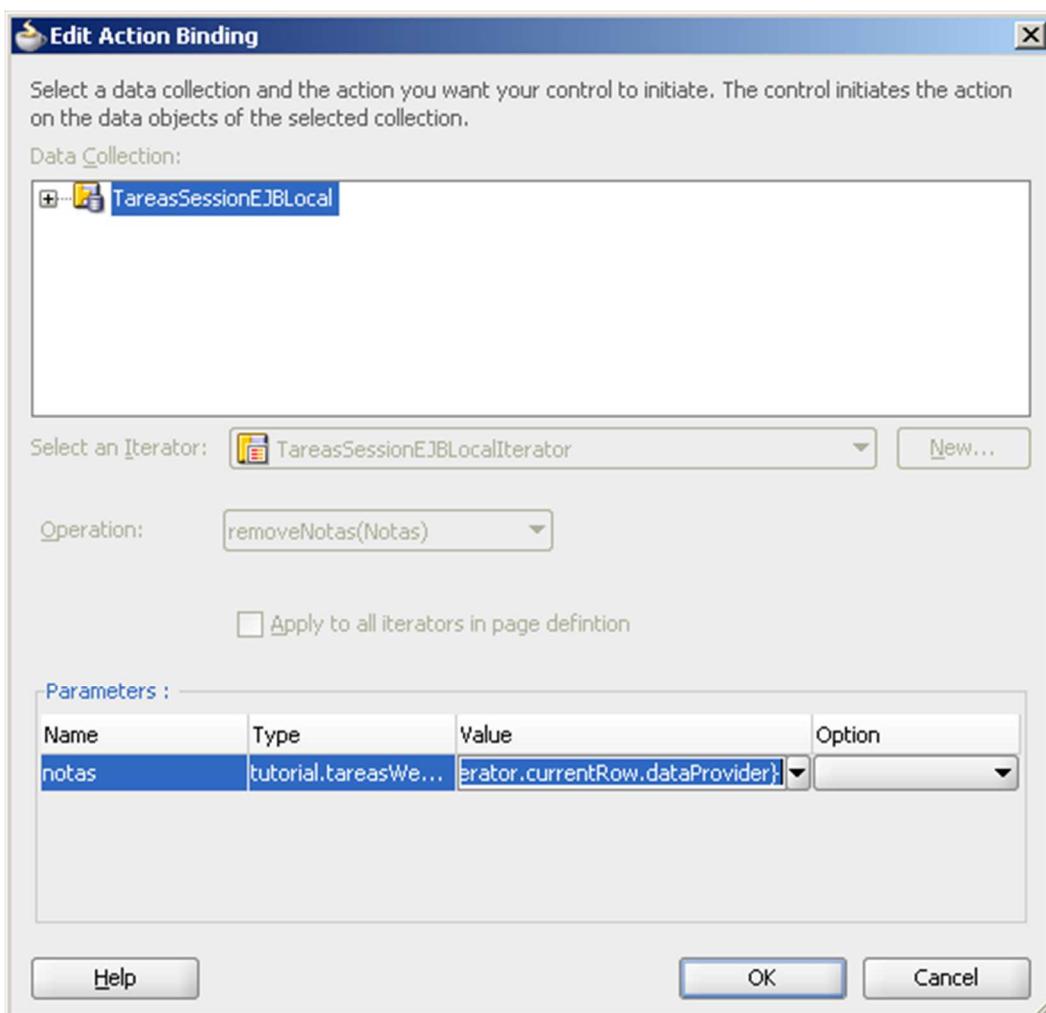


Ilustración 113: Editar enlace de acción

Si nos fijamos en el código o las propiedades del botón, se ha añadido valor al atributo *actionListener*. Si con el botón derecho elegimos la opción para ir al *pageDef*, podemos ver la definición que se ha dado al *methodAction* para invocar al EJB.

```
<methodAction id="removeNotas" RequiresUpdateModel="true"
    Action="invokeMethod" MethodName="removeNotas"
    IsViewObjectMethod="false" DataControl="TareasSessionEJBLocal"
    InstanceName="TareasSessionEJBLocal.dataProvider">
    <NamedData NDName="notas"
        NDValue="#{bindings.notasListIterator.currentRow.dataProvider}"
        NDType="tutorial.tareasWeb.entities.Notas"/>
</methodAction>
```

Ilustración 114: Código de MethodAction

Pero en este caso no queremos invocar directamente al EJB, sino que queremos pasar por el *backing bean* que hemos definido. Por tanto, cambiamos el valor del *action* del botón como se muestra, y vaciamos el atributo *actionListener*.

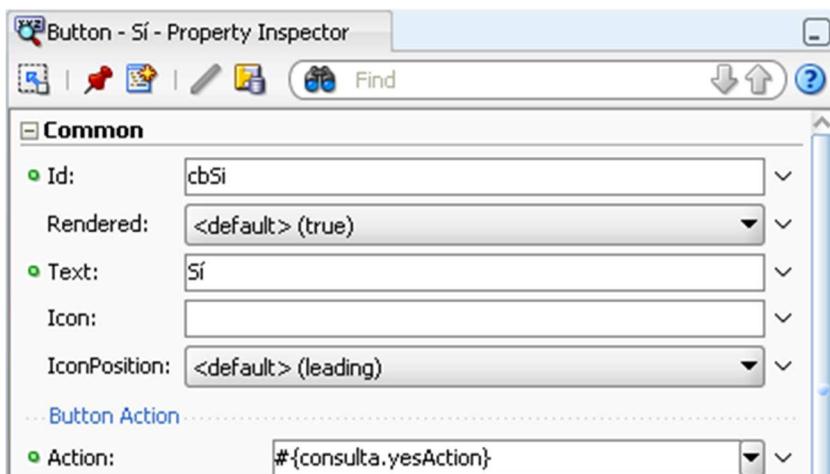


Ilustración 115: Propiedades de botón

Recordemos que *consulta* es el identificador que le dimos al bean que creamos en nuestro faces-config, de tipo Consulta. Por tanto, lo que estamos diciendo es que cuando se pulse el botón, queremos que se invoque el método *yesAction* de la clase Consulta.

Necesitamos crear un binding también para el método *Execute* de *temasFindAll*, pero no podemos hacerlo arrastrando sobre el mismo botón porque entonces el IDE eliminaría el binding de *removeNotas*. Por tanto lo más sencillo es editar en este caso directamente el *pageDef*.

