**Proyecto PIS 2013 Grupo 10**

**DUSA**

**Documentación Técnica**

**Versión 1.0**

**Historia de revisiones**

|  |  |  |  |
| --- | --- | --- | --- |
| Fecha | Versión | Descripción | Autor |
| 24/11/2013 | 1.0 | Creación del documento | Elisa Sanguinetti |
| 24/11/2013 | 1.0 | Revisión del documento | Carlos Vélez |
|  |  |  |  |
|  |  |  |  |

Contenido

[1. Objetivo 3](#_Toc373149477)

[2. JPA 3](#_Toc373149478)

[2.1. PERSISTENCE.XML 3](#_Toc373149479)

[2.2. CAMBIOS A LA BASE DE DATOS DESDE JPA 4](#_Toc373149480)

[2.3. AGREGAR NUEVAS TABLAS AL MODELO 4](#_Toc373149481)

[2.4. AGREGAR CAMPOS A UNA TABLA 5](#_Toc373149482)

[2.5. TABLAS CON CLAVE PRIMARIA DE UN SOLO CAMPO 5](#_Toc373149483)

[2.6. TABLAS CON CLAVE PRIMARIA MULTI-CAMPO 5](#_Toc373149489)

[2.7. AGREGAR RELACIONES ENTRE TABLAS 6](#_Toc373149496)

[2.8. CAMPOS QUE PERTENECEN A CLAVES PRIMARIAS Y FORÁNEAS 8](#_Toc373149497)

[2.9. CONTADORES AUTO-INCREMENTALES 8](#_Toc373149498)

[3. SCM – Cómo Generar archivos ejecutables para la aplicación 9](#_Toc373149499)

[4. MOTOR DE BÚSQUEDA 22](#_Toc373149500)

1. Objetivo

El objetivo de este documento es el de unificar toda la documentación técnica del sistema.

1. JPA

## PERSISTENCE.XML

Este es el archivo de configuración para JPA. Contiene, entre otras cosas:

* Las entidades que se utilizan: anotadas entre las etiquetas **<class>** y **</class>**

Ejemplo:

**<class>com.dusa.modelo.Farmacia</class>**

* Propiedades de conexión a la base de datos; como string de conexión, usuario y contraseña.
* Propiedades que tienen como resultado la modificación de la base de datos según los cambios realizados en el modelo JPA.
* Si se quieren hacer modificaciones sobre la base, del tipo agregar entidades, se deben agregar las siguientes propiedades y correr una vez la aplicación.

<property name="eclipselink.ddl-generation" value="create-tables" />

<property name="eclipselink.ddl-generation.output-mode" value="database" />

<property name="eclipselink.create-ddl-jdbc-file-name" value="createDDL\_ddlGeneration.jdbc"/>

<property name="eclipselink.drop-ddl-jdbc-file-name" value="dropDDL\_ddlGeneration.jdbc"/>

La clase **com.dusa.utils.CrearPersistence** se utiliza para re-generar este archivo de configuración según los parámetros del archivo **config.properties**. Se ejecuta de la siguiente forma:

**CrearPersistence persistance = new CrearPersistence();**

**persistance.CrearPersistenceXml();**

Los parámetros que se toman del archivo config.properties son los siguientes:

* **dataBaseUser**: Nombre de usuario.
* **dataBasePassword**: Contraseña.
* **createTable**: Booleano que indica si se deben crear las tablas o no.

## CAMBIOS A LA BASE DE DATOS DESDE JPA

Para realizar cambios desde el código de la aplicación (en atributos, entidades ,tipos) y que estos impacten en la base de datos, debe ejecutarse la aplicación habiendo configurado el archivo PERSISTENCE.XML para creación de tablas (ver descripción del archivo PERSISTENCE.XML). Las tablas que se desean modificar deben borrarse manualmente previo a la ejecución.

## AGREGAR NUEVAS TABLAS AL MODELO

Las clases que están relacionadas con tablas de las base de datos se denominan entidades.

Se utilizan las anotaciones **@Entity** y **@Table** para definirlas. El parámetro **name** de la anotación **@Table** sirve para indicar el nombre de la tabla a la que se referencia. Ejemplo:

**@Entity**

**@Table(name="WEBPERMISOS")**

**public class Permiso implements Serializable {**

**(...)**

**}**

Además de agregar la entidad, es necesario agregar la referencia a la clase en el archivo **PERSISTENCE.XML** (como ya se explicó en la descripción de este archivo).

## AGREGAR CAMPOS A UNA TABLA

Para campos que no pertenecen a ninguna clave primaria o foránea, se debe agregar el atributo a la clase correspondiente bajo la anotación **@Column**.

Ejemplo:

**@Column(name = "CANTIDAD\_WEB", nullable = false)**

**private int cantWeb;**

Notas:

* El atributo "**name**" de la anotación **@Column** es el nombre del campo en la base de datos, y el atributo "nullable" indica si el campo puede recibir valores nulos.
* Casos de campos que pertenecen a claves foráneas son estudiados en la sección "Agregar relaciones entre tablas" de este documento.
* Para agregar campos a la clave primaria de una tabla, ver sección "Claves con clave primaria multi-campo" de este documento.

## TABLAS CON CLAVE PRIMARIA DE UN SOLO CAMPO

Se agrega el campo y la anotación **@Id**. Ejemplo de la clase Permiso:

**@Id**

**@Column(name = "IDPERMISO", nullable = false)**

**private String idPermiso;**



## TABLAS CON CLAVE PRIMARIA MULTI-CAMPO

Para lasclaves primarias compuestas por más de un campo, además de crear la entidad para la tabla, se debe crear una nueva clase que alojará solo a los campos la clave. La clase para la clave no tiene la anotación **@Entity**, por el contrario, se anota con **@Embeddable**. Desde la entidad se referencia a la clave primaria con la anotación **@EmbeddedId**.

