|  |
| --- |
| **NIMBUS** |
| Guía de uso |
|  |
| Entorno de desarrollo para Php |
|  |
| **Sergio Pérez Sánchez** |
| **18/07/2011** |
|  |

Tabla de contenido

[1. INTRODUCCIÓN 4](#_Toc309858895)

[2. PATRÓN MVC 5](#_Toc309858896)

[3. ESTRUCTURA DE DIRECTORIOS 10](#_Toc309858897)

[3. EL MODELO 11](#_Toc309858898)

[3.1. Recomendaciones al definir las tablas (entidades de datos) 11](#_Toc309858899)

[3.2. Clases del modelo 12](#_Toc309858900)

[3.2.1. Clase para las propiedades 12](#_Toc309858901)

[3.2.2. Clase para los métodos específicos 14](#_Toc309858902)

[3.2.3. Clases abstractas 15](#_Toc309858903)

[4. EL CONTROLADOR 16](#_Toc309858904)

[4.1. Controlador Frontal: index.php 16](#_Toc309858905)

[4.2. Controlador de Módulo 16](#_Toc309858906)

[5. LA VISTA 18](#_Toc309858907)

[6. FICHEROS DE CONFIGURACIÓN 19](#_Toc309858908)

[6.1. App/config/config.xml 19](#_Toc309858909)

[6.2. App/modules/nombreModulo/config.xml 21](#_Toc309858910)

[6.3. App/modules/nombreModulo/listados.xml 24](#_Toc309858911)

[6.4. App/docs/docsXXX/formats/ 26](#_Toc309858912)

[7. GENERADOR DE LA APLICACIÓN: AppBuilder 29](#_Toc309858913)

[7.1. Generador de entidades. EntityBuilder 30](#_Toc309858914)

[7.2. Generador de controladores. ControllerBuilder 33](#_Toc309858915)

[7.3. Generador de config.xml. ConfigxmlBuilder 34](#_Toc309858916)

[7.4. Generador de listados.xml. ListadosxmlBuilder 35](#_Toc309858917)

[7.5. Generador de templates. TemplateBuilder 36](#_Toc309858918)

[8. HOW TO DO? 37](#_Toc309858919)

[8.1. Cómo establecer la relación “padre-hijo” entre dos controladores 37](#_Toc309858920)

[ANEXO 39](#_Toc309858921)

[Las clases de Nimbus 39](#_Toc309858922)

[Autoloader.class.php 39](#_Toc309858923)

[ControlAcceso.class.php 39](#_Toc309858924)

[Controller.class.php 39](#_Toc309858925)

[DocumentoPdf.class.php 39](#_Toc309858926)

[Entity.class.php 39](#_Toc309858927)

[EntityManager.class.php 39](#_Toc309858928)

[Fecha.class.php 39](#_Toc309858929)

[Form.class.php 39](#_Toc309858930)

[Listado.class.php 39](#_Toc309858931)

[Log.class.php 39](#_Toc309858932)

[Mail.class.php 39](#_Toc309858933)

[Menu.class.php 39](#_Toc309858934)

[Password.class.php 39](#_Toc309858935)

[Permisos.class.php 39](#_Toc309858936)

[Request.class.php 40](#_Toc309858937)

[TableDescriptor.class.php 40](#_Toc309858938)

[XmlBuilder.class.php 40](#_Toc309858939)

[XmlRead.class.php 40](#_Toc309858940)

# INTRODUCCIÓN

El objetivo del framework Nimbus es permitir el desarrollo de aplicaciones software de una forma rápida, sistemática y elegante, obviando al desarrollador las tareas repetitivas de acceso a datos, formularios y listados, para que éste se centre exclusivamente en los procesos de negocio específicos de cada aplicación.

Nimbus sigue el paradigma **MVC** (Modelo Vista Controlador) y **POO** (Programación Orientada a Objetos) con PHP.

Para la VISTA se utiliza:

1. Para generar documentos html el motor de renderización de templates TWIG de SensioLabs (<http://twig.sensiolabs.org/>)
2. Para generar documentos PDF la clase FPDF

Para el MODELO se utiliza el patrón ORM (Mapeado de Objetos Relacionales) donde cada tabla de datos (entidad de datos) se traduce en una clase con sus propiedades (las columnas) y sus métodos genéricos setters y getters; y otra con los métodos específicos.

Para el CONTROLADOR, Nimbus tiene las clases necesarias para analizar las peticiones (request) del usuario, y realizar las tareas típicas de acceso a datos: consultar, editar, crear, eliminar, generar listados, enviar emails, etc.

Las url’s son amigables en base a las reglas definidas en un archivo .htaccess y a la existencia de un único controlador frontal (index.php) que se encarga de analizar las peticiones y disparar el controlador y método solicitado.

Las clases que componen Nimbus pueden estar ubicadas en cualquier carpeta del servidor; bien en una común, o bien dentro de la estructura de directorios de cada aplicación. Si en el mismo servidor se alojan varias aplicaciones que utilizan Nimbus, se aconseja utilizar una carpeta común. En este manual nos referiremos a dicha carpeta como **PathToNimbus**. Existe un anexo donde se explican someramente las clases que constituyen Nimbus**.**

Para envío de emails se utiliza la librería PHPMailer\_v2.0.0.

Paralelamente, se ha desarrollado una utilidad (AppBuilder) que permite construir de forma automática todos los objetos que constituyen el esqueleto básico de una aplicación (entidades de datos, templates, controladores, etc).

# PATRÓN MVC

Nimbus está basado en el clásico patrón de diseño conocido como arquitectura MVC, que consiste en tres niveles:

* El Modelo representa la información sobre la que la aplicación opera.
* La Vista renderiza el modelo en páginas web para la interacción con el usuario.
* El Controlador responde a las acciones del usuario, provocando cambios en el modelo o la vista.

La arquitectura MVC separa la lógica del negocio (modelo) de la presentación (vista), resultando un producto fácilmente mantenible. Por ejemplo, si la aplicación debe correr tanto en navegadores estándar, como en dispositivos móviles, tú solo necesitas una nueva vista; puedes mantener el controlador y el modelo original. El controlador oculta al modelo y a la vista los detalles del protocolo utilizado para la petición (HTTP, modo consola, mail, etc.). El modelo abstrae la lógica de los datos, haciendo independientes a la vista y al controlador de ella. La vista y el controlador no conocen el tipo de base de datos utilizado por la aplicación.



**MVC Layering**

To help you understand MVC’s advantages, let’s see how to convert a basic PHP application to an MVC-architectured application. A list of posts for a weblog application will be a perfect example.

**Flat Programming**

In a flat PHP file, displaying a list of database entries might look like the script presented in Listing 2-1.