Accedemos al mismo y, desde la vista de estructura, seleccionamos el grupo *bindings*. Con el botón derecho accedemos a *Insert inside binding – ADF Swing Bindings – action*.

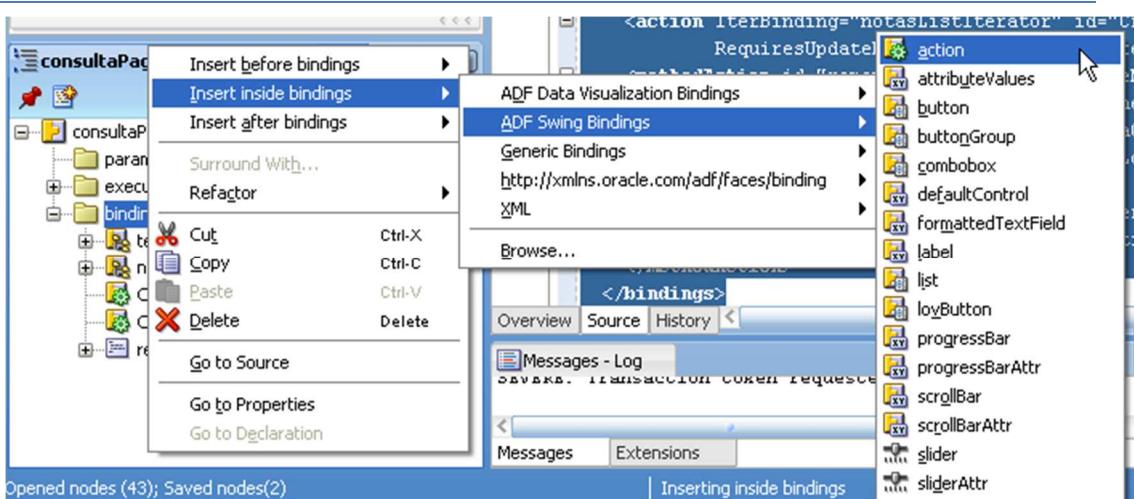


Ilustración 116: Insertar action

Aparece un nuevo diálogo para configurar la acción. Queremos actualizar *temasFindAll*, luego lo seleccionamos en *Data Collection*, y en *Operation* elegimos *Execute*.

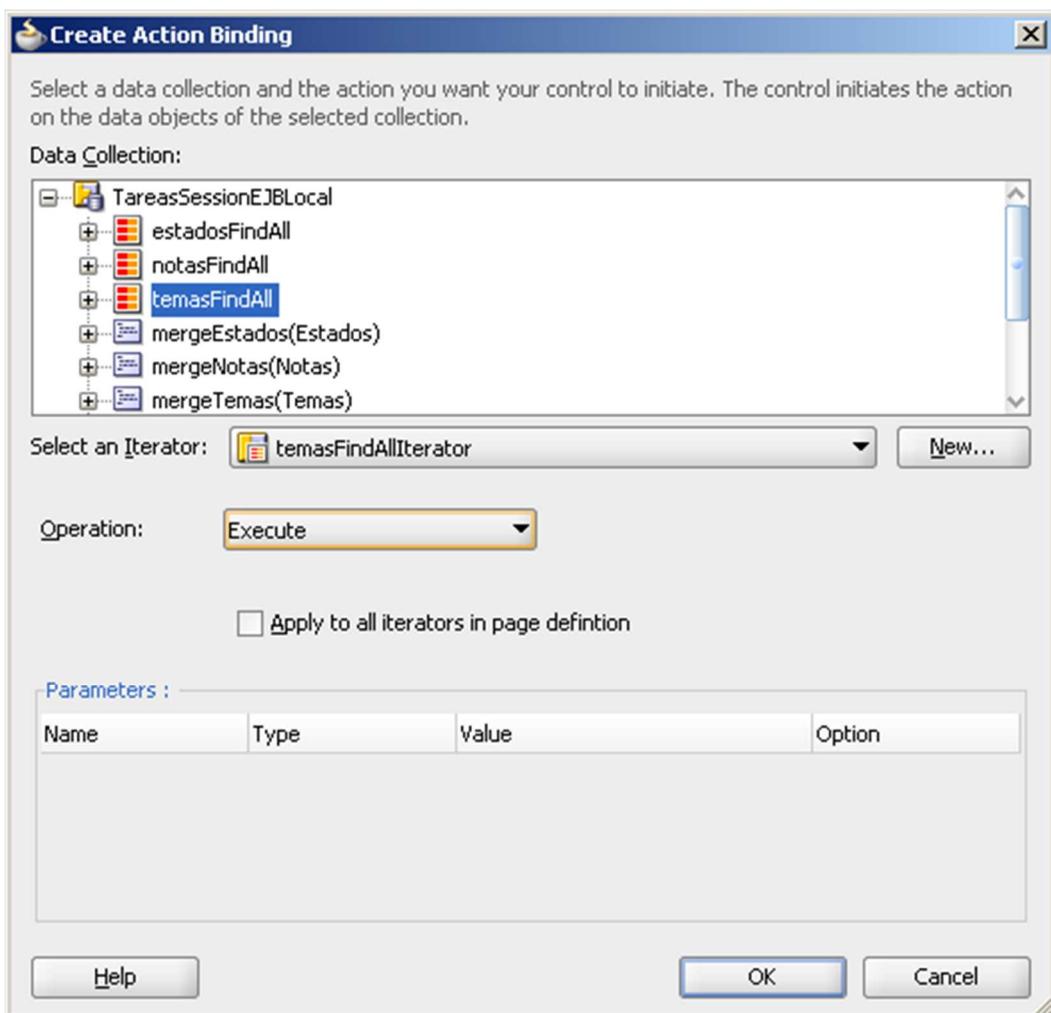


Ilustración 117: Crear enlace de acción

Es importante que comprobemos el valor dado al atributo xmlns. Si es <http://xmlns.oracle.com/adfm/jcuimodel>, tendremos que cambiarlo por <http://xmlns.oracle.com/adfm/uimodel>

Una vez que tenemos creados los enlaces necesarios, desde `yesAction` los invocamos, cerrando antes diálogo. Posteriormente navegamos de vuelta a la página de consulta. Para navegar tras la ejecución de un método no tenemos más que devolver en el mismo una cadena con el nombre de la regla de navegación que queremos usar.