Ejemplo de las clase PedidoLinea y su id PedidoLineaPK:

* En PedidoLinea:

**@EmbeddedId**

**private PedidoLineaPK pedidoLineaPK;**

* En PedidoLineaPK:

**@Embeddable**

**public class PedidoLineaPK implements Serializable{**

**@Column(name = "IDPEDIDO")**

**private int idPedido;**

**(...)**

**}**



## AGREGAR RELACIONES ENTRE TABLAS

Para cada tabla, se distinguen los siguientes casos según la cardinalidad de la relación:

* + 1. **Relación "uno a muchos"**

Para este tipo de relaciones, a nivel de base de datos no se agregan los campos en la tabla que estamos tratando (el lado con cardinalidad uno). De todas formas, JPA permite agregar referencias en ambos lados de la relación para poder navegar bidireccionalmente. Como veremos en los ejemplos, hay que definir explícitamente los campos de join por los que se establece la relación, que deben coincidir con la clave primaria de la tabla con cardinalidad uno.

Ejemplo:

* Desde la clase Pedido se referencia a una lista de la clase BackOrderLinea:

**@OneToMany ()**

**@JoinColumn(name="IDPEDIDO")**

**private List<BackOrderLinea> lineasBackOrder;**

* Para casos de join por varios campos se anota de la siguiente manera:

**@OneToMany**

**@JoinColumns({**

**@JoinColumn(name="REPARTO",referencedColumnName = "REPARTO"),**

**@JoinColumn(name="NRO\_CLIENTE",referencedColumnName = "NRO\_CLIENTE")**

**})**

**private List<Reparto> repartos;**

* + 1. **Relación "muchos a uno"**

Los únicos cambios con respecto al caso anterior son que se utiliza la anotación **@ManyToOne**, y que en lugar de una lista se utiliza una única instancia como referencia.

Ejemplo de la clase PedidoLinea en relación con su respectivo Pedido:

**@ManyToOne ()**

**@JoinColumn(name="IDPEDIDO",referencedColumnName = "IDPEDIDO")**

**private Pedido pedido;**

Este cambio agregaría, de no estar presente, el campo IDPEDIDO en la tabla WEBPEDIDOLINEAS.

Nota: El parámetro **referencedColumnName** de la anotación **@JoinColumn** sirve para indicar el nombre del campo referenciado en el otro lado de la relación. En el caso de que el campo se llame igual en ambos extremos, el parámetro es opcional.

* + 1. **Relación "muchos a muchos"**

Para este caso, a nivel de base de datos se utiliza una tabla intermedia para establecer la relación, y no se agregan campos en las tablas en ninguno de los extremos.

El siguiente es un ejemplo que muestra cómo se referencian los permisos que tiene un rol determinado:

**@ManyToMany**

**@JoinTable(**

**name="WEBROLESPERMISOS",**

**joinColumns={@JoinColumn(name="IDROL", referencedColumnName="IDROL")},**

**inverseJoinColumns={@JoinColumn(name="IDPERMISO", referencedColumnName="IDPERMISO")}**

**)**

**private List<Permiso> permisos;**

Vemos que se define explícitamente el nombre de la tabla intermedia WEBROLESPERMISOS, a la que no es obligatorio asociarle un entidad.

Notas:

* El atributo **inverseJoinColumns** indica los campos de join para el otro extremo de la relación.
* En este caso en particular sí se agregó una entidad para la tabla intermedia(ver clase RolesPermisos).
* Es recomendable que JPA genere la tabla intermedia en lugar de crearla manualmente.
  + 1. **Relación "uno a uno"**

Se utiliza la anotación **@OneToOne** y se referencia únicamente a una instancia para el otro extremo de la relación (al igual que el caso ManyToOne).

Ejemplo:

**@OneToOne ()**

**@JoinColumn(name="IDPERSONA",referencedColumnName = "IDPERSONA")**

**private Persona persona;**

## CAMPOS QUE PERTENECEN A CLAVES PRIMARIAS Y FORÁNEAS

Este caso no representa un problema si la clave primaria está formada por un solo campo, pero en cualquier otro caso tiene alguna dificultad adicional.

Un ejemplo es el caso del atributo idPedido de la entidad PedidoLinea. Es utilizado como campo de join para relacionarse con el pedido:

**@JoinColumn(name="IDPEDIDO", referencedColumnName = "IDPEDIDO")**

**private Pedido pedido;**

Y a su vez forma parte de los atributos incluidos en la clase PedidoLineaPK que aloja a una clave de múltiples campos:

**@Column(name = "IDPEDIDO",nullable = false, updatable = false, insertable = false)**

**private int idPedido;**

Una de las dos apariciones del campo debe tener evaluados en false los atributos "updatable" e "insertable" de la anotación **@Column**. En el ejemplo puede verse que esto se hizo en la aparición del campo en la clase PedidoLineaPK.

## CONTADORES AUTO-INCREMENTALES

Ejemplo de definición de atributos con valores auto-incrementales:

**@Id**

**@Column(name = "IDPEDIDO", nullable = false)**

**@GeneratedValue(generator="GeneradorIdPedido")**

**@SequenceGenerator(name="GeneradorIdPedido",sequenceName="SeqWebPedido)**

**private idPedido;**

Las anotaciones **@GeneratedValue** y **@SequenceGenerator** sirven para crear una secuencia con valores autogenerados (autosecuencias). El atributo "**sequenceName**" es el nombre de la secuencia que será creada (si aún no existe) o que será utilizada (para secuencias existentes). El valor de los atributos "**generator**" y "**name**" de una y otra etiqueta deben coincidir como en el ejemplo.