*Listing 2-1 - A Flat Script*

<?php

// Connecting, selecting database

$link = mysql\_connect('localhost', 'myuser', 'mypassword');

mysql\_select\_db('blog\_db', $link);

// Performing SQL query

$result = mysql\_query('SELECT date, title FROM post', $link);

?>

<html>

<head>

<title>List of Posts</title>

</head>

<body>

<h1>List of Posts</h1>

<table>

<tr><th>Date</th><th>Title</th></tr>

<?php

// Printing results in HTML

while ($row = mysql\_fetch\_array($result, MYSQL\_ASSOC)){

echo "\t<tr>\n";

printf("\t\t<td> %s </td>\n", $row['date']);

printf("\t\t<td> %s </td>\n", $row['title']);

echo "\t</tr>\n";

}

?>

</table>

</body>

</html>

<?php

// Closing connection

mysql\_close($link);

?>

That’s quick to write, fast to execute, and impossible to maintain. The following are the major problems with this code:

• There is no error-checking (what if the connection to the database fails?).

• HTML and PHP code are mixed, even interwoven together.

• The code is tied to a MySQL database.

**Isolating the Presentation**

The echo and printf calls in Listing 2-1 make the code difficult to read. Modifying the HTML code to enhance the presentation is a hassle with the current syntax. So the code can be split into two parts. First, the pure PHP code with all the business logic goes in a controller script, as shown in Listing 2-2.

*Listing 2-2 - The Controller Part, in* ***index.php***

<?php

// Connecting, selecting database

$link = mysql\_connect('localhost', 'myuser', 'mypassword');

mysql\_select\_db('blog\_db', $link);

// Performing SQL query

$result = mysql\_query('SELECT date, title FROM post', $link);

// Filling up the array for the view

$posts = array();

while ($row = mysql\_fetch\_array($result, MYSQL\_ASSOC)){

$posts[] = $row;

}

// Closing connection

mysql\_close($link);

// Requiring the view

require('view.php');

?>

The HTML code, containing template-like PHP syntax, is stored in a view script, as shown in listing 2-3.

*Listing 2-3 - The View Part, in* ***view.php***

<html>

<head>

<title>List of Posts</title>

</head>

<body>

<h1>List of Posts</h1>

<table>

<tr><th>Date</th><th>Title</th></tr>

<?php foreach ($posts as $post): ?>

<tr>

<td><?php echo $post['date'] ?></td>

<td><?php echo $post['title'] ?></td>

</tr>

<?php endforeach; ?>

</table>

</body>

</html>

A good rule of thumb to determine whether the view is clean enough is that it should contain only a minimum amount of PHP code, in order to be understood by an HTML designer without PHP knowledge. The most common statements in views are echo, if/endif,foreach/endforeach, and that’s about all. Also, there should not be PHP code echoing HTML tags.

All the logic is moved to the controller script, and contains only pure PHP code, with no HTML inside. As a matter of fact, you should imagine that the same controller could be reused for a totally different presentation, perhaps in a PDF file or an XML structure.

**Isolating the Data Manipulation**

Most of the controller script code is dedicated to data manipulation. But what if you need the list of posts for another controller, say one that would output an RSS feed of the weblog posts? What if you want to keep all the database queries in one place, to avoid code duplication? What if you decide to change the data model so that the post table gets renamed weblog\_post? What if you want to switch to PostgreSQL instead of MySQL? In order to make all that possible, you need to remove the data-manipulation code from the controller and put it in another script, called the model, as shown in Listing 2-4.

*Listing 2-4 - The Model Part, in* **model.php**

<?php

function getAllPosts(){

// Connecting, selecting database

$link = mysql\_connect('localhost', 'myuser', 'mypassword');

mysql\_select\_db('blog\_db', $link);

// Performing SQL query

$result = mysql\_query('SELECT date, title FROM post', $link);

// Filling up the array

$posts = array();

while ($row = mysql\_fetch\_array($result, MYSQL\_ASSOC)){

$posts[] = $row;

}

// Closing connection

mysql\_close($link);

return $posts;

}

?>

The revised controller is presented in Listing 2-5.

*Listing 2-5 - The Controller Part, Revised, in* **index.php**

<?php

// Requiring the model

require\_once('model.php');

// Retrieving the list of posts

$posts = getAllPosts();

// Requiring the view

require('view.php');

?>

The controller becomes easier to read. Its sole task is to get the data from the model and pass it to the view. In more complex applications, the controller also deals with the request, the user session, the authentication, and so on. The use of explicit names for the functions of the model even makes code comments unnecessary in the controller.

The model script is dedicated to data access and can be organized accordingly. All parameters that don’t depend on the data layer (like request parameters) must be given by the controller and not accessed directly by the model. The model functions can be easily reused in another controller.

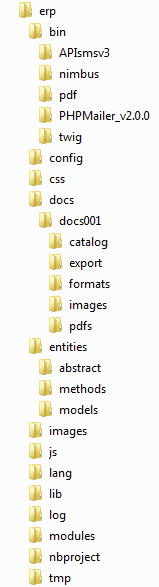
**Resumiendo**:



* Model layer
  + Database abstraction
  + Data Access
* View Layer
  + View
  + Template
  + Layout
* Controller layer
  + Front controller
  + Action

# ESTRUCTURA DE DIRECTORIOS

En este ejemplo ‘erp’ es el nombre de la aplicación que cuelga del document-root.



* bin, para todas las clases genéricas:
  + APIsmsv3, clase para envío de sms’s
  + nimbus, las clases del framework
  + pdf, clase para generar pdfs
  + PHPMailer\_v2.0.0, clase para envío de emails, se puede usar otro
  + twig, motor de templates
* config, archivos xml de configuración general de la aplicación. Actualmente sólo uno: config.xml
* css, archivos de estilos
* docs, archivos generados y configuraciones específicas de cada empresa que utiliza la aplicación
  + catalog, archivos de imágenes de productos
  + export, archivos de exportación de información
  + formats, archivos xml de configuración de formatos de documentos pdf
  + images, archivos de imagen corporativa
  + pdfs, archivos pdf generados
* entities, donde se almacena el modelo de datos
  + abstract, entidades de datos que no se corresponden con tablas de la base de datos
  + methods, para las clases que definen los métodos específicos de las entidades de datos
  + models, para las clases que definen las propiedades y métodos get y set de las entidades de datos
* images, imágenes de decoración de la aplicación
* js, para archivos javascript
* lang, para gestionar el multi-idioma
* lib, clases y/o scripts específicos de la aplicación
* log, para archivos logs
* modules, para los archivos del controlador y de la vista. Habrá una carpeta por cada controlador de la aplicación.
* tmp, archivos temporales para la caché del motor de templates y del autoloader de clases.

# EL MODELO

El modelo es todo aquello que está relacionado con la definición y acceso a la información.

Nimbus dispone de una clase (**Entity**) dotada de todos los métodos necesarios para acceder a la información (select, insert, update, delete, etc) y de una capa de abstracción de datos (**EntityManager**) que hace que el código generado en la aplicación sea independiente del motor de base de datos utilizado.