Para obtener una referencia a los distintos elementos que hemos definido en el `pageDef` accedemos al `BindingContext`.

```
public String yesAction() {
    popup.hide();

    BindingContainer bc = BindingContext.getCurrent().getCurrentBindingsEntry();
    bc.getOperationBinding("removeNotas").execute();

    bc.getOperationBinding("Execute").execute();
    return "aConsulta";
}

public void noAction(){
    popup.hide();
}
```

Ilustración 118: Código de yesAction y noAction

En el caso del botón para cancelar la eliminación no tenemos que hacer más que cerrar el diálogo, por lo que directamente podemos invocar al método `noAction` en su `actionListener`.

Sólo nos queda indicar que cuando se acceda al menú *Eliminar nota* se debe mostrar el popup. Arrastramos el componente `showPopupBehavior`, del grupo *Operations*, sobre el *menuItem* correspondiente. En las propiedades indicamos el identificador del popup (por defecto el identificador será `p1`, o algo similar; lo he cambiado por `popup`).

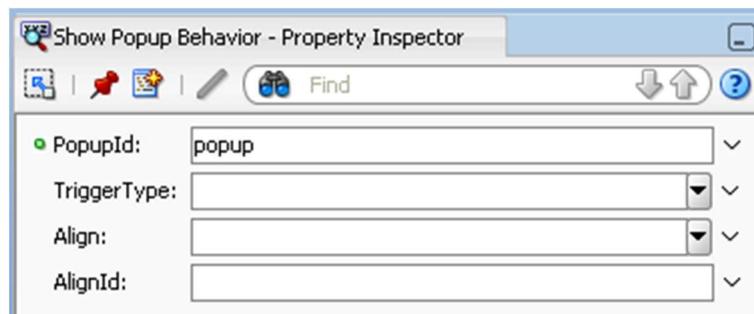


Ilustración 119: Propiedades de showPopupBehavior

De esta forma hemos completado la funcionalidad para eliminar notas. Vamos a pasar ahora a incluir la gestión de estados para completar la aplicación.

Gestión de estados

En este caso vamos a incluir una única página desde la que podremos crear, editar y eliminar los estados, dado que es una entidad muy simple. Procedemos como en los casos anteriores, creando un nuevo menú *Estados*, con un elemento *Gestión de estados*. Arrastramos una nueva página a nuestro `faces-config.xml`, a la que llamamos

editarEstados, y haciendo doble clic sobre ella, la creamos. Definimos una regla de navegación para accederla desde la página de consulta, y la ponemos en el action del nuevo menú creado.

Ya en la página editarEstados, vamos a arrastrar el iterador estadosFindAll del DataControl sobre la vista de diseño, y a elegir la opción *ADF Form*. En este caso, en el diálogo, elegimos únicamente el atributo *estado*, y metemos tanto el botón de submit como los de navegación. De esta forma podremos editar todos los estados registrados en el sistema.

Vamos a crear botones para añadir y eliminar estados, arrastrando los métodos *Create*, del grupo *operations* dentro de *estadosFindAll*, y *removeEstados*. En el primer caso se mostrará un diálogo en el que simplemente tenemos que seleccionar el tipo de componente, *ADF Button*. Para el botón de borrado tendremos que indicar el parámetro (la fila actual de *estadosFindAllIterator*), y modificamos la propiedad *action*, de forma que cuando lo pulsamos se vuelva a la pantalla de consulta.

Lo siguiente es enlazar el botón submit que ya tenemos, arrastrando sobre él el método *mergeEstados*. Como siempre pasaremos como parámetro el estado actual: `#{bindings.estadosFindAllIterator.currentRow.dataProvider}`.

Y por último, añadimos un botón para volver a la página de consulta. Para ello arrastramos la operación *Execute* del iterador *temasFindAll* del DataControl como botón. Una vez creado, modificamos la propiedad *action* a *aConsulta*.

Antes de continuar, podemos cambiar el texto de todos los botones y ponerlos en español: anterior, siguiente, guardar, borrar...

De esta forma tenemos una página en la que podemos ir modificando los estados, y guardando los cambios hechos a cada uno de ellos pulsando el botón *submit*. Cualquier cambio que no haya sido persistido cuando se pulse el botón *volver* será desechado.

Probando la aplicación

Una vez que hemos completado el desarrollo de nuestro programa, podemos lanzarlo para probarlo. Para ello no tenemos más que seleccionar la página inicial (en nuestro caso, *consulta.jspx*), y en el menú contextual seleccionar la opción *Run*. Al hacerlo jDeveloper arrancará el servidor WebLogic que lleva embebido, y desplegará en él la aplicación. Este proceso puede llevar bastante tiempo, y una vez finalizado se abrirá automáticamente nuestro navegador predeterminado mostrando la Web.



Ilustración 120: Aplicación

Puliendo detalles

Con los pasos seguidos hasta el momento tenemos una aplicación completamente operativa. Sin embargo, hay algunos detalles que se pueden mejorar para que su funcionamiento sea más real, y vamos a ir viéndolos.

¿Qué pasa si no hay datos?

Si la base de datos está vacía se producirá un error al presentar la página de consulta debido a que ésta intentará obtener las notas del tema seleccionado, estando la lista de temas vacía. Para evitar este comportamiento le decimos a la tabla de notas que sólo tiene que dibujarse si hay un tema seleccionado.



Sin estados no hay temas

Si empezamos a utilizar la aplicación con la base de datos vacía, podremos acceder al alta de temas sin necesidad de haber creado estados previamente. Esto significa que el selector de estados aparecerá vacío, y se permitirá al usuario crear un tema sin estado (lo que supondrá que se obtenga una excepción si hemos marcado el campo *FK_ESTADO* de la tabla *TEMAS* como no nulo).

Para evitar que el usuario pueda hacer esto, vamos a desactivar el menú *Nuevo tema* cuando la lista de estados esté vacía. Y para ello necesitamos acceder desde la página de consulta a dicha lista, por lo que vamos a crear un nuevo *binding* que nos de acceso.

Abrimos el *pageDef*, y desde la vista estructura, elegimos el grupo *executables*, hacemos clic con el botón derecho, y elegimos *Insert inside executables – accessorIterator*. En el diálogo damos un nombre al iterador, y elegimos la colección de estados, como se muestra.