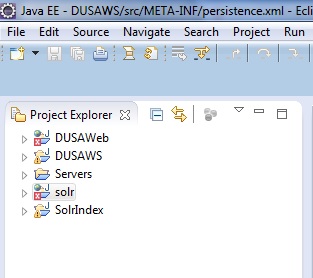
1. SCM – Cómo Generar archivos ejecutables para la aplicación
   1. Propósito

La creación de archivos ejecutables, permite ejecutar el sistema sin necesitad de un IDE. El mismo sólo es necesario para poder generar dichos archivos.

En las siguientes secciones se explica cómo generar los archivos ejecutables desde Eclipse.

* 1. Puesta a punto

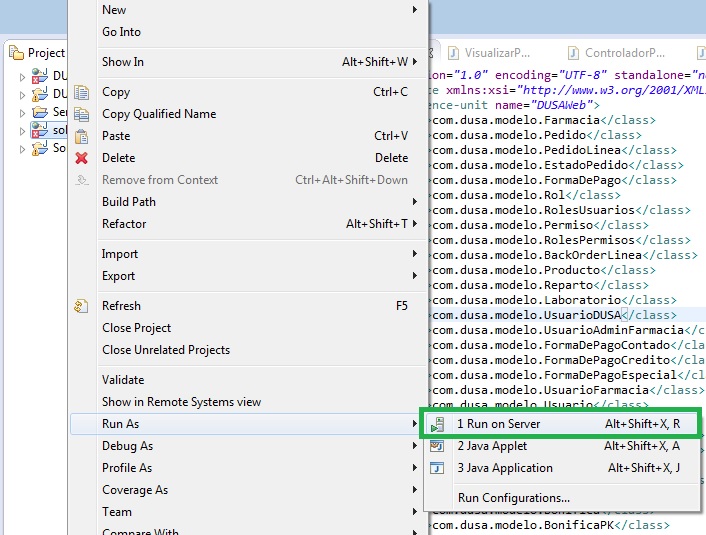
Existen tres proyectos que deben ser ejecutados en conjunto para el correcto funcionamiento de la aplicación. Estos son: DUSAWeb (Proyecto Web), DUSAWS (Lógica de la aplicación) y solr (Motor de Búsqueda).



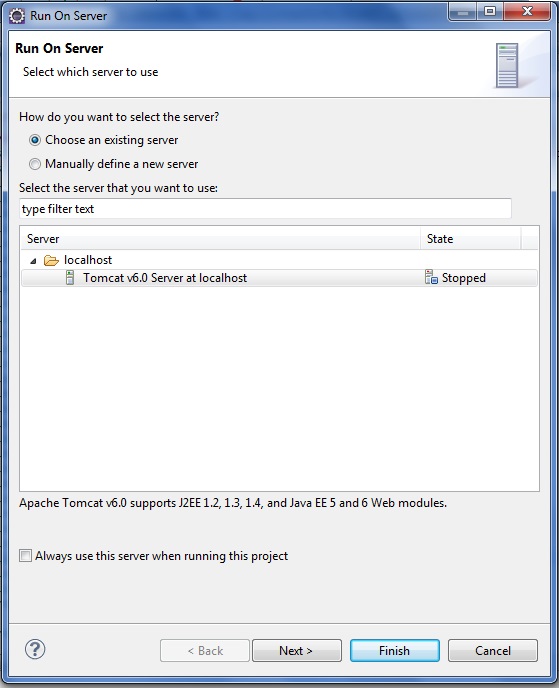
* + 1. Solr

Lo primero que se debe hacer una vez abierto el Eclipse es ejecutar el proyecto de nombre *solr*, para ello, hacer clic con el botón derecho del mouse sobre el proyecto, seleccionar del menú la opción *Run As* y del submenú, elegir *Run on Server.*

Ver imagen presentada a continuación:



Nos aparecerá la siguiente ventana:



Seleccionar el servidor Tomcat v6.0 y presionar botón *Finish*.

* + 1. DUSAWS

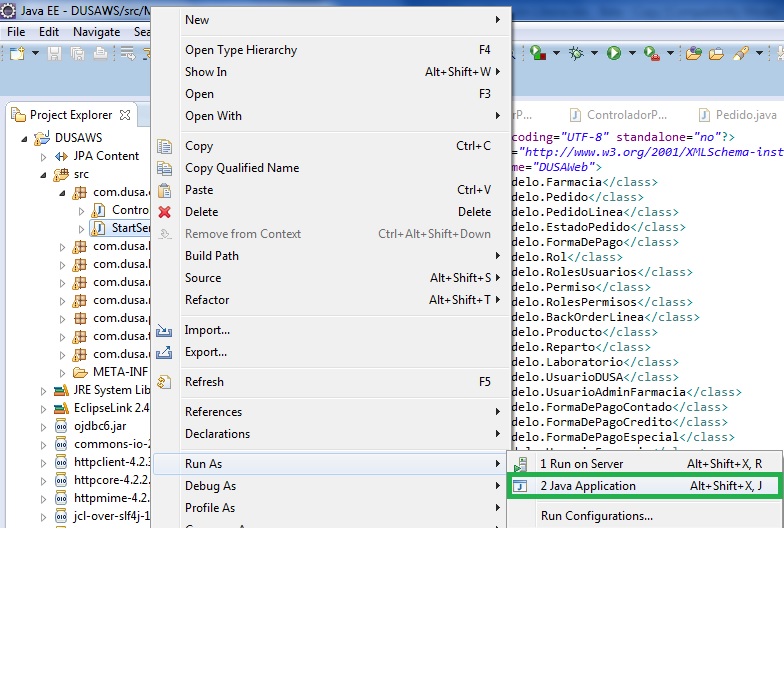
En segundo lugar, se debe ejecutar el proyecto que contiene la lógica de la aplicación, de esta manera se publicarán los web services, una vez publicados, podremos generar los archivos de comunicación en el proyecto web (DUSAWeb), que serán los que consuman dichos web services.