Los motores de base de datos soportados hasta el momento son: MySql y SQL Server.

Cada **tabla** de la base de datos es representada por **dos clases**: una para las propiedades y métodos set’s y get’s; y otra para los métodos específicos derivados de la lógica de negocio, por ejemplo: inicializar valores, validaciones lógicas, etc.

## Recomendaciones al definir las tablas (entidades de datos)

Dado que se dispone de un módulo (EntityBuilder) capaz de construir las clases para cada tabla, es necesario que la definición de estas cumpla las siguientes reglas:

* Una columna primarykey autoincrementada.
* Definir columnas de auditoría (CreatedBy, CreatedAt, ModifiedBy, ModifiedAt).
* Las columnas CreatedBy y ModifiedBy deben ser definidas como índices.
* El nombre de las columnas que hacen referencia a otras tablas-columnas debe comenzar por ‘ID’ y deben ser definidos como índice.
* Para las columnas que hacen referencia a otras tablas se debe incluir en el apartado ‘comment’, separado por comas: baseDeDatosReferenciada, tablaReferenciada, ColumnaReferenciada.

Ejemplo de la tabla ‘cursos’ que tiene cuatro columnas haciendo referencia a otras tantas tablas:

CREATE TABLE IF NOT EXISTS `cursos` (

`IDCurso` bigint(8) NOT NULL AUTO\_INCREMENT,

`Descripcion` varchar(50) NOT NULL,

**`IDTipoFormacion` bigint(8) NOT NULL DEFAULT '0' COMMENT 'foc\_ppuelr001,tipos\_formacion,IDTipoFormacion',**

**`PublicarWeb` tinyint(1) NOT NULL COMMENT 'foc\_ppuelr001,ValoresSN,IDTipo',**

**`CreatedBy` int(4) NOT NULL DEFAULT '0' COMMENT 'foc\_ppuemp,agentes,IDAgante',**

`CreatedAt` datetime NOT NULL DEFAULT '0000-00-00 00:00:00',

**`ModifiedBy` int(4) NOT NULL DEFAULT '0' COMMENT 'foc\_ppuemp,agentes,IDAgante',**

`ModifiedAt` datetime NOT NULL DEFAULT '0000-00-00 00:00:00',

PRIMARY KEY (`IDCurso`),

KEY `IdTipoFormacion` (`IDTipoFormacion`),

KEY `PublicarWeb` (`PublicarWeb`),

KEY `CreatedBy` (`CreatedBy`),

KEY `ModifiedBy` (`ModifiedBy`)

) ENGINE=MyISAM DEFAULT CHARSET=latin1 AUTO\_INCREMENT=1 ;

NOTA: Obsérvese que una columna puede hacer referencia a una tabla de otra base de datos.

## Clases del modelo

Cada tabla está representada por dos clases.

### Clase para las propiedades

Se almacena en “entities/models”. Para la tabla anterior, el modelo sería el siguiente:

/\*\*

\* entities/models/CursosEntity.class.php

\*/

Class CursosEntity **extend Entity** {

protected $IDCurso;

protected $Descripcion;

protected $IDTipoFormacion='0';

protected $PublicarWeb;

protected $CreatedBy='0';

protected $CreatedAt='0000-00-00 00:00:00';

protected $ModifiedBy='0';

protected $ModifiedAt='0000-00-00 00:00:00';

protected $\_conectionName = 'datos#';

protected $\_tableName = 'cursos';

protected $\_primaryKeyName = 'IDCurso';

public function setIDCurso($IDCurso){ $this->IDCurso = $IDCurso; }

public function getIDCurso(){ return $this->IDCurso; }

public function setDescripcion($Descripcion){ $this->Descripcion = trim($Descripcion); }

public function getDescripcion(){ return $this->Descripcion; }

public function setIDTipoFormacion($IDTipoFormacion){ $this->IDTipoFormacion = $IDTipoFormacion;}

public function **getIDTipoFormacion()**{

if (!($this->IDTipoFormacion instanceof TiposFormacion))

$this->IDTipoFormacion = new TiposFormacion($this->IDTipoFormacion);

return $this->IDTipoFormacion;

}

public function setPublicarWeb($PublicarWeb){ $this->PublicarWeb = $PublicarWeb; }

public function **getPublicarWeb()**{

if (!($this->PublicarWeb instanceof ValoresSN))

$this->PublicarWeb = new ValoresSN($this->PublicarWeb);

return $this->PublicarWeb;

}

}

Algunas aclaraciones **importantes**:

* Esta clase debe extender a la clase Entity que tiene los métodos genéricos de acceso a los datos y que a su vez se apoya en el EntityManager que proporciona la capa de abstracción a los mismos.
* Todas las propiedades deben ser definidas por ‘protected’
* Es conveniente inicializarlas a los valores por defecto según su definición en la base de datos
* Para cada propiedad se definen dos métodos públicos: set y get, para asignarle y obtener su valor respectivamente.
* Se definen tres propiedades especiales (empiezan por guión bajo) para indicarle al modelo el nombre de la conexión a la base de datos, el nombre físico de la tabla, y el nombre de la clave primaria.
* El método get de las propiedades que referencian a otro objeto, no devuelve el valor de la propiedad, sino un objeto de la clase al que referencian.

Como se puede observar, el método getIDTipoFormacion() no devuelve un valor plano, sino que instancia un objeto de la clase TiposFormacion.

Aunque para simplificar no se ha incluido en el ejemplo, se debe definir una propiedad de tipo array almacenando las relaciones que servirán para velar por la integridad referencial:

protected $\_parentEntities = array(

array('SourceColumn' => 'IDCurso', 'ParentEntity' => 'CursosActividades', 'ParentColumn' => 'IDCurso'),

array('SourceColumn' => 'IDTarea', 'ParentEntity' => 'CursosDocumentos', 'ParentColumn' => 'IDTarea'),

array('SourceColumn' => 'IDActividad', 'ParentEntity' => 'CursosTareas', 'ParentColumn' => 'IDActividad'),

);

**Todo lo mencionado, se genera de forma automática por el EntityBuilder.**

### Clase para los métodos específicos

Se almacena en “entities/methods”.

En esta clase se definen los métodos específicos de cada entidad. Por ejemplo, imagina que cada vez que se crea un curso nuevo hay que registrar un determinado evento; el método que lleva a cabo esto debería estar aquí.

/\*\*

\* entities/methods/CursosEntity.class.php

\*/

class Cursos **extends CursosEntity** {

public function \_\_toString() { return $this->getDescripcion(); }

public function validaLogico() {

if ($this->Inicio > $this->Fin)

$this->\_errores[] = "La fecha de finalización del curso debe ser superior a la de inicio.";

}

}

Esta clase debe extender a la que define las propiedades, que a su vez extiende a la que define los métodos globales. Por lo tanto la jerarquía de las clases del modelo es:

Cursos.class.php --> CursosEntity.class.php --> Entity.class.php

Esta jerarquía es totalmente transparente para el CONTROLADOR y la VISTA. Ambas capas solo instanciarán objetos de las clases de nivel inferior (Cursos.class.php).