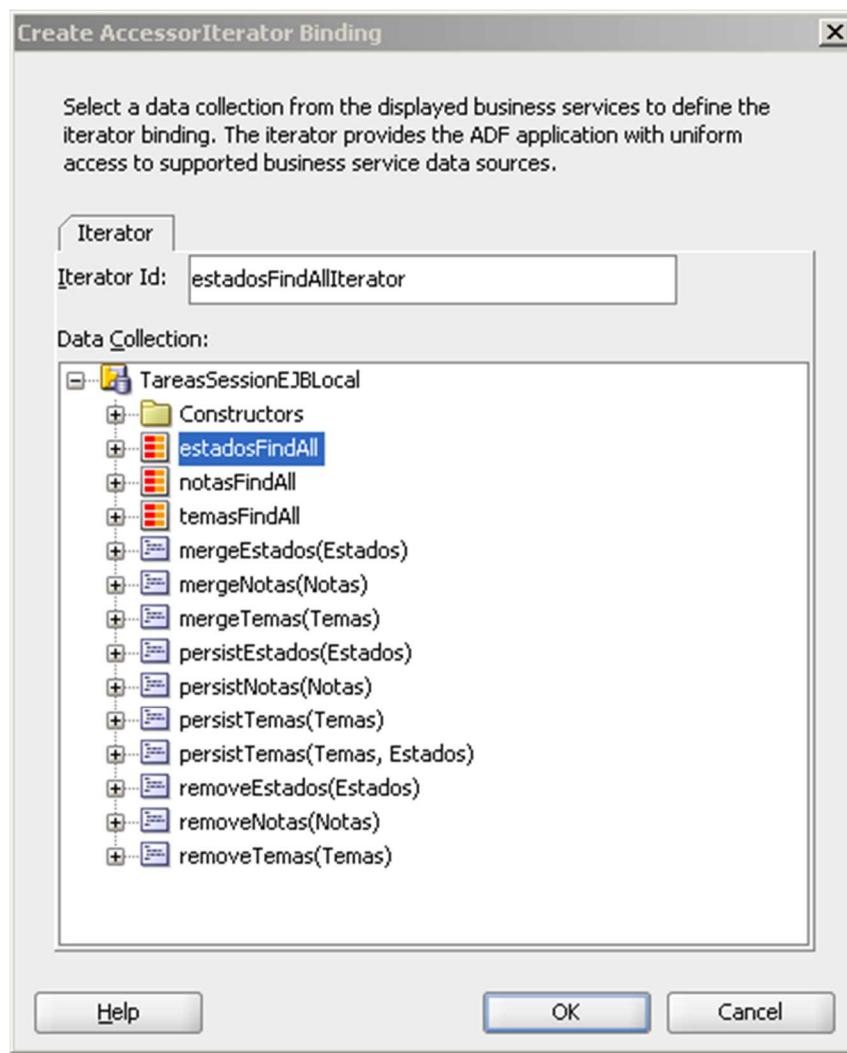


Ilustración 122: Crear AccessorIterator

Ahora que la colección de estados registrados está disponible desde la página de consulta, podemos poner la siguiente expresión en el atributo *disabled* del menú *Nuevo Tema*.

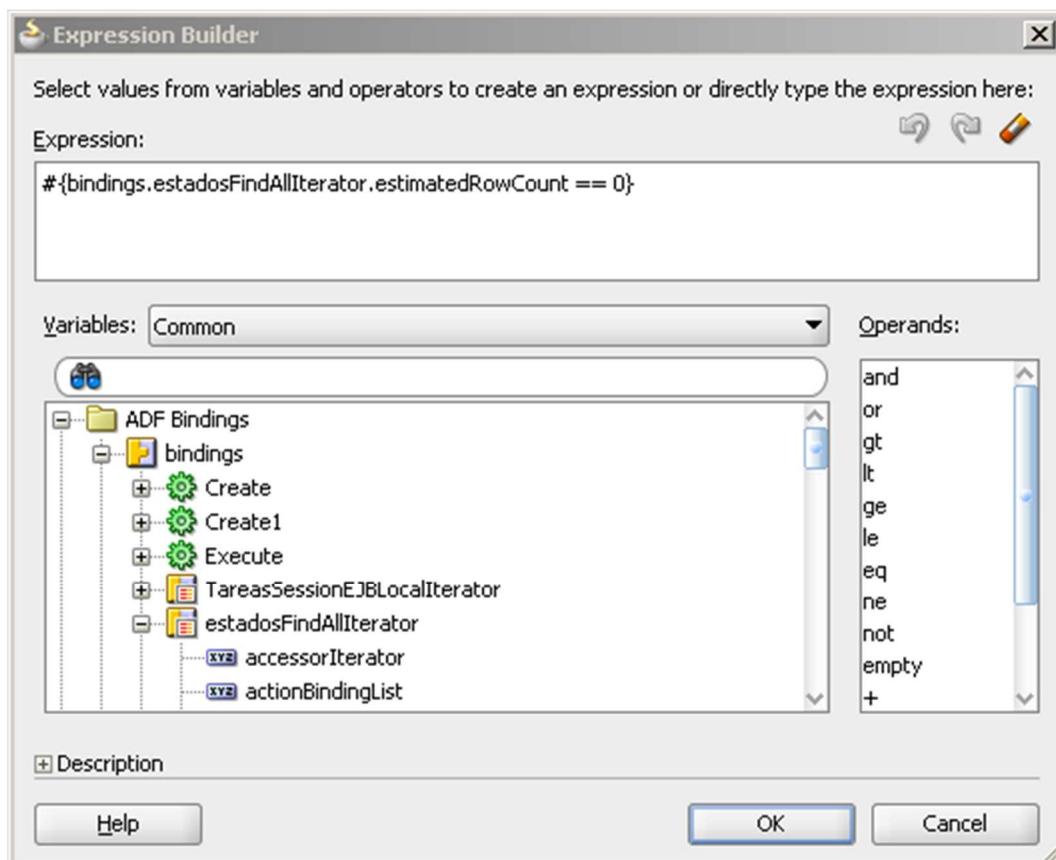


Ilustración 123: Constructor de expresiones

Y así evitamos que los usuarios creen temas si no hay estados en la base de datos.

Error al crear temas con estados nuevos

Si creamos un estado, y posteriormente queremos crear o editar un tema con ese estado, nos encontraremos una excepción *IllegalArgumentException*, que indica que se ha provisto una entidad con clave primaria nula para una operación de búsqueda.

Para este error vamos a dar una solución que nos llevará a crear una nueva *namedQuery*, de forma que veamos cómo se trabaja con ellas.

La excepción es lanzada en el EJB de sesión, en los métodos *persistTemas* y *mergeTemas*. Anteriormente los modificamos para hacer una búsqueda del estado que se va a asignar al tema mediante el método *find*, filtrando por su *sid*. Sin embargo no tuvimos en cuenta la posibilidad de que el campo *sid* del estado fuese nulo. En este caso sucede porque estamos utilizando una entidad recién creada, que no recogió ese valor tras persistirse.

Lo que vamos a hacer es comprobar si el campo *sid* está informado para la entidad; si es así, seguimos haciendo la búsqueda como hasta ahora; si no, haremos la búsqueda por el campo *estado* mediante una *namedQuery*.

Para crear la *namedQuery* tenemos que ir a la entidad *Estados*, y modificar la anotación de la cabecera de la clase. Incluimos una nueva línea con la consulta. Los parámetros se identifican poniendo dos puntos delante.

```
@Entity
@NamedQueries({
    @NamedQuery(name = "Estados.findAll", query = "select o from Estados o"),
    @NamedQuery(name = "Estados.findByEstado", query = "select o from Estados o where o.estado = :estado")
})
```

Ilustración 124: Código de named query

Una vez definida la consulta, podemos utilizarla desde el código del EJB, como se muestra.

```
public Temas persistTemas(Temas temas, Estados estados) {
    if (estados.getSid() != null) {
        estados = em.find(Estados.class, estados.getSid());
    } else {
        Query query = em.createNamedQuery("Estados.findByEstado");
        query.setParameter("estado", estados.getEstado());
        estados = (Estados)query.getSingleResult();
    }
    temas.setEstados(estados);
    estados.addTemas(temas);
    em.persist(temas);
    return temas;
}
```

Ilustración 125: Código de persistTemas

Así asignaremos el estado que le corresponde al tema, independientemente de si su campo *sid* está informado en la entidad.

Exactamente la misma modificación tenemos que hacerla en el método *mergeTemas*.

```
public Temas mergeTemas(Temas temas) {
    Estados estados = temas.getEstados();
    if (estados.getSid() != null) {
        estados = em.find(Estados.class, estados.getSid());
    } else {
        Query query = em.createNamedQuery("Estados.findByEstado");
        query.setParameter("estado", estados.getEstado());
        estados = (Estados)query.getSingleResult();
    }
    temas.setEstados(estados);
    return em.merge(temas);
}
```

Ilustración 126: Código de mergeTemas

Edición de temas

Lógicamente, como hemos hecho con las notas, sólo podremos acceder a la edición de temas si hay uno seleccionado en la página. Cuando no haya ninguno registrado no habrá selección, y por tanto el menú debería estar deshabilitado.

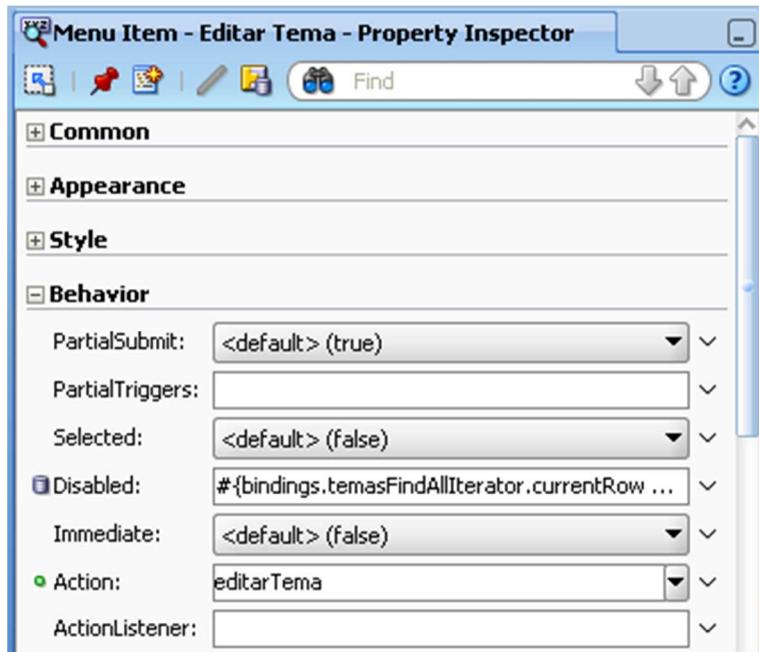


Ilustración 127: Propiedades de elemento menú

Error al borrar estados

La aplicación no realiza ningún tipo de control cuando un usuario intenta borrar un estado. Sin embargo, dado que en la base de datos tenemos una referencia desde la tabla *Temas* hacia la tabla *Estados*, no podremos eliminar registros de ésta última que estén siendo referenciados desde la primera. Si lo intentamos, se mostrará el error producido por la base de datos en el log de jDeveloper.

```
Caused by: java.sql.SQLIntegrityConstraintViolationException: ORA-02292: restricción de integridad (TAREAS.TEMAS_ESTADOS_FK1) violada - registro secundario encontrado
    at oracle.jdbc.driver.SQLStateMapping.newSQLException(SQLStateMapping.java:85)
    at oracle.jdbc.driver.DatabaseError.newSQLException(DatabaseError.java:133)
    at oracle.jdbc.driver.DatabaseError.throwSqlException(DatabaseError.java:206)
    at oracle.jdbc.driver.T4CTTIoer.processError(T4CTTIoer.java:455)
    at oracle.jdbc.driver.T4CTTIoer.processError(T4CTTIoer.java:413)
    at oracle.jdbc.driver.T4C8Oall.receive(T4C8Oall.java:1035)
    at oracle.jdbc.driver.T4CPPreparedStatement.doGetLast(T4CPPreparedStatement.java:194)
    at oracle.jdbc.driver.T4CPPreparedStatement.executeForRows(T4CPPreparedStatement.java:953)
    at oracle.jdbc.driver.OraclePreparedStatement.executeUpdateWithTimeout(OraclePreparedStatement.java:1224)
    at oracle.jdbc.driver.OraclePreparedStatement.executeInternal(OraclePreparedStatement.java:3386)
    at oracle.jdbc.driver.OraclePreparedStatement.executeUpdate(OraclePreparedStatement.java:3467)
    at oracle.jdbc.driver.OraclePreparedStatementWrapper.executeUpdate(OraclePreparedStatementWrapper.java:1350)
    at weblogic.jdbc.wrapper.PreparedStatement.executeUpdate(PreparedStatement.java:172)
    at org.eclipse.persistence.internal.databaseaccess.DatabaseAccessor.executeDirectNoSelect(DatabaseAccessor.java:792)
    ... 104 more
```

Ilustración 128: Excepción ORA-02292

El comportamiento de la aplicación es el que realmente queremos, ya que el borrado no se lleva a cabo, pero lo correcto sería informar al usuario del porqué. Así que vamos a ver cómo mostrar un mensaje de error personalizado.

Importante: Lo explicado en este apartado no funciona en la versión 11.1.1.3 de jDeveloper. Hasta la versión 11.1.1.2, cuando se produce un error en tiempo de ejecución, éste es mostrado al usuario mediante un diálogo, además de escrito en el log del IDE. Sin embargo, cuando trabajamos con la versión 11.1.1.3 no sucede así.

Lo primero que tenemos que hacer es crear una clase que extienda de `DCErrorHandlerImpl`. Accedemos al menú `New` dentro del proyecto `ViewController`, y en la sección `General` elegimos `Java Class`. En el diálogo indicamos el nombre, paquete, y clase padre.

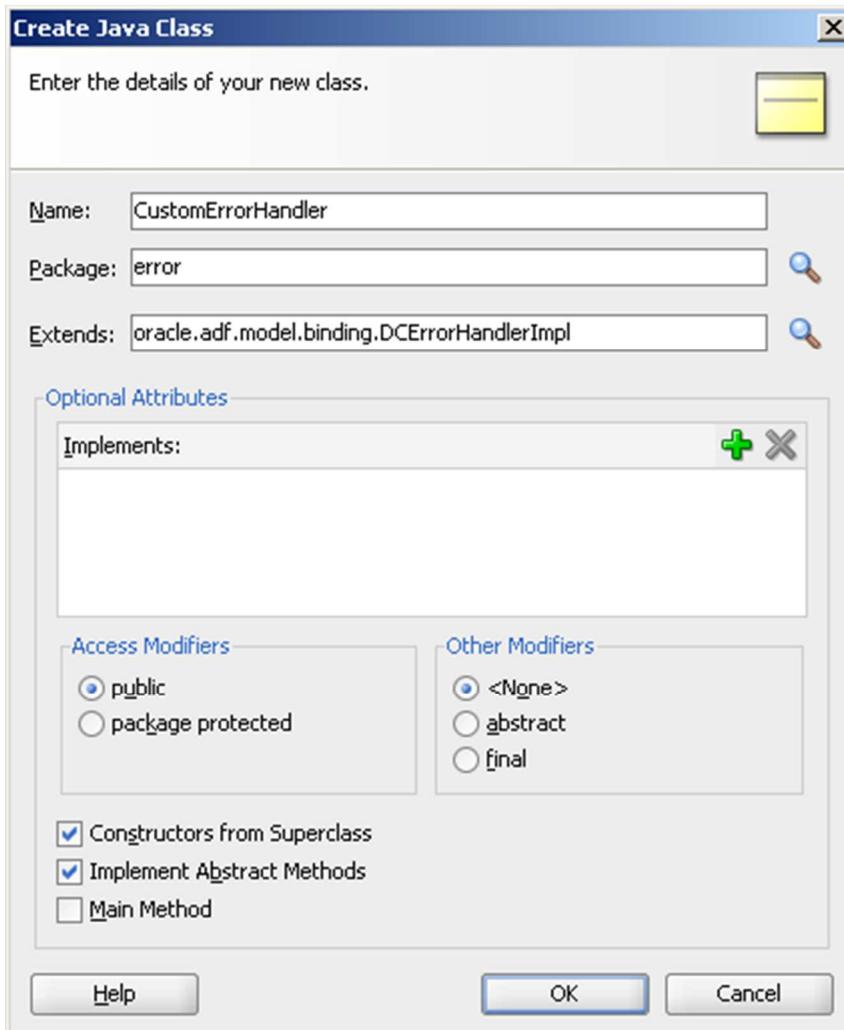


Ilustración 129: Crear clase java

El IDE crea la estructura de la clase, que completamos con el siguiente código.

```
package error;

import com.sun.jndi.cosnaming.ExceptionMapper;
import java.util.ArrayList;
import java.util.List;
import oracle.adf.model.BindingContext;
import oracle.adf.model.RegionBinding;
import oracle.adf.model.binding.DCBindingContainer;
import oracle.adf.model.binding.DCErrorHandlerImpl;
import oracle.adf.model.binding.DCErrorMessage;

public class CustomErrorHandler extends DCErrorHandlerImpl {

    List<ExceptionMapper> exceptionMapperList = new ArrayList<ExceptionMapper>();

    public CustomErrorHandler(boolean b) {
        super(b);
    }

    public CustomErrorHandler() {
        this(true);
    }

    public void reportException(DCBindingContainer bc, Exception ex) {
        super.reportException(bc, ex);
    }

    public DCErrorMessage getDetailedDisplayMessage(BindingContext ctx,
                                                    RegionBinding ctr,
                                                    Exception ex) {
        return new MyDCErrorMessage(ctr, ex);
    }
}
```

Ilustración 130: Código de CustomErrorHandler

Lo que hace esta clase, básicamente, es sobrescribir el método *getDetailedDisplayMessage*, que es el que el framework utiliza para obtener el mensaje de error que se muestra al usuario. El mensaje de error se representa mediante una clase hija de *DCErrorMessage*. Esta clase tendrá un método *getText*, cuyo resultado es el que se mostrará por pantalla.

En el código de *getDetailedDisplayMessage* lo que hacemos es crear una instancia de la clase *MyDCErrorMessage*, que será nuestra propia clase de error. Como todavía no existe, el IDE la subraya en rojo. Si pulsamos sobre la bombilla roja que podemos ver a la izquierda del código, tenemos la opción de crearla.

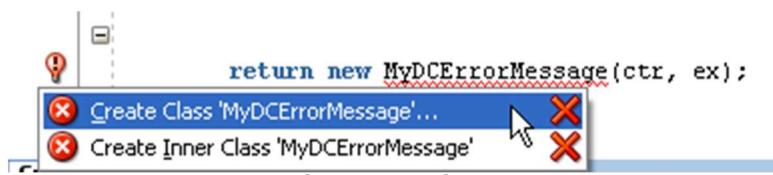


Ilustración 131: Menú crear clase

Se abre un diálogo para indicar el paquete en que queremos situarla. Elegimos *error*, y aceptamos. Podemos ahora abrir la clase, y darle el siguiente código.

```
package error;

import oracle.adf.model.RegionBinding;
import oracle.adf.model.binding.DCErrorMessage;

public class MyDCErrorMessage implements DCErrorMessage {

    RegionBinding m_regionBinding;
    Exception m_ex;

    public MyDCErrorMessage(RegionBinding ctr, Exception ex) {
        super();
        this.m_regionBinding = ctr;
        this.m_ex = ex;
    }

    public String getText() {
        return m_ex.getMessage();
    }

    public String getHTMLText() {
        if (m_ex.getMessage().contains("ORA-02292")){
            return "No se puede eliminar un estado asignado a algún tema.";
        }
        else{
            return m_ex.getMessage();
        }
    }
}
```

Ilustración 132: Código de MyDCErrorMessage

Vemos que lo único que se ha hecho es guardar la excepción, y utilizarla en los métodos `getText` para devolver el mensaje de error. Cuando este mensaje contenga la cadena ORA-02292, que es el código Oracle de nuestro error, devolveremos un mensaje informando al usuario de que no puede eliminar estados asignados a temas.