Abrir el proyecto, ubicar la carpeta scr y adentro de ésta un paquete llamado com.dusa.comunicacion, el cual contiene dos archivos: ControladorComunicacion.java y StartService.java.

Al posicionarse en el archivo StartService.java, y hacer click con el botón derecho del mouse, del menú se selecciona *Run As y* del submenú, se elige la opción *Java Application.*

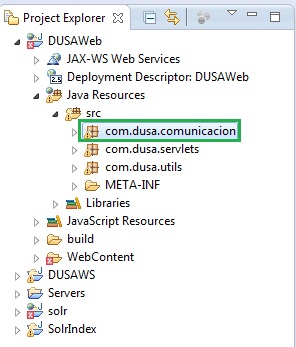
El archivo StartService.java además de publicar web services, ejecuta dos aplicaciones una para realizar indexado y otra para guardar pedidos automáticos cada 5 minutos (TareaGuardaPedido).

Ver imagen presentada a continuación:



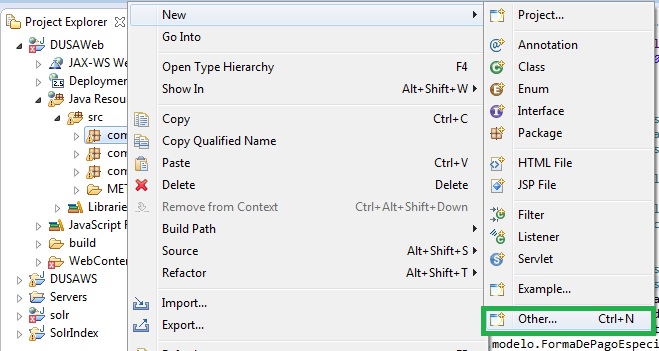
* + 1. DUSAWeb

Por último, antes de correr el proyecto web, hay que generar los archivos de comunicación que se encuentran dentro del proyecto:

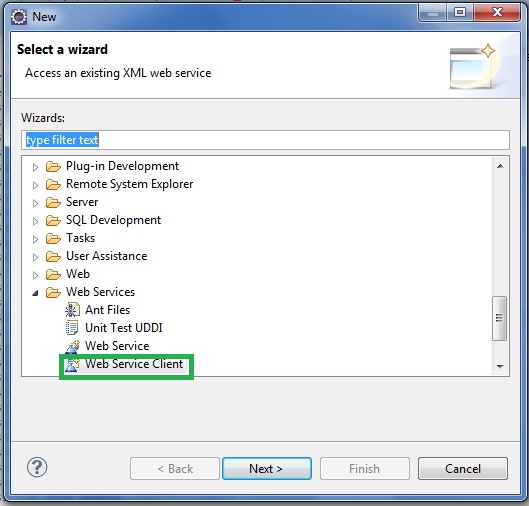


Primero hay que borrar todos los archivos que estén en esa carpeta, abrimos la carpeta, los seleccionamos todos y apretamos la tecla suprimir.

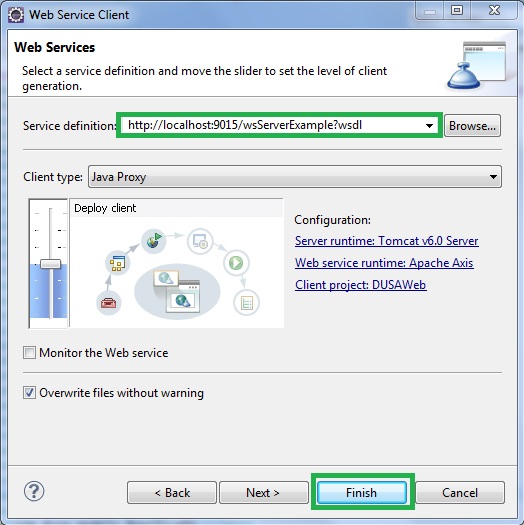
Luego de borrarlos, hay que generarlos, para ello hacemos clic derecho sobre la carpeta comunicación, seleccionamos *New* y del sub menú; *Other…*

**

Nos aparece una ventana y buscamos la opción *Web Service Client:*



Una vez seleccionada la opción hacemos clic en *Finish* y nos aparecerá la siguiente ventana:



Completamos el campo *Service definition:* como indica la imagen.

La url donde está publicando el servidor los web services está definida dentro del archivo StartService.java que se encuentra dentro del proyecto DUSAWS, ver la sección anterior.

(Se puede definir la url dentro del archivo config, aunque en nuestro caso no está definido de esa manera.)

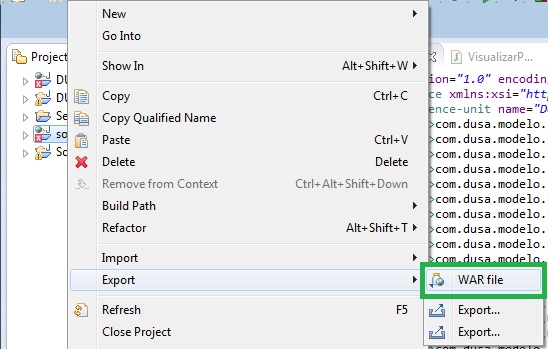
Ahora que están generados los archivos de comunicación se puede correr el proyecto web de la misma forma que el proyecto sorl, ver punto 3.2.1. Solr