### Clases abstractas

A menudo, se han de definir valores estáticos y que a criterio del diseñador, no merece la pena que estén implementados en tablas físicas. Para estos casos, se definen clases sin métodos, sólo con los valores estáticos.

Estas clases se almacenan en “entites/abstract” y todas ellas extienden a la clase “Tipos” implementada por Nimbus.

Ejemplo:

/\*\*

\* entities/abstract/DiasSemana.class.php

\*/

class DiasSemana **extends Tipos** {

protected $tipos = array(

array('Id' => '1', 'Value' => 'Lunes'),

array('Id' => '2', 'Value' => 'Martes'),

array('Id' => '3', 'Value' => 'Miercoles'),

array('Id' => '4', 'Value' => 'Jueves'),

array('Id' => '5', 'Value' => 'Viernes'),

array('Id' => '6', 'Value' => 'Sábado'),

array('Id' => '7', 'Value' => 'Domingo'),

);

}

# EL CONTROLADOR

La capa Controlador es la que recibe las peticiones del cliente, identifica las acciones a realizar, las lleva a cabo y devuelve el resultado. Esta capa se materializa en dos archivos: el controlador frontal y el controlador del módulo solicitado.

## 4.1. Controlador Frontal: index.php

Existe sólo uno por aplicación y su lógica interna es totalmente independiente del tipo de aplicación a implementar y además, ya está hecho. El controlador frontal es el archivo “**index.php**” ubicado en el directorio raíz de la aplicación.

Básicamente hace:

* Carga los parámetros de configuración global de la aplicación (config/config.xml)
* Activa el autoloader que se encarga de hacer los includes de las clases que se vayan requiriendo en tiempo de ejecución.
* Activa el motor de templates twig.
* Analiza la petición del usuario, identificando el módulo/controlador y acción/método solicitado
* Ejecuta el método requerido, que devuelve un conjunto de datos y un template.
* Entrega al motor de templates (twig) el binomio datos-template que es el que compone la respuesta hacia el cliente.
* Adicionalmente, y si se ha activado el modo debugger, muestra los valores de todas las variables involucradas en el proceso.

## 4.2. Controlador de Módulo

Se entiende por módulo la agrupación de todas aquellas funcionalidades cuyo objetivo es común. Normalmente una tabla de la base de datos equivale a un módulo; pero puede haber módulos que no se correspondan con ninguna entidad de datos. Es el caso del módulo “Index” que está encargado de presentar el template inicial de la aplicación; o del módulo “\_help” que se encarga de mostrar los templates de ayuda.

Habrá tantos módulos como estime oportuno el desarrollador y se alojan en la carpeta “modules”. Dentro de la carpeta de cada módulo debe haber al menos un archivo ‘.class.php’ con la definición de la clase del módulo, un archivo ‘config.xml’ con los parámetros de configuración y uno o varios archivos ‘.html.twig’ que son el/los template/s que se renderizarán.

Un ejemplo vale más que mil palabras. Veamos un sencillo módulo que sirve para saludar:

Tenemos que implementar un módulo que nos diga “hola” o “adiós” según las siguientes peticiones:

<http://app/Saludo/Hola/Sergio> y <http://app/Saludo/Adios/Sergio>

En esas urls “Saludo” es el nombre del módulo/controlador, “Hola” y “Adios” son las acciones/métodos y “Sergio” es el parámetro. Por lo tanto tenemos que crear un controlador llamado “Saludo” con dos métodos: “Hola” y “Adios”. [[1]](#footnote-2)

<?php

// modules/Saludo/SaludoController.class.php

class SaludoController **extends Controller**{

public function HolaAction(){

$this->values['persona'] = $this->request['3'];

return array('template' => 'Saludo/hola.html.twig', 'values' => $this->values);

}

public function AdiosAction(){

$this->values['persona'] = $this->request[3];

return array('template' => 'Saludo/adios.html.twig', 'values' => $this->values);

}

}

?>

Cualquier método **público** definido en un controlador será accesible vía request (get ó post) y debe devolver (al controlador fontral index.php) un array con dos entradas:

* template : el nombre del template que se renderizará.
* values: es un array donde guardaremos todos los valores generados por la lógica de negocio y que serán renderizados en el template.

En nuestro ejemplo, la lógica de negocio tiene la mínima expresión, consistente en recoger la posición 3 del request. Si la lógica de negocio es más complicada, se pueden definir métodos privados que realicen esta tarea y en los que se apoyarán los métodos públicos que devuelven el resultado.

# LA VISTA

La capa vista es la encargada de mostrar los resultados generados por el controlador. En ella nunca se incluye código php sino html. Nimbus utiliza el motor TWIG.

Siguiendo con el ejemplo anterior, hay que definir los dos templates:

{# modules/Saludo/hola.html.twig #}

{% extends '\_global/layout.html.twig' %}

{% block content %}

<h2>Hola amigo, {{values.persona}}</h2>

{% endblock %}

{# modules/Saludo/adios.html.twig #}

{% extends '\_global/layout.html.twig' %}

{% block content %}

<h2>Adios amigo, {{values.persona}}</h2>

{% endblock %}

Existe abundante documentación de TWIG, pero aquí diré:

* {# #} : es para comentarios
* {% %}: son directivas de twig
* {{ valor }}: equivale a <?php echo $valor;?>
* Los bloques que se definen en un template hijo sustituyen a los de mismo nombre del template al que extienden (el padre).

En nuestro ejemplo, ambos templates extiende al template ‘layout.html.twig’ ubicado en ‘modules/\_global’. Por lo tanto, el bloque ‘content’ del layout será sustituido por este, manteniéndose intacto el resto; y el layout resultante será el que se renderice.

Un posible layout sería:

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="es-ES" lang="es-ES">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<title>.: {{ app.name }} {% block title %} {{ titulo }} {% endblock %}</title>

</head>

<body>

<div id="Cabecera">Aplicacion de ejemplo</div>

<div id="Contenido">

{% block content %}

{% endblock %}

</div>

</body>

</html>

# FICHEROS DE CONFIGURACIÓN

## App/config/config.xml

Fichero de configuraciones globales a toda la aplicación. Se definen en el las conexiones a la base de datos, el servidor de correo, el motor de templates, el título de la aplicación y la ruta de acceso, el modo debuger, etc.