Sólo nos queda decirle al framework que utilice las clases que hemos creado como manejadoras de errores. Lo hacemos abriendo el fichero `DataBinding.cpx`, y seleccionando la raíz en la vista estructura. Entre las propiedades encontramos `ErrorHandlerClass`. Seleccionamos `CustomErrorHandler`.

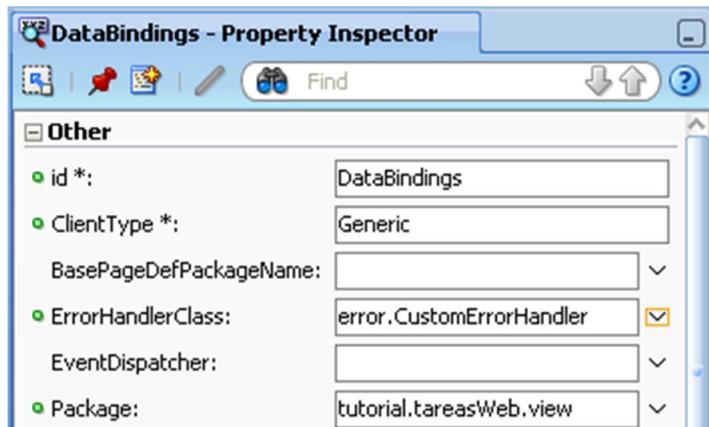


Ilustración 133: Propiedades de DataBindings

Presentación de las tablas

La tabla maestra de la página de consulta, en la que se muestran los temas registrados en el sistema, no se ve completa debido a que está situada en la primera posición de un panel dividido, cuya anchura no es suficiente. El usuario puede modificar el tamaño del panel en ejecución mostrando todas las columnas. Sin embargo, vamos a dar un valor específico a la propiedad *SplitterPosition*, de forma que se vean todos los datos desde el primer momento.

No tenemos más que abrir la página *consulta.jspx*, seleccionar desde la vista diseño o estructura el componente *panelSplitter*, y dar el valor 430 a la propiedad *SplitterPosition*.



Ilustración 134: Propiedad splitterPosition

Por otro lado nos encontramos que la anchura del campo *nota*, de la segunda tabla, es también demasiado pequeña. Lo normal es que las notas contengan un texto largo, por lo que vamos a aprovechar al máximo el tamaño de la pantalla, fijando una anchura para la columna.



Ilustración 135: Propiedad width

Con estos cambios la presentación de la pantalla mejora notablemente.



Ilustración 136: Aplicación

Fecha por defecto

Lo más normal es que cuando vayamos a introducir una nota lo hagamos con la fecha del día, por lo que vamos a dar ese valor por defecto al campo *fecha* de la página *nuevaNota*.

Para ello lo que tenemos que hacer realmente es modificar el constructor de la entidad *Nota*. En él haremos que se asigne al campo el valor de *sysdate*.

Conclusiones

En este tutorial hemos visto cómo desarrollar una pequeña aplicación Web utilizando la suite de desarrollo de Oracle 11. En el centro de esta suite se sitúa el framework Oracle ADF, que proporciona la base para desarrollar aplicaciones Java, tanto en el lado del cliente como del servidor.

Aunque no está entre los frameworks más conocidos y utilizados, ADF es una opción muy recomendable, y tiene grandes ventajas, como la posibilidad de elegir casi

cualquier combinación de tecnologías a la hora de desarrollar una aplicación: Swing + ADF BC; JSF + JPA; JSP + Struts + Web Services...

Oracle ofrece además jDeveloper como IDE, proporcionando las herramientas necesarias para trabajar con estas tecnologías de forma sencilla, y cubriendo todos los pasos del proceso de desarrollo, desde el modelado hasta la codificación y pruebas.

Referencias

Información sobre ADF

- http://es.wikipedia.org/wiki/Oracle_Application_Development_Framework
- http://www.oracle.com/technology/products/adf/pdf/ADF_11_overview.pdf
- <http://easyadf.blogspot.com/>
- Descripción de los ficheros xml de ADF:
http://docs.tpu.ru/docs/oracle/en/owl/E14571_01/web.1111/b31974/appendixa.htm
- Personalizar los mensajes de error:
http://download.oracle.com/docs/cd/E12839_01/web.1111/b31974/web_adv.htm#CIHHBEEJ

Información sobre EJB 3.0

- <http://java.sun.com/products/ejb/>
- Pro EJB 3, Java Persistence API, Mike Keith and Merrick Schincariol.
http://es.wikipedia.org/wiki/Enterprise_JavaBeans
- <http://java.sun.com/developer/technicalArticles/J2EE/jpa/>

Tutoriales de jDeveloper

- <http://www.oracle.com/technology/products/jdev/11/cuecards111/index.html#adf>

Descarga de jDeveloper

- <http://www.oracle.com/technetwork/developer-tools/jdev/downloads/index.html>

Anexo: Abrir una aplicación existente con jDeveloper

Si disponemos de un módulo de aplicación desarrollado con jDeveloper, y queremos cargarlo en el entorno, podemos hacerlo siguiendo este sencillo proceso.

Si no tenemos ninguna aplicación abierta actualmente, veremos directamente sobre el navegador de aplicaciones dos opciones, una para crear una nueva aplicación, y otra para abrir una existente.