* 1. Generar archivos ejecutables

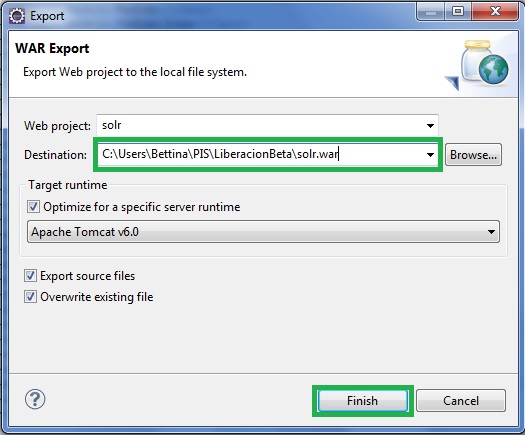
De los proyectos *solr y DUSAWeb*  hay que generar archivos war, que son comprimidos que contienen el proyecto web y motor de búsqueda.

* + 1. solr.war

Para generar el archivo .war desde Eclipse, hacemos clic derecho sobre el proyecto *solr* y seleccionamos la opción *Export* del submenú, seleccionamos *WAR File*; ver imagen:



Nos aparece una ventana y completamos los datos como indica la imagen y hacemos clic en *Finish.*

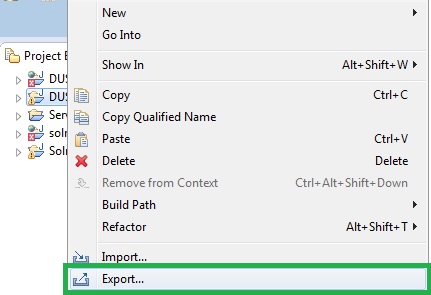


* + 1. DUSAWeb.war

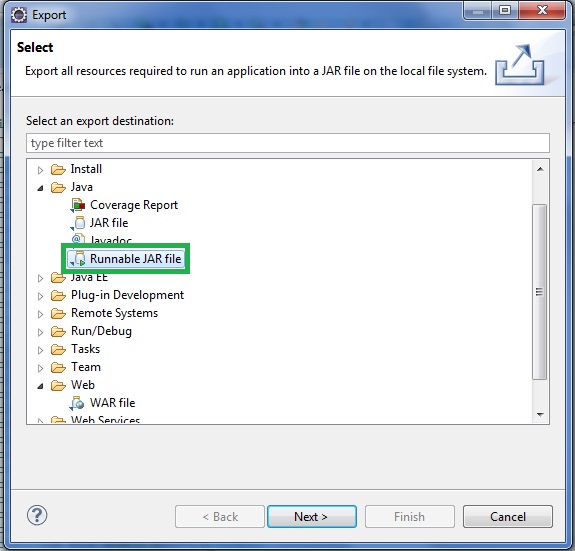
El archivo DUSAWeb.war se genera de forma análoga al solr.war.

* + 1. ws.jar

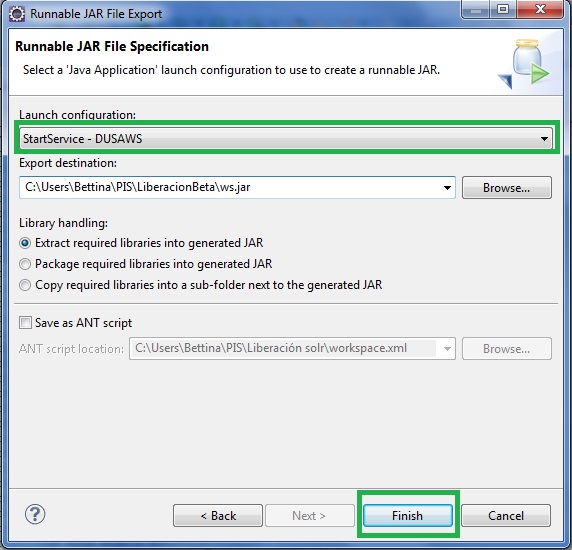
Para generar el ejecutable del proyecto, hacemos clic derecho sobre el proyecto DUSAWS y seleccionamos la opción *Export…* del menú.



Nos aparece una ventana, seleccionamos la opción *Runnable JAR file,* dentro de la carpeta Java y hacemos clic en *Next.*



Nos aparece otra ventana, seleccionamos *StartService – DUSAWS* del menú *Launch configuration:,* le damos una ubicación para que guarde el archivo y hacemos clic en *Finish*.



* 1. Ejecutar proyectos

Para poder ejecutar el proyecto se deben copiar los archivos solr.war y DUSAWeb.war en la carpeta webapps que se encuentra dentro de la carpeta apache-tomcat-6.0.37.

Además en la misma dirección donde se encuentra el archivo ws.jar deben copiar el archivo config.properties que dentro contiene la url donde se está ejecutando el proyecto solr y además hay que crear un archivo llamado nohup.out donde se loguearán las primeras líneas que ejecute el programa.

* + 1. Ejecutar por terminal

Abrir una terminal y ubicarse en la dirección donde se encuentra el archivo ws.jar y ejecutar el siguiente comando: nohup java -jar ws.jar &

Para actualizar tanto el jar como el war, se pueden copiar los archivos java compilados (.class)dentro de las carpetas correspondientes.

La compilacion de cada archivo java se puede ejecutar desde la consola (shell)

y solo actualizar el .class correspondiente a el archivo .java modificado, esto es interesante dado que solo se actualiza el compilado del archivo que se modificó y no es necesario generar todo el jar (o war nuevamente).