**<config>**

*Para activar/desactivar el modo debugger. Valores posibles: true, false*

**<debug\_mode>**true**</debug\_mode>**

*Para activar/desactivar el modo auditoría. Valores posibles: true, false*

**<audit\_mode>**true**</audit\_mode>**

**<app>**

**<name>**el titulo de la aplicación**</name>**

**<url>**dirección url del dominio de la aplicación “http://www.dominio.com”**</url>**

**<path>**ruta en el formato “/nombreruta”**</path>**

**<path\_framework>**ruta a Nimbus terminada en /**</path\_framework>**

**</app>**

*Se definen los nombres/alias de las conexiones a las bases de datos utilizadas. Se pueden definir N*

**<conections>**

**<empresas>**

**<dbEngine>**mysql**</dbEngine>**

*Tipo de motor de la BD. Valores posibles: mysql, mssql, oracle. Este valor es utilizado por el entityManager (EM) para acceder a las tablas. Actualmente el EM sólo implementa mysql y mssql. Por lo tanto, el sistema podrá interactuar con diferentes motoros de BD simultáneamente.*

**<host>**localhost**</host>**

*El nombre del servidor (Dirección IP ó DNS)*

**<user>**root**</user>**

*Usuario*

**<password>**xxxx**</password>**

*Password*

**<database>**xxxx**</database>**

*Nombre de la base de datos*

**</empresas>**

**<datos001>**

**…..**

**</datos001>**

**</conections>**

*Parámetros de configuración del motor de renderización de templates. Actualmente twig*

**<twig>**

**<motor**>../lib/twig/lib/Twig/Autoloader.php**</motor>**

**<templates\_folder>**modules**</templates\_folder>**

*Carpeta contenedora de los templates*

**<debug\_mode>**true**</debug\_mode>**

**<cache\_folder></cache\_folder>**

*Se se indica un valor, se active la cache e indica la carpeta donde se almacenan las renderizaciones*

**<charset></charset>**

*Juego de caracteres*

**</twig>**

*Servidor de correo electrónico*

**<pdf>**bin/pdf/fpdf.class.php**</pdf>**

**<mailer>**

**<plugin\_dir**>../lib/PHPMailer\_v2.0.0/**</plugin\_dir>**

**<socket>**smtp**</socket>**

**<host>**mail.albatronic.com**</host>**

**<smtp\_auth>**true**</smtp\_auth>**

**<user\_name>**info@albatronic.com**</user\_name>**

**<password>**xxxx**</password>**

**<from>**info@albatronic.com**</from>**

**<from\_name>**ALBATRONIC**</from\_name>**

**<timeout>**30**</timeout>**

**</mailer>**

**</config>**

## App/modules/nombreModulo/config.xml

Este archivo de configuración debe existir para cada modulo. En él se definen los parámetros de configuración del módulo: la conexión de datos, el nombre de la tabla de la BD, las columnas que conforman la entidad, los parámetros del validador, las columnas que se mostrarán en el listado por pantalla y los filtros adicionales para el listado por pantalla, etc.

Explicación detallada de los nodos:

**<Clientes>**

**<login\_required>**YES**</login\_required>**

**<permission\_control>**YES**</permission\_control>**

*Fichero de ayuda (app/modules/\_help/contents/…)*

**<help\_file>**clientes.html.twig**</help\_file>**

*Título que se muestra en el formulario de mantenimiento*

**<title>**Clientes**</title>**

*Nombre de la conexión a la BD y que estará definida en app/config/config.xml. El carácter ‘#’ será sustituido en tiempo de ejecución por la variable $\_SESSION[‘emp’] para poder así acceder a distintas conexiones dependiendo de la empresa seleccionada*

**<conection>**datos#**</conection>**

*Nombre de la entidad de datos a la que se dará mantenimiento. Debe existir en app/entities*

**<entity>**Clientes**</entity>**

*Nombre físico de la table de la BD*

**<table>**clientes**</table>**

*Nombre de la primaryKey de la tabla*

**<primarykey>**IDCliente**</primarykey>**

*Nombre de la columna por la que está entidad está enlazada a su entidad padre.*

**<linkBy>**IDCliente**</linkBy>**

*Registros por página por defecto que se mostrarán en el listado por pantalla*

**<records\_per\_page>**15**</records\_per\_page>**

*Nombre de la columna por la que se realiza por defecto la ordenación de registros en la consulta*

**<order\_by>**RazonSocial**</order\_by>**

*Nombre de la columna por la que se realiza por defecto la búsqueda de registros en la consulta*

**<search\_default>**RazonSocial**</search\_default>**

**Relación de entidades con las que está relacionada la entidad en curso**

**<referenced\_entities>**

**<entity>**nombre de la entidad (no de la tabla física)**</entity**

**……**

**</referenced\_entities>**

**Relación de todas las columnas que constituyen la entidad, excepto las columnas de auditoría (createdBy, createdAt, modifiedBy, modifiedAt)**

**<columns>**

**<column>**

**<title>**Codigo**</title>**

**<field>**IDCliente**</field>**

**<filter>**YES**</filter>**

**<list>**YES**</list>**

**<form>**YES**</form>**

**<link><route/><param/><title/><target/></link>**

**<default></default>**

**Parámetros para construir un filtro adicional por esta columna (opcional).**

**<aditional\_filter>**

*El orden en el que se mostrará en el filtro*

**<order>**1**</order>**

*El texto que se mostrará en el filtro a la izquierda del input, select o check*

**<caption>**Texto**</caption>**

*Si procede, la entidad auxiliar por la que filtrar valores; el método de dicha entidad que devolverá un array de valores(Id,Value) para rellenar el tag select y eventuales parámetros que necesite ese método.*

**<entity>**entidad**</entity>**

**<method>**método de la entidad**</method>**

**<params>**parámetros para el método**</params>**

*Tag html que se mostrará para el filtro. Valores posibles: input, select, range, check. Por defecto se toma input.*

*Si el tipo es ‘select’ y la entidad tiene valor, el sistema generará un tag select para el filtro relleno de los valores obtenidos con el método fetchAll de la entidad indicada (entidad referenciada).*

*Si el tipo es ‘input’ y la entidad tiene valor, el sistema generará un tag input para el filtro y construirá la sentencia SQL mediante joins y utilizando la sentencia LIKE (sin %). Esta forma es la recomendada cuando la columna por la que filtrar puede tener muchos registros referenciados en otra tabla.*

*Si el tipo es ‘input’ y la entidad no tiene valor, el sistema generará un tag input. Este es el caso más simple de filtrado directo sobre una columna propia de la tabla.*

*Si el tipo es ‘range’, no se tiene en cuenta el ‘entity’, se entiende que se filtrará por un rango de valores (normalmente utilizado para fechas). El sistema generará dos tags input para delimitar el rango. El operador de comparación que se indique en ‘operator’ será el que determine el rango inferior.*

*Si el tipo es ‘check’, no se tiene en cuenta el ‘entity’, ya que el sistema filtrará por valores 1 ó 0 según el check esté marcado o no respectivamente.*

**<type>**input**</type>**

*Operador de comparación Para la sentencia sql SELECT. Igual, distinto, mayor o igual, menor o igual, menor, mayor o LIKE. Por defecto se utiliza “=”.*

**<operator>**= | <> | >= | <= | > | < | LIKE**</operator>**

*Evento a realizar con la sintáxis javascript. Ej: onchange='submit();'*

**<event>**evento**</event>**

El valor por defecto que tiene que tomar el filtro. Útil en el caso del tipo ‘check’ para que por defecto valga ‘on’.

**<default>**valor\_por\_defecto**</default>**

**</aditional\_filter>**

**Parámetros para el validador (opcional).**

**<validator>**

**<nullable>**NO**</null>**

*Si el valor es YES, no se realizará validación. Valores posibles: YES, NO*

**<type>**string**</type>**

*El tipo de dato que almacena la columna*

**<length>**15**</length>**

*Longitud máxima de caracteres para la columna*

**<min></min>**

*Valor mínimo para datos numéricos. Opcional. Si se indica, se comparará con el valor máximo*

**<max></max>**

*Valor máximo para datos numéricos. Opcional. Si se indica, se comparará con el valor mínimo*

**<message>**Valor Requerido**</message>**

*Mensaje que mostrará el sistema si no se cumplen las reglas de validación*

**</validator>**

**</column>**

**<column>**

**…..**

**</column>**

**</columns>**

## App/modules/nombreModulo/listados.xml

Este archivo de configuración debe existir para cada modulo. En él se definen los parámetros de configuración de los listados posibles. El generador automático de listados (PathToNimbus/Listado.class.php) se apoya en esta información.

Para cada módulo se pueden definir N tipos de listados, tantos como nodos <listado> se indiquen.

Explicación detallada de los nodos:

**<Agencias>**

**<listado>**

**<title>**Agencias**</title>**

El texto que aparecerá en la cabecera del listado a modo de título

**<order\_by>**IDAgencia ASC**</order\_by>**

Columnas separadas por coma por las que ordenar, al final ASC ó DESC para indicar el orden ascendente o descendente.

**<break\_field></break\_field>**

Campo por el que se romperá el listado para hacer agrupaciones de registros.

Se puede indicar una lista de campos separados por coma. El primer valor será el que se utilice para hacer la ruptura del listado; los siguientes deben ser campos (propiedades) del objeto instanciado en base al primero.

**<break\_page>YES | NO</break\_page>**

Si existe valor para <break\_field> y el valor de <break\_page> es YES, se producirá un cambio de página cada vez que cambie el campo de ruptura.

**<idPerfil></idPerfil>**

IDs de perfiles de usuarios separados por coma que pueden acceder a este listado.

**<orientation>**P**</orientation>**

Orientación de la página: **P** (portrait vertical), **L** (landscape, apaisado).Valor por defecto es **P**.

**<unit>**mm**</unit>**

Unidad de medida: **pt** punto, **mm** milímetro, **cm** centímetro, **in** pulgada

Un punto es igual a 1/72 de pulgada, es decir cerca de 0.35 mm (siendo una pulgada 2.54 cm). Esta es una unidad muy común en tipografía; los tamaños de fuentes son expresados en esa unidad.   
Valor por defecto es mm.

**<format>**A4**</format>**

Tipo de papel a utilizar: **A3**, **A4**, **A5**, **Letter**, **Legal**

**<margins>**10,10,10,10**</margins>**

Márgenes del documento: superior, derecho, inferior, izquierdo

**<body\_font>**Courier, , 8**</body\_font>**

Tipo de letra, estilo y tamaño para el cuerpo del listado. Se aconseja un tipo de letra cuyo tamaño sea de anchura fija, en caso contrario no se calcularán bien las anchuras de las columnas.

Tipos de letra:

Courier (fixed-width)

Helvetica o Arial (sinónimo; sans serif)

Times (serif)

Symbol (symbolic)

ZapfDingbats (symbolic)

También es posible pasar una cadena vacía. En este caso, se mantiene la familia actual.

Estilo:

Estilo de fuente. Los valores posibles son (no sensible a mayúsculas o minúsculas):

cadena vacia: regular

B: bold

I: italic

U: underline

o cualquier combinación. El valor por defecto es regular. Los estilos Bold e italic no aplican para Symbol y ZapfDingbats.

Tamaño:

Tamaño de fuente en puntos. El valor por defecto es el tamaño actual. Si ningún tamaño ha sido especificado desde el comienzo del documento, toma el valor de 12.

**<columns>**

Relación de las columnas que se mostrarán en el listado

**<column>**

**<title>**IDAgencia**</title>**

**<field>**IDAgencia**</field>**

**<length>**4**</length>**

Número de caracteres a impirmir.

**<align>**R**</align>**

Alineación del texto dentro de la columna. Valores posibles: **L** (left), **C** (Center), **R** (Right) , **J** (Justified). Por defecto **R**

**<type>**integer | tinyint |date | string | text**</type>**

Si se indica el tipo ‘text’ se pinta una celda que crece hacia abajo tanto como sea necesario (no se recorta el texto)

**<total>**NO**</total>**

Calcular e imprimir o no la totalización de esta columna.

**<formula></formula>**

Fórmula para calcular la columna.

**<format></format>**

Formato de impresión del valor, especialmente para datos numéricos

**</column>**

**<column>**

**…**

**</column>**

**</columns>**

**</listado>**

**<listado>**

**…..**

**</listado>**

**</Agencias>**

## App/docs/docsXXX/formats/

Aquí se definen los diferentes formatos de impresión de documentos. Cada empresa suele tener un formato propio de documentos (albaranes, facturas, pedidos, etc). XXX representa el código de la empresa.

En esta carpeta habrá tantos archivos xml como tipos de documentos impresos que el sistema generará. Normalmente será: albaranes.xml, pedidos.xml, facturas.xml, etc.

Dentro de cada archivo se definen los formatos distintos para ese tipo de documento. Por ejemplo, para albaranes: formato de albarán valorado, sin valorar, formato ticket tpv, formato orden de trabajo, etc.

Los archivos contienen comandos del motor de generación PDF (actualmente la clase FPDF, pero puede ser cualquiera) que son leídos por la clase PathToNimbus/documentoPdf.class.php y generan el documento pdf en la carpeta app/docs/docsXXX/pdfs.

Contenido de cada archivo:

<albaranes>

<formato>

<title>Albaran Valorado</title>

<idPerfil></idPerfil>

<orientation>P</orientation>

<unit>mm</unit>

<format>A4</format>

<margins>10,10,15,10</margins>

<header> … </header>

<body> … </body>

<footer> … </footer>

</formato>

<formato>

<title>Albaran sin valorar</title>

<idPerfil></idPerfil>

<orientation>L</orientation>

<unit>mm</unit>

<format>A4</format>

<margins>10,10,15,10</margins>

<header> *comandos[[2]](#footnote-3)* </header>

<body> *comandos* </body>

<footer> *comandos* </footer>

</formato>

…..

</albaranes>

Explicación de cada nodo:

**<title>**

Es el título del formato. Es lo que el sistema muestra al usuario en el momento de seleccionar el formato a imprimir.

**<idPerfil>**

Vacio o lista de números separados por coma.

Sirve para indicar qué perfiles de usuarios tienen acceso a este formato. Si no se indica nada, todos tendrán acceso. Los números que se indiquen deben corresponder con el ID del perfil que tiene acceso.

**<orientation>**

Orientación del papel. *P* vertical, *L* apaisado.

**<unit>**

Unidad de medida para los valores de posicionamiento dentro del documento:

* pt puntos
* mm milímetros
* cm centímetros
* in pulgadas

Normalmente milímetros *mm*.

Un punto es igual a 1/72 de pulgada, es decir cerca de 0.35 mm. Ésta es una unidad muy común es tipografía; los tamaños de fuentes son expresados en esa unidad.

**<format>**

Formato del papel a utilizar. *A3*, *A4*, *A5*, *Letter*, *Legal* o un formato personalizado en la forma de un array de dos elementos conteniendo el ancho y el alto expresado en la unidad dada por <unit>.

**<margins>**

Márgenes del documento en el orden: superior, derecho, inferior, izquierdo; expresados en la unidad de medida dada por <unit>. Se deben indicar los cuatro valores separados por coma.

**<header>, <body> y <footer>**

Aquí se indican los comandos del motor pdf que se deben ejecutar para construir la cabecera, cuerpo y pie del documento. Cada comando es un nodo tipo <c> (<c>comando</c>).

Ejemplos:

Pintar una imagen

<c>$this->Image($empresa->getLogo(),15,15,40,30)</c>

Pintar una caja de texto de 1cm de ancho y 4 de alto:

<c>$this->Cell(10,4,’Hello World’)</c>

Establecer el tipo de letra a Arial, negrita y 10pt:

<c>$this->SetFont(‘Arial’,’B’,10)</c>

Se debe indicar **$this** antes del comando, ya que $this hace referencia la clase del motor pdf y el comando es un método público de dicha clase.

# GENERADOR DE LA APLICACIÓN: AppBuilder

EL origen de todo es la definición de la base de datos y a partir de ésta, el appBuilder de Nimbus genera el esqueleto de la aplicación, las clases de las entidades de datos, el controlador de cada módulo y los documentos html que permiten al usuario interactuar con la aplicación. Por módulo se entiende todo lo necesario para la gestión de una entidad de datos. Por lo tanto, si la base de datos tiene 10 tablas, la aplicación tendrá, al menos, 10 módulos.

* Estructura de directorios
* Clases de las entidades de datos
* Para cada modulo:
  + Controlador (lógica de negocio)
  + Archivos de configuración XML para filtros, validaciones y configuración de listados
  + Templates para consultar, editar y listar la información.
  + Template de ayuda

appBuilder

Base

De

Datos

## Generador de entidades. EntityBuilder

Genera dos clases por cada tabla:

* entities/models/**NombreTablaEntity.class.php**

Es la clase que tiene por propiedades las columnas de la tabla y los métodos setters y getters de las propiedades. Para las columnas que hagan referencia a otras, el método get correspondiente devuelve un objeto de la clase del objeto al que referencian.

/\*\*

\* @orm:Entity(agencias)

\*/

class AgenciasEntity **extends Entity** {

/\*\*

\* @orm:GeneratedValue

\* @orm:Id

\* @orm:Column(type="integer")

\* @assert:NotBlank(groups="agencias")

\*/

protected $IDAgencia;

/\*\*

\* @orm:Column(type="string")

\* @assert:NotBlank(groups="agencias")

\*/

protected $Agencia;

/\*\*

\* Nombre de la conexion a la DB

\* @var string

\*/

protected $\_conectionName = 'datos#';

/\*\*

\* Nombre de la tabla física

\* @var string

\*/

protected $\_tableName = 'agencias';

/\*\*

\* Nombre de la primaryKey

\* @var string

\*/

protected $\_primaryKeyName = 'IDAgencia';

protected $\_parentEntities = array(

array('SourceColumn' => 'IDAgencia', 'ParentEntity' => 'TablaPortes', 'ParentColumn' => 'IDAgencia'),

array('SourceColumn' => 'IDAgencia', 'ParentEntity' => 'AlbaranesCab', 'ParentColumn' => 'IDAgencia'),

array('SourceColumn' => 'IDAgencia', 'ParentEntity' => 'FemitidasCab', 'ParentColumn' => 'IDAgencia'),

array('SourceColumn' => 'IDAgencia', 'ParentEntity' => 'PedidosCab', 'ParentColumn' => 'IDAgencia'),

);

/\*\*

\* GETTERS Y SETTERS

\*/

public function setIDAgencia($IDAgencia) { $this->IDAgencia = $IDAgencia; }

public function getIDAgencia() { return $this->IDAgencia; }

public function setAgencia($Agencia) { $this->Agencia = $Agencia; }

public function getAgencia() { return $this->Agencia; }

}

Se incluyen 5 propiedades especiales (no tienen correspondencia con columnas de la tabla) que son utilizadas por el EntityManager:

* + *$\_conectionName* : nombre de la conexión de datos que está definida en el /config/config.xml
  + *$\_tableName*: El nombre físico de la tabla
  + *$\_primaryKeyName*: El nombre de la columna primarykey
  + *$\_parentEntities*: Array con las relaciones que hacen referencia a esta.
  + *$\_childEntities*: Array con las entidades a las que referencia

Esta clase extiende a la clase *Entity* de Nimbus que es la que implementa todos los métodos necesarios para acceder, insertar, actualizar, borrar datos, entre otros.

* Entities/methods/**NombreTabla.class.php**

Es la clase que tendrá los métodos específicos de la lógica de negocio. Por defecto se le genera el método \_\_toString() que devuelve el valor de la primarykey.

class Agencias **extends AgenciasEntity** {

public function \_\_toString() { return $this->getIDAgencia(); }

}

Esta clase extiende a la que contiene las propiedades y los métodos setters y getters.

Por lo tanto la jerarquía de las clases de datos es:

Agencias.class.php --> AgenciasEntity.class.php --> Entity.class.php

**INSTANCIACIÓN AUTOMATICA DE ENTIDADES REFERENCIADAS**

Como se ha comentado antes, el método get de una propiedad que referencie a otra, devuelve un objeto de la clase referenciada. Veamos un ejemplo de la potencia de ello. Supongamos que la tabla ‘agencias’ tiene la columna ‘IDProvincia’ que hace referencia a la columna ‘IDProvincia’ de la tabla ‘provincias’. Entonces el método getIDProvincia de la clase Agencias (*generado automáticamente*) sería:

public function getIDProvincia() {

if (!($this->IDProvincia instanceof Provincias))

$this->IDProvincia = new Provincias ($this->IDProvincia);

return $this->IDProvincia;

}

Supongamos que a partir de un objeto ‘Agencias’ queremos conocer la descripción de la provincia a la que pertenece. Basta con:

// Instanciar un objeto de la clase ‘Agencias’

$agencia = new Agencias(1);

// $provincia es un objeto de la clase ‘Provincias’

$provincia = $agencia->getIDProvincia();

echo $provincia->getDescripcion();

O más rápido aún:

$agencia = new Agencias(1);

echo $agencia->getIDProvincia()->getDescripcion();

Y aún más, si la tabla ‘provincias’ estuviera relacionada con ‘zonas’ a través de la columna ‘IDZona’, podríamos conocer la descripción de la zona desde un objeto ‘Agencias’:

$agencia = new Agencias(1);

echo $agencia->getIDProvincia()->getIDZona()->getDescripcion();

Como se verá más adelante, esta técnica también se puede utilizar dentro de los templates.