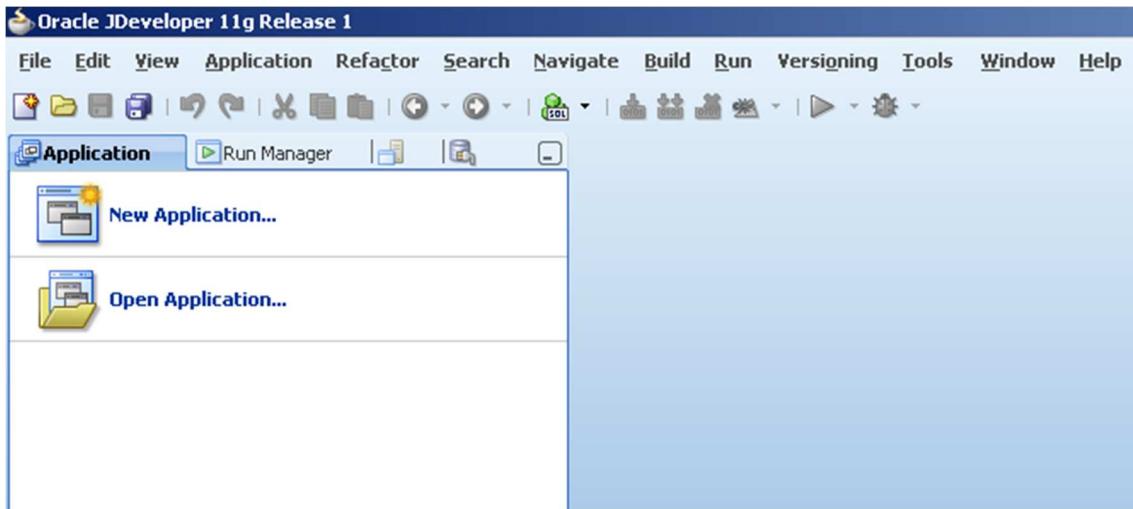


Ilustración 137: Open Application

En caso de que ya tuviésemos aplicaciones abiertas, esta opción la podemos encontrar también en el combo del propio navegador de aplicaciones.

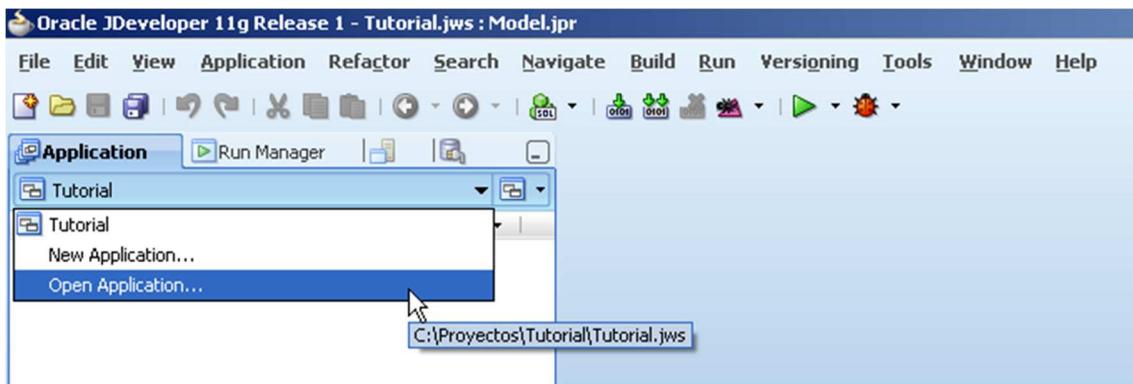


Ilustración 138: Menú Open Application

En cualquiera de los dos casos accedemos a un diálogo en el que podemos seleccionar un fichero de espacio de trabajo de jDeveloper (extensión *jws*). Tras elegirlo la aplicación se abrirá en el IDE.

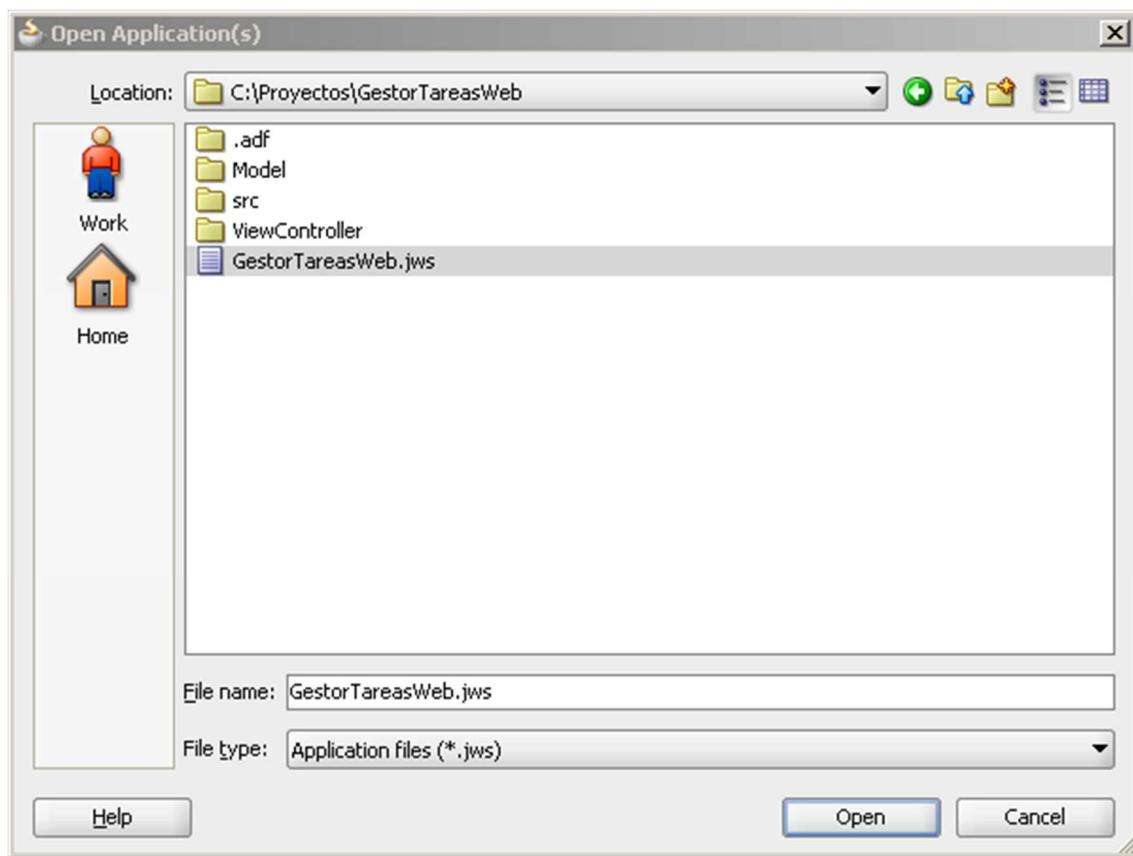


Ilustración 139: Selección del fichero jws