En caso que cambien los web services, se agregue o quite una función en el controlador o se modifique un cabezal de función **si es conveniente generarlo mediante eclipse, dado que estos cambian en el cliente (proyecto DUSAWeb)**

1. MOTOR DE BÚSQUEDA
   1. **Objetivos**
   * Facilitar la búsqueda de productos y farmacias al usuario
   * Permitir búsqueda de productos utilizando diferentes criterios con prioridades independientes asociadas
   * Utilizar algoritmos de búsqueda eficientes
   * Tiempos de respuesta considerablemente bajos en búsquedas masivas
   * Reutilización para los distintos sectores del sitio
   * Arquitectónicamente independiente y reemplazable
   1. **Solución implementada**

Para la solución implementada se utilizó **Apache SOLR**.

* + Por qué SOLR?
    - Open source
    - Proyecto de Apache
    - Basado en Lucene (motor de búsqueda más popular)
    - Buen nivel de soporte y documentación
    - Arquitectura robusta
    - Realizado en Java
    - Configuración externa a la aplicación
    - Probado y utilizado en aplicaciones de gran porte (<http://wiki.apache.org/solr/PublicServers>)
    - Ofrece:
      * Búsqueda de texto completo
      * Búsqueda difusa
      * Autocompletar
      * Búsqueda por facetas (categorías)
      * Resaltador de resultados
      * Integración con bases de datos
      * Replicación de índices
      * Consola web de administración
      * Ver más en <http://lucene.apache.org/solr/features.html>)
  + Versión utilizada

Se utilizó versión 4.4.

* + Posibilidades de intercambio

<http://www.manageability.org/blog/stuff/full-text-lucene-jxta-search-engine-java-xml/view>

<http://java-source.net/open-source/search-engines>

* + 1. Descripción de la implementación

Se utiliza una única instancia de Solr, la cual consta de una interfaz web de administración que ejecuta en el servidor tomcat donde reside la aplicación. Esta instancia tiene 2 “cores”, uno llamado “productos” y otro “farmacias”.

Cada core funciona por separado y tiene su propia configuración, ubicada en la carpeta “apache-solr-home/solr/collectionX/conf”, donde Collection1 es la carpeta para productos, y Collection2 para farmacias.

La aplicación web realiza la consulta a la capa lógica por intermedio de Web Services, y ésta se comunica con Solr utilizando su API para java. Las consultas a Solr son realizadas utilizando protocolo HTTP, y el mismo devuelve los resultados en un formato estándar como ser xml o json. El manejo de las respuestas también se hace a través de la API.

* + 1. **Índice y la necesidad de indexación**

Solr aplica sus algoritmos de búsqueda sobre un índice propio generado, no así directamente sobre las tablas de la base de datos. Por esto mismo, en la configuración hay que indicarle a Solr qué tablas y qué campos de las mismas indexar.

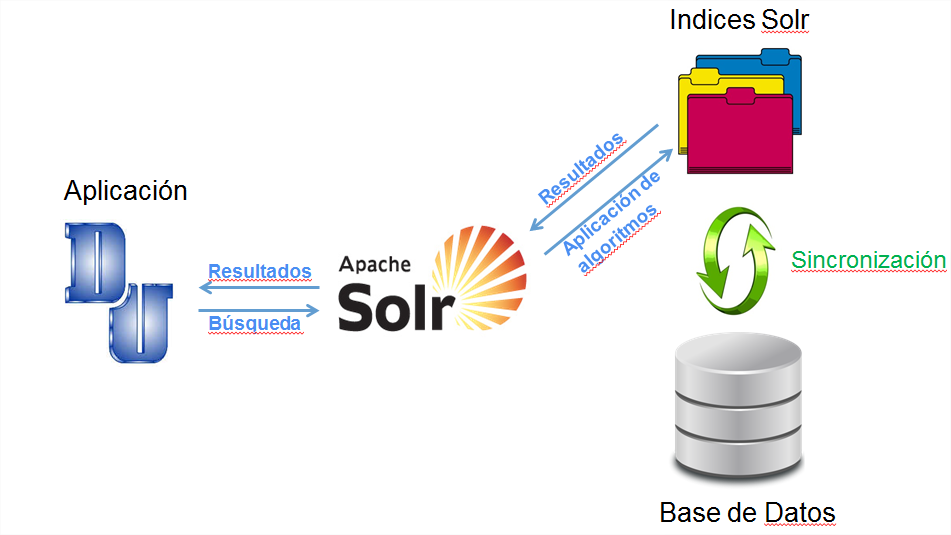
Los índices se ubican en “apache-solr-home\solr\collectionX\data\index”. Una vez realizada la configuración, se ejecuta un comando en el core de Solr y el mismo indexará los datos necesarios para poder buscar sobre los ítems requeridos.

* + 1. **Sincronización entre base de datos e índice**

Dado que la búsqueda se realizan sobre un índice y no directamente sobre los datos de la base de datos, dicho índice actúa como un caché de la base de datos. Por consiguiente, resulta necesario mantener actualizado dicho índice para que los resultados de búsqueda retornen los resultados correctos.

Los métodos de sincronización (indexación) serán detallados más adelante en este documento.

* + 1. **Arquitectura**



* + 1. **DCD del motor dentro de la aplicación**

El componente que implementa el motor de búsqueda dentro de la aplicación está arquitectónicamente separado en el paquete com.dusa.logica.busqueda. Las clases dentro de ese paquete se disponen como se muestra en el documento adjunto (DCD Motor Búsqueda).

Se utilizaron los patrones de diseño Factory, Facade, Template Method y Singelton.

* + 1. **Configuración**
    - Carpeta solr/home

La carpeta solr/home es la carpeta que tiene toda la configuración de la instancia de Solr. En este caso es la carpeta “apache-solr-home/solr” y la misma es referenciada en el web.xml de la aplicación solr. Para cada core se deben configurar (mínimo) los siguientes 3 archivos:

* solrconfig.xml

Ubicado en CollectionX/conf, en este archivo se configura:

- Librerías utilizadas

- Referencia a data-donfig.xml

- Configuración del spellchecker

* schema.xml

Se configura el schema del índice, es decir, se definen los campos y tipos posibles de agregar en el índice. Estos campos son los que se deben utilizar en el data-config.xml, donde se define la consulta.