**Resumiendo: a partir de un objeto podemos acceder al valor de cualquier propiedad de todas sus entidades referenciadas sea cual sea el nivel de herencia descendente.**

## Generador de controladores. ControllerBuilder

Genera el controlador de cada módulo.

Lo ubica en ‘*modules/Agencias/AgenciasController.class.php*’

En esta clase se definen los métodos necesarios para la lógica de negocio; extiende al controlador principal de Nimbus donde están escritos los métodos para editar, crear, borrar, listar. Por lo tanto, si un módulo no necesita ningún tratamiento especial, el aspecto será:

class AgenciasController **extends Controller** {

protected $entity = "Agencias";

protected $parentEntity = "";

}

En la propiedad *$entity* se debe indicar el nombre de la entidad de datos objeto del control.

La propiedad *$parentEntity* es opcional y debe ser utilizada en el caso de entidades con relación padre-hijo. Es el caso típico de albaranes y líneas de albaranes. En este caso, y suponiendo que las entidades se llamen AlbaranesCab y AlbaranesLineas respectivamente, en el controler de AlbaranesLineas, el valor de $parentEntity debe ser ‘AlbaranesCab’.

Al decirle esto a Nimbus, este hará varias cosas entre ellas el controlador hijo heredará los mismos permisos de acceso que el padre. Ver apartado .

## Generador de config.xml. ConfigxmlBuilder

Genera el archivo de configuración de cada módulo, en base al cual Nimbus construye los filtros, las reglas de validación, el formato del listado por pantalla.

Lo ubica en ‘*modules/Agencias/config.xml*’, el contenido se explica de forma detalla en el apartado

## Generador de listados.xml. ListadosxmlBuilder

Genera el archivo de configuración de cada módulo, en base al cual Nimbus construye los diferentes listados en formato pdf.

Lo ubica en ‘*modules/Agencias/listados.xml*’, el contenido se explica de forma detalla en el apartado

## Generador de templates. TemplateBuilder

# HOW TO DO?

## Cómo establecer la relación “padre-hijo” entre dos controladores

*CP*: Controlador padre

*CH*: Controlador hijo

*APPPATH*: Ruta a la aplicación

1.- En alguna parte del CP hay que llamar al CH, normalmente en form.html.twig donde se incluirá un tag **<IFRAME src=”/APPPATH/CH/list/{{values.ID}}”>**, que llama al método “list” pasándole el valor de la columna (ID) que hay en común entre los dos controladores.

2.- En el CH:

* Incluir en **config.xml** la entrada **<linkBy>IDCliente</linkBy>** para que el *CH* sepa por qué columna se establece la relación.
* Reescribir el método “list” para que tome como filtro el valor pasado en la posición 3 del request:

public function listAction($linkValue=’’) {

If ($linkValue == ‘’) $linkValue = $this->request[3];

// Crear un array con los objetos hijos

$lis = new $this->entity();

$rows = $lis->cargaCondicion($lis->getPrimaryKeyName(), "{$this->values['linkBy']['id']} = '" . $linkValue . "'", 'Orden ASC');

foreach ($rows as $row) {

$lineas[] = new $this->entity($row[$lis->getPrimaryKeyName()]);

}

// El template que se rendizará

$template = $this->entity . '/form.html.twig';

// El valor de enlace. Se pone en el array values para que esté disponible en el template y al hacer el POST lo volverá a enviar a los métodos.

$this->values['linkBy']['value'] = $linkValue;

$this->values['listado']['data'] = $lineas;

unset($lis);

unset($lineas);

return array('template' => $template, 'values' => $this->values);

}

* En el *index.html.twig* para el método NEW sepa a qué padre pertenece:

{% block navegador %}

(<a href='/**APPPATH**/**CH**/new/{{values.linkBy.value}}'>Nuevo</a>)

{% endblock %}

* En el *list.html.twig* para el método EDIT sepa a qué padre pertenece:

<td>

<a href="/ **APPPATH** /**CH**/edit/{{objeto.IDDirec}}/{{values.linkBy.value}}" title="Editar...">{{objeto.Direccion}}</a>

</td>

* En el *edit.html.twig* para el método NEW sepa a qué padre pertenece:

{% block navegador %}

(<a href='/ **APPPATH** /**CH**/new/{{values.linkBy.value}}'>Listado</a>)

{% endblock %}

* En el *new.html.twig* para el método NEW sepa a qué padre pertenece:

{% block navegador %}

(<a href='/ **APPPATH** /**CH**/new/{{values.linkBy.value}}'>Listado</a>)

{% endblock %}

NOTA:

El controller principal se encarga de todo lo demás. Para ello se apoya en el array **values**, al que se le ha incluido el elemento linkBy que a su vez es otro array:

$this->values['linkBy'] = array(

'id' => $this->form->getLinkBy(),

'value' => '',

);

Donde se almacena el nombre de la columna de vinculación y el valor.

# ANEXO

## Las clases de Nimbus

Aquí se describen en orden alfabético las clases que constituyen el core de Nimbus.

### Autoloader.class.php

### ControlAcceso.class.php

### Controller.class.php

### DocumentoPdf.class.php

### Entity.class.php

### EntityManager.class.php

### Fecha.class.php

### Form.class.php

### Listado.class.php

### Log.class.php

### Mail.class.php

### Menu.class.php

### Password.class.php

### Permisos.class.php

### Request.class.php

### TableDescriptor.class.php

### XmlBuilder.class.php

### XmlRead.class.php

1. Todos los controladores se deben llamar LoQueSeaController, deben extender de la clase Controller implementada por Nimbus, y se deben guardar en un archivo llamado LoQueSeaController.class.php ubicado en la carpeta LoQueSea dentro de la carpeta modules. [↑](#footnote-ref-2)
2. Obsérvese que aquí se han de introducir los comandos que utiliza el motor de PDF. Si cambiamos de motor, bastar con cambiar los comandos sin necesidad de tocar el código, consiguiendo así cumplir el patrón MVC separando la capa de la vista y la del controlador. [↑](#footnote-ref-3)