

Empresa GEOCUBA Pinar del Río

**Arquitectura para desarrollo de Aplicaciones de Gestión en
Ambiente Web**

Manual de Usuarios

AÑO 2012

**Agencia de Inv. y Desarrollo Aplicada a
la Geomática**

1. ¿QUÉ ES LA ARQUITECTURA?	1
1.1 Requisitos previos	1
1.2 Filosofía general	1
1.3 Estructura de directorios	2
1.4 Archivos de configuración.	4
1.4.1 Configuración general de la aplicación	4
1.4.1.1 Subsistemas integrados a la aplicación (config.xml)	4
1.4.1.2 Modo de ejecución de la aplicación (app_config.php)	4
1.4.1.3 Configuración de acceso a la base de datos de la seguridad (db_access.php)	6
1.4.2 Configuraciones de los subsistemas	8
1.4.2.1 Configuración de la parte cliente de los subsistemas	8
1.4.2.2 Configuración de la parte servidor de los subsistemas	10
1.4.2.3 Configuración del acceso a datos en la parte servidor de los subsistemas	11
1.4.3 Configuración del acceso a datos	12
2. LA PARTE CLIENTE DE LA ARQUITECTURA	14
2.1 El objeto App	14
2.1.1 Funciones para interactuar con el cliente	15
2.1.2 Funciones para la comunicación con el servidor	16
2.2 Definición de módulos en la parte cliente	18
2.3 Definición de funcionalidades en la parte cliente	21
3. LA PARTE SERVIDOR DE LA ARQUITECTURA	25
3.1 La clase abstracta Module	25
3.2 El acceso a base de datos	27
3.2.1 El manejador de conexiones	27
3.2.2 El manejador de base de datos	28
3.2.3 El manejador de tablas	29
3.2.4 El manejador de selecciones	34
3.2.5 El 'iterador' de selecciones	35
3.3 Clase para validación de datos	35
3.4 El manejo de errores	38
4. LA SEGURIDAD DE LA APLICACIÓN	39

5. COMUNICACIÓN CLIENTE – SERVIDOR	39
6. SUBSISTEMA DE ADMINISTRACIÓN.....	40
6.1 Administración de usuarios.....	40
6.1.1 Permiso de acceso a subsistemas, módulos y funciones	42
6.2 Administración de subsistemas.....	45
7. PROCESO DE DESARROLLO E INTEGRACIÓN.....	47
ANEXO 1	49
ANEXO 2	50

1. ¿Qué es la arquitectura?

La arquitectura es un conjunto de carpetas y clases que le brindan al programador una forma de estructurar y desarrollar de manera rápida y eficiente aplicaciones seguras para la gestión en ambiente web.

1.1 Requisitos previos

La arquitectura requiere de los siguientes requisitos para su funcionamiento:

- Computadora Pentium III 1024Mhz o superior.
- Memoria RAM 256 ó Superior
- Espacio libre en disco duro 64 MB ó Superior
- Sistema Operativo Windows XP, superior ó LINUX.
- Navegador Web (Moxilla Firefox preferiblemente).
- Servidor Web instalado (XAMPP, MapServer, AppServer).

1.2 Filosofía general

La arquitectura está concebida para que sea un gran sistema que contenga o maneje varios subsistemas. Ejemplo: un subsistema para el control de la producción, otro para el control de la contratación, otro para el control de los recursos humanos, en fin tantos como se quiera.

Cada subsistema está compuesto por módulos. Ejemplo: el subsistema del control de la producción tiene un módulo para gestionar los productos que se realizan, uno para controlar la calidad de los productos, uno para controlar la cantidad de productos, etc.

Finalmente cada módulo tiene un conjunto de funcionalidades. Ejemplo: el módulo que gestiona los productos del subsistema del control de la producción permite adicionar, modificar y eliminar los productos.

Bajo esta filosofía de organizar la aplicación en subsistemas que contienen módulos y a su vez estos tienen funcionalidades se desarrolla la arquitectura que se presenta.

Además se incluye un subsistema para la administración de usuarios. Conjuntamente se registran los intentos de autenticación y las llamadas al servidor así como los errores.