* + data-config.xml

Archivo donde se configura la conexión a la base de datos, y la consulta que se realiza a la misma para indexar localmente. El string de conexión va especificado aquí.

* + - Web.xml proyecto solr

En este archivo se configura la ubicación de la carpeta solr/home, y el método de autenticación a la aplicacion web de solr. Este es el único archivo que se modificó del war descargado de solr, por lo que a la hora de realizar una actualización solamente se debería modificar este archivo.

Dentro del proyecto solr, se encuentra un archivo de configuración llamado web.xml, que se ubica en: solr\WebContent\WEB-INF\web.xml

Dentro de este archivo, se especifica la dirección absoluta donde está alojada la carpeta apache-solr-home, buscar el siguiente código y reemplazar la dirección en caso de cambiar la carpeta apache-solr-home de ubicación:

<env-entry>

<description>Ubicacion de la carpeta solr/home, donde se encuetra toda la configuración de SOLR</description>

<env-entry-name>solr/home</env-entry-name>

<env-entry-value>/home/grupo10/apache-solr-home/solr</env-entry-value>

<env-entry-type>java.lang.String</env-entry-type>

</env-entry>

* + 1. **Cores de Solr**

Para agregar un nuevo core basta con copiar una de las carpetas collection y configurarla de acuerdo a las necesidades. Actualmente hay dos cores:

* + - Productos
    - Farmacias
    1. **Librerías utilizadas**

Están ubicadas en la carpeta lib en el proyecto WS. Son todas menos el driver de la base de datos. De todas formas, este driver también está en apache-solr-home/lib. Es con este driver que solr se conecta a la base de datos.

* + 1. **Proyectos relacionados**
       1. **Solr (consola de administración)**

Se importó desde Eclipse el war del proyecto solr para poder administrar y configurarlo desde allí. se modificó el web.xml de la aplicación para determinar la ubicación de la carpeta solr/home y configurar la autenticación a la aplicación.

Los usuarios que se pueden autenticar a la aplicación son aquellos que tengan el rol manager-ui. Actualmente el realm utilizado es el nativo del tomcat, por lo tanto, estos usuarios se definen en el archivo tomcat-users.xml.

Como la autenticación implementada no es la misma que la de la aplicación, y resulta necesario que la aplicación tenga acceso a la búsqueda y a la indexación de cada core, las URLs que realizan esto quedaron sin autenticación. Las mismas son:

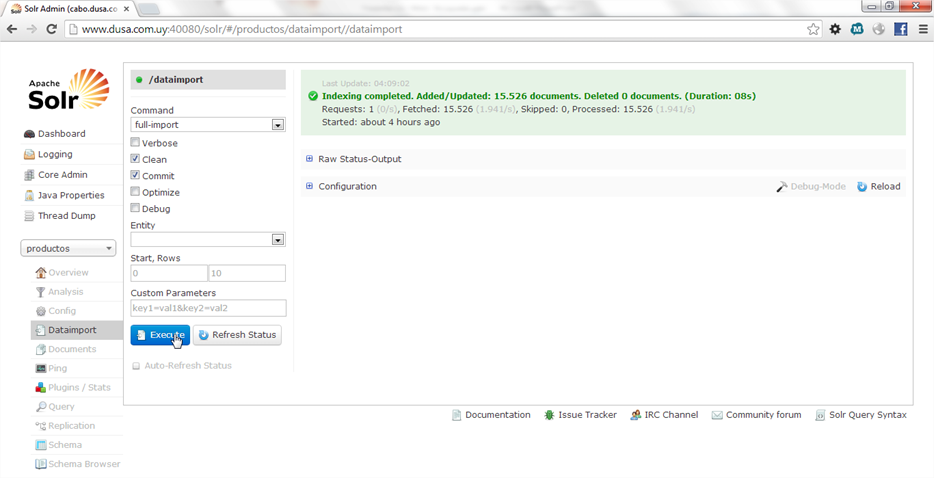
* /solr/productos/\*
* /solr/farmacias/\*

Por lo tanto si se agregara un nuevo core, debería sacarse la restricción a la nueva URL.

La forma ideal de implementar esto sería hacer un Single Sign On (SSO) entre la aplicación y solr, y permitir a ciertos roles acceder a determinadas URLs. No fue posible hacerlo así porque la implementación de la autenticación en la aplicación web no fue realizada de la misma manera que en solr. Lo que se debería hacer es un JDBCRealm (o similar) a nivel de tomcat, y que ambas aplicaciones autentiquen contra dicho realm. De esta forma, Solr y DusaWeb compartirían el grupo de usuarios y roles posibles de autenticarse, y se podría realizar el SSO entre ambos. Con esto, al autenticarse en DusaWeb el usuario quedaría automáticamente autenticado en Solr, y según el rol dependería de a qué sectores de la aplicación Solr puede acceder.

Es la forma de evitar que queden URLs a las que se accede sin autenticación.

Para utilizar la consola de administración se accede a la URL donde se encuentra publicando el tomcat bajo el directorio solr, luego se debe elegir el core a manipular, luego se debe ir a dataimport y hacer click en “execute”. Si se quiere ver el estado se debe hacer click en “refresh status” o seleccionar “auto-refresh status”.



* + - 1. **SolrIndex**

SolrIndex es una aplicación para indexar el motor de búsqueda con simplemente un comando. Utiliza las funciones de IMotorBusqueda en el proyecto WS, por lo que en caso de haber un cambio en la forma de indexar, se realiza en un solo lugar.

Ideal para ser ejecutado como trigger de otro software.

El uso es: SolrIndex [-full | -delta] [-productos | -farmacias]

El primer parámetro indica si se desea hacer una indexación full o delta, mientras que el segundo parámetro indica el core que se desea indexar.

* + 1. **Indexación**
       1. **Indexación full**

Es la indexación de toda la información necesaria de la base de datos (de dos tablas en particular stock y clientes), se puede utilizar haciendo un clean o no del index. Este tipo de indexación es necesaria en cada core, para reflejar la actualización en tablas en búsqueda de productos y farmacias.

* + - 1. **Indexación delta**

Es una forma de indexar que solamente modifica el índice en los archivos que fueron actualizados desde la última indexación.

* + - **Problema encontrado**

Este método de indexación no puede ser implementado con la configuración actual de las tablas en la base de datos. Para que esto fuera posible, debería haber un campo “fecha\_ultima\_modificación” en la base de datos, para poner esta condición en el query de indexación delta. Dado que estos campos no existen en las tablas, esta implementación no se pudo realizar.

* + - **Solución realizada**

Simplemente se hace un full-import con el parámetro clean en false (no borra el índice). Esto supone la actualización de todos los ítems, incluso aquellos que no estaban desactualizados, pero como el tiempo de respuesta no es significante, se toma como aceptable.

* + - **Solución ideal**

Tener en la base de datos un campo que indique la fecha de última modificación. Posterior a esto se deberían configurar los data-config.xml según alguno de estos links:

<http://wiki.apache.org/solr/DataImportHandler#Using_delta-import_command>

<http://wiki.apache.org/solr/DataImportHandlerDeltaQueryViaFullImport>

Una posibilidad sería crear una nueva tabla en la base de datos con los IDs y el campo "ultima\_actualización", actualizando dicha tabla con un trigger en la tabla original. En el data-config.xml se debería hacer el join con esta tabla y obtener el dato desde ahí.

* + - 1. **Indexación mediante método Polling**

Este método está habilitado desde la página, y es únicamente para indexar el buscador de productos. La funcionalidad está disponible únicamente para usuarios administradores de DUSA que tengan el permiso INDEXAR\_BUSCADOR.



1. **DESCRIPCIÓN INTERFAZ Y PROYECTO WEB**

En la interfaz web podemos destacar los siguientes componentes:

* **Lógica de la interfaz:** dentro de la lógica hay tres paquetes con funcionalidades muy distintas.
  + **Comunicación**: en donde se engloba la comunicación por medio de web services con la aplicación.
  + **Servlets**: en donde se encuentran todos los Servlets utilizados en la interfaz, con nombres que identifican claramente su función.
  + **Utils**: aquí se encontraran diferentes clases con distintas funcionalidades, utilizadas en la interfaz web. Por ejemplo, la clase Encriptado, utilizada para la encriptación de las contraseñas, o la clase PermisosUsuario, utilizada, como bien indica su nombre, para el manejo de los permisos de usuario en la interfaz web.
* **WebContent**

En esta carpeta se engloba la parte web de la interfaz, donde podemos encontrar todas las páginas del sitio web en la base de la carpeta, así como los siguientes componentes:

* + Carpeta js: donde se encuentra parte del código javascript utilizado en la interfaz.
  + Carpeta css: usada para almacenar las hojas de estilo utilizadas.
  + Carpeta imagenes: que como indica su nombre, es donde se almacenan las imágenes utilizadas en la interfaz.
  + Carpeta includes

En esta carpeta podremos encontrar segmentos de código, que por repetirse continuamente en casi todas las paginas, se los incluye por medio de la sentencia <@include %>, facilitando así, su mantenimiento y modificación.

En dicha carpeta, actualmente se pueden encontrar 4 archivos:

* Head.jsp : incluida en todas las páginas del sitio, y encargada de controlar que haya un usuario con sesión iniciada (en caso contrario, redirigir al index), aparte de hacer declaraciones correspondientes al código html, como la importación de css, javascript, favicon, etc. (por lo cual debe estas siempre declarada dentro de la etiqueta html head)
* Header.jsp : siendo esta la cabecera de la página (por lo cual, se repite a lo largo de la misma)
* Menú-izquierdo.jsp: tal cual lo indica su nombre, esta porción de código es la encargada de mostrar el menú lateral de la página (menú de usuario).
* Footer.jsp: este código es el encargado de “dibujar” el footer de la página, pero aparte de esto, también es el responsable de capturar los mensajes, y mostrarlos en pantalla (como se detallara más adelante).

**Consideraciones importantes:**

Manejo de permisos

En cuanto al manejo de permisos, para evitar continuos llamados al servidor donde este alojada la lógica de la aplicación, al usuario loguearse, se piden todos los permisos de dicho usuario, y se los guarda en la sesión, utilizando la clase PermisosUsuario (ubicada en src/com/dusa/utils), y bajo el nombre de “permisosUsuario”. Tras esto (que ocurre en el servlet IniciarSesion), en cualquier momento que se necesite consultar los permisos del usuario, tan solo se los extrae de la sesión.

Mensajes de error

Como cabe esperar, en todo momento, al ejecutar funcionalidades se reciben mensajes de éxito, error o warnings. Para mostrar estos al usuario (ya sea una respuesta de un servlet, una redirección o otro) se los debe enviar mediante la sesión, seteando el cuerpo del mensaje, como atributo de la misma, con el nombre “mensaje”, y setear a su vez un atributo adicional, llamado “tipo-mensaje”, que tomara los valores mensaje-ok, mensaje-error o mensaje-warning, dependiendo de la naturaleza del mismo.