1.3 Estructura de directorios

La estructura inicial de directorios de la arquitectura cuenta básicamente con tres carpetas y dos ficheros. La carpeta App almacena las clases controladoras de la aplicación en general, esta se divide en parte cliente y parte servidor. La carpeta Framework almacena en su parte cliente el Framework Ext Js en su versión 4.0.3 y en su parte servidor contiene las clases que intervienen en el acceso a datos, el manejo de módulos y la seguridad de la aplicación. La tercera carpeta, Subsystems, contiene los subsistemas que soporta la aplicación en general. ***El nombre de los subsistemas tiene que coincidir con el nombre de las carpetas por ello se recomienda que el nombre solo contenga letras, sin espacio ni puntos ni tildes.*** Cada subsistema contiene una carpeta (*Config*) para la configuración del mismo y una carpeta para cada módulo que contiene el subsistema. Cada carpeta de los módulos tiene también una parte cliente (*Client*) y una parte servidor (*Server*). También ***el nombre de la carpeta del módulo tiene que coincidir con el nombre del módulo.***

Finalmente cuenta con dos ficheros, *index.php* que es la página para que el usuario se autentique para entrar al sistema y *main.php* que es la página donde se carga el sistema la que se accede

App //Carpeta que contiene los ficheros que contienen las clases controladoras de la aplicación así como su configuración en general.

- |----- **Client** //Carpeta que contiene todo lo necesario para el funcionamiento y la visualización de la parte cliente de la aplicación.
- | |----- **css** //Carpeta que contiene las hojas de estilo que deben ser cargadas en la aplicación
- | |----- **img** //Carpeta que contiene las imágenes e íconos que deben ser cargadas en la aplicación
- | |----- **js** //Carpeta que contiene la clase controladora de la aplicación y el fichero de configuración de la parte visual
- |----- **Server** //Carpeta que contiene los ficheros que intervienen en el funcionamiento de la parte servidor de la aplicación.
- |----- **Config** //Carpeta que contiene los ficheros de configuración de la aplicación en forma general.

Framework //Carpeta que contiene el Framework sobre el cual se sustenta la aplicación de forma general, tanto en la parte cliente como en la parte servidor

- |----- **Client** //Carpeta que contiene los Framework sobre los que se sustenta la parte cliente de la aplicación
 - |----- **ExtJs** //Carpeta que contiene los ficheros del Framework de JavaScript Ext en su versión 4.0.3
- |----- **Server** //Carpeta que contiene el Framework sobre el que se sustenta la parte servidor de la aplicación.
 - |----- **DataAccess** //Carpeta que contiene los ficheros encargados del acceso a los datos
 - |----- **Security** //Carpeta que contiene los ficheros encargados de manejar la seguridad en la aplicación
 - |----- **ServerManagment** //Carpeta que contiene los ficheros encargados de manejar la comunicación cliente - servidor

SubSystems //Carpeta que contiene los sistemas o subsistemas que contiene la aplicación general

- |----- **Admin** //Carpeta que contiene el subsistema para la administración de los roles y los usuarios que pueden acceder a la aplicación
- |----- **(Subsistema 1)** //Carpeta con el contenido del subsistema
 - |----- **Config** //Carpeta que contiene la configuración del subsistema en la parte cliente y en la parte servidor
 - |----- **Client** //Carpeta que contiene la configuración del subsistema en la parte cliente
 - |----- **Server** //Carpeta que contiene la configuración del subsistema en la parte servidor
 - |----- **(Módulos del sistema)**
 - |----- **Client** //Carpeta que contiene los ficheros que definen la parte cliente del módulo
 - |----- **Server** //Carpeta que contiene los ficheros que intervienen en el funcionamiento de la parte servidor del módulo

1.4 Archivos de configuración.

Con el objetivo de hacer flexible la aplicación se necesitan ficheros que definan la configuración de determinados elementos del sistema en su conjunto.

Primero, es necesario definir tres elementos en la aplicación general, los subsistemas que contiene, el modo en tiempo de ejecución y el acceso a datos a la base de datos que interviene en la seguridad de la aplicación.

Segundo, es necesario definir para cada subsistema dos ficheros que intervienen en la visualización de la aplicación y un tercer fichero donde se definen las conexiones a base de datos.

1.4.1 Configuración general de la aplicación

1.4.1.1 Subsistemas integrados a la aplicación (*config.xml*)

El archivo donde se definen los subsistemas de la aplicación se localiza en App\Config\config.xml y su estructura se la siguiente:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<App>
  <subsystem name="Admin"/>
  <subsystem name="RecursosHumanos"/>
</App>
```

Nota: Este archivo de configuración no puede tener comentarios

Donde el valor de la propiedad “name” de la etiqueta “subsystem” indica el nombre del subsistema, de esta forma se puede definir tantos subsistemas como se desee.

1.4.1.2 Modo de ejecución de la aplicación (*app_config.php*)

Pensando en el proceso de desarrollo, se decidió que la aplicación pueda correr en de tres formas:

1. Construcción (DEBUG): Este modo de correr de la aplicación es para el momento en el que el programador está desarrollando la aplicación. En este modo la aplicación le muestra al desarrollador los errores generados en la parte servidor, ya sea en el acceso a datos o en la lógica de la programación. No se realiza el registro llamadas al servidor.
2. Prueba (TESTING): Este modo de correr de la aplicación es para probar la aplicación teniendo en cuenta la seguridad de la misma, o sea, permitiendo a cada usuario ver y hacer lo que se estableció. (Ver más adelante la administración de la aplicación). Se realiza el registro de llamadas al servidor y se muestra al usuario la descripción completa del error.
3. Ejecución (RELEASE): Este es el modo de correr de la aplicación cuando esta está lista. Se realiza el registro de llamadas y errores cuando ocurre un error interno se notifica al usuario un mensaje de error muy general.

El fichero contiene esta configuración se localiza en App\Config\app_config.php y su estructura es la siguiente:

```
<?php
define(DEBUG_MODE, 1); // Modo de construcción
define(TESTING_MODE, 2); // Modo de prueba
define(RELEASE_MODE, 3); // Modo de ejecución.

// Define el modo en que se ejecuta la aplicación.
define(RUN_APP_MODE, DEBUG_MODE);

// Define el sistema que se carga en el modo de construcción.
$_SUBSYSTEM = 'RecursosHumanos';
?>
```

Es necesario notar que la última línea define el subsistema que se cargará por defecto estando la aplicación en modo de construcción y que el programador debe entrar directamente a la página *main.php*.

1.4.1.3 Configuración de acceso a la base de datos de la seguridad (db_access.php)

Teniendo en cuenta que la aplicación final puede estar montado en varios sistemas gestores de base de datos (hasta el momento PostgreSQL, MySQL, Microsoft SQL Server y Microsoft Access) es posible también montar la seguridad en alguno de estos gestores, incluso puede ser que el programador quiera darle distintos nombres a las tablas o puede que el sistema gestor de base de datos no soporte *schemas* como el PostgreSQL y por consiguiente el nombre de las tablas cambia.

Nota: Puede cambiar el sistema gestor de base de datos, puede cambiar el nombre de las tablas que intervienen en la seguridad pero no pueden cambiar los nombres de los campos de las tablas ni las relaciones entre estos.

Para paliar este posible conflicto se define en el fichero App\Config\db_access.php la clase SecurityConnections. En esta se definen los parámetros de conexión a base de datos y los nombres de las tablas que intervienen en la seguridad de la aplicación. A continuación se muestra el fichero.

<?php

```
final class SecurityConnections
{
    private $_tables;
    private $connections_config;
    private static $_instance = 0;

    private function __construct() {
        // Define los parámetros de conexión a la base de datos que interviene en la seguridad
        // La llave 'security' del arreglo asociativo no puede ser cambiada.
        $this->connections_config['security'] = array();
        $this->connections_config['security']['type'] = 'PostgreSQL';
        $this->connections_config['security']['host'] = 'localhost';
        $this->connections_config['security']['user'] = 'postgres';
        $this->connections_config['security']['port'] = '5432';
        $this->connections_config['security']['pass'] = 'postgres';
        $this->connections_config['security']['dbase'] = 'security';
    }
}
```

```

// Define los nombres de las tablas que tienen que ver con la seguridad.
$this->_tables = array();
// Tabla donde se almacenan los subsistemas registrados en la aplicación
$this->_tables['subsystems'] = 'app_security.subsystems';
// Módulos que contienen los subsistemas
$this->_tables['modules'] = 'app_security.modules';
// Funciones que tienen los módulos
$this->_tables['functions'] = 'app_security.functions';
// Tabla que almacena el acceso de los usuarios a los subsistemas
$this->_tables['subsystems_access'] = 'app_security.subsystems_access';
// Tabla que almacena los usuarios registrados en la aplicación
$this->_tables['users'] = 'app_security.users';
// Tabla que almacena las funciones a las que tienen acceso los usuarios
$this->_tables['users_access'] = 'app_security.users_access';
}

public static function GetInstance() {
    if(self::$_instance == 0) self::$_instance = new SecurityConnections ();
    return self::$_instance;
}

public function GetConnectionParams($ConnectionName) {
    return $this->connections_config[$ConnectionName];
}

public function GetTableName($Table) {
    return $this->_tables[$Table];
}
}
?>

```

En el acápite 1.4.3 se abordarán los parámetros de conexión que se establecen para cada tipo de sistema gestor de base de datos que soporta la arquitectura y en el Anexo 1 se mostrará el modelo de base de datos de las tablas que intervienen en la seguridad de la aplicación.

1.4.2 Configuraciones de los subsistemas

Cada subsistema tiene su configuración, tanto de la parte cliente como de la parte servidor. En la parte cliente se define mayormente parte de la interface visual de la aplicación. Se tienen en cuenta elementos como el banner, el área de trabajo, el ícono del sistema, el título y la apariencia del Framework Ext. En la parte servidor se definen los módulos del subsistema y las funcionalidades de cada módulo.

1.4.2.1 Configuración de la parte cliente de los subsistemas

Teniendo en cuenta que cada subsistema puede verse de forma diferente, entonces es necesario tener en cuenta este aspecto y por ello entra en juego la configuración de la parte visual de cada subsistema. Para ello existen dos archivos, el primero *config.xml* y el segundo *Interface.conf.js* (el nombre del segundo puede cambiar).

El archivo config.xml define la hoja de estilo propia del subsistema que se usa mayormente para establecer los ícono del sistema, el ícono del sistema, el título que aparecerá en el navegador cuando el usuario final acceda a él, la apariencia del Framework Ext Js y el nombre del archivo de configuración de la interface visual (Interface.conf.js).

El archivo se encuentra en Subsystems\<Subsistema>\Config\Client\config.xml y su estructura se muestra a continuación.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<subsystem name="RecursosHumanos">
  <!--Ubicación de la hoja de estilo (\Config\Client\css\imagenes.css), en caso de ser 0 no se carga el css -->
  <css>css/imagenes</css>
  <!--Ubicación del icono (\Config\Client\icon\Icono.ico), en caso de ser 0 no se carga el icono-->
  <icon>icon/Icono</icon>
  <!-- Archivo de configuración de la interface visual (\Config\Client\ Interface.conf.js) -->
  <interface>Interface.conf</interface>
  <!-- Apariencia del Framework Ext Js (Framework/Client/ExtJs/resources/css/ext-all.css)-->
  <ext-theme>ext-all</ext-theme>
```

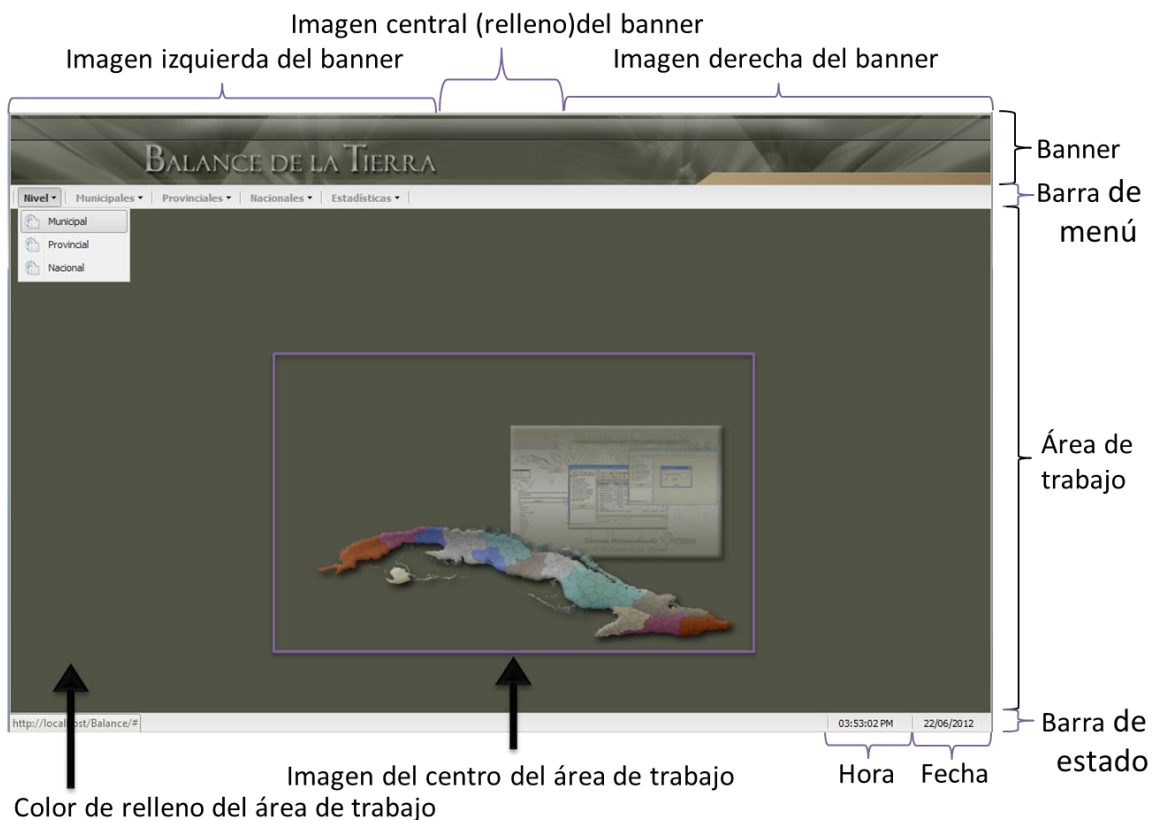
<!--Título que se mostrará en el navegador -->

<title>Control de la produccion Ver 1.0</title>

</subsystem>

Nota: Este archivo de configuración no puede tener comentarios

El otro fichero que interviene en la interface visual del sistema es Interface.conf.js. En este fichero aparece la configuración del banner de la aplicación y de la configuración del área de trabajo. A continuación se muestra una imagen de un sistema realizado, tomando como base la arquitectura que se presenta, donde aparecen las partes de la aplicación en su parte visual.



La estructura que debe tener el fichero se muestra a continuación:

```
InterfaceConf = {  
    // Anchura de la imagen de la izquierda del banner  
    left_image_width : 507,  
    // Imagen de la izquierda del banner  
    left_image : 'Images/lzq.png',  
}
```

```

// Imagen del centro del banner
center_image : 'Images/BG.png',
// Anchura de la imagen de la derecha del banner
righth_image_width : 507,
// Imagen de la derecha del banner
righth_image : 'Images/Der.png',
// Altura del banner
banner_height : 80,
// Color de fondo del área de trabajo
desktop_color : '#515344',
// Imagen del centro del área de trabajo
desktop_image : 'Images/Desktop.png',
// Anchura de la imagen del centro del área de trabajo
desktop_image_width : 560,
// Altura de la imagen del centro del área de trabajo
desktop_image_height : 440
}

```

1.4.2.2 Configuración de la parte servidor de los subsistemas

Es evidente que es necesario definir la configuración interna de cada subsistema, o sea, definir los módulos y las funcionalidades de los módulos. Para ello entra en juego la configuración de la parte servidor de los subsistemas.

El fichero de configuración de la parte servidor de cada subsistema se encuentra ubicado en SubSystems\<Subsistema>\Config\Server\config.xml. Básicamente la estructura del archivo XML consiste en un nodo *subsystem* donde se define el nombre del subsistema. Dentro de dicho nodo están los nodos *module* donde se definen los módulos que contiene el subsistema, estos nodos tienen la descripción del módulo que definen. Cada nodo que define el módulo tiene otros dos nodos: *js* y *functions*. El contenido del nodo *js* no es más que la lista de ficheros js (sin la extensión js y separados por coma sin espacio entre ellos) que requieren ser incluidos a la hora de cargar el módulo. El contenido del nodo *functions* son las funcionalidades asociadas al módulo donde para cada funcionalidad se define su nombre, los ficheros que deben cargarse en la aplicación para su funcionamiento y su descripción.

A continuación se muestra un ejemplo de un fichero de configuración de la parte servidor de un subsistema.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<subsystem name="RecursosHumanos">
<!--El atributo name tiene con coincidir con el nombre del subsistema y el nombre de la carpeta
donde está el subsistema-->
  <module name="Agencias" desc="Módulo para administrar las agencias de una empresa">
    <js>Agencias</js>
    <functions>
      <Adicionar source="Adicionar" desc="Función para crear una nueva agencia"/>
      <Modificar source="Modificar" desc="Función para modificar una agencia"/>
      <Eliminar source="Eliminar" desc="Función para eliminar una agencia"/>
    </functions>
  </module>
  <module name="Empleados" desc="Módulo para administrar los empleados de las agencias">
    <js>Empleados</js>
    <functions>
      <Adicionar source="Adicionar" desc="Función para adicionar un empleado"/>
      <Modificar source="Modificar" desc="Función para modificar los datos de un
empleado"/>
      <Eliminar source="Eliminar" desc="Función para eliminar un empleado"/>
    </functions>
  </module>
</subsystem>
```

Nota: Este archivo de configuración no puede tener comentarios

1.4.2.3 Configuración del acceso a datos en la parte servidor de los subsistemas

Para definir las conexiones a bases de datos que se empleen en un subsistema se utiliza el fichero `Subsystems\<Subsistema>\Config\Server\db_access.php`. En este fichero se debe definir al menos una clase que hereda de *ConnectionsStorage* y su nombre tiene que ser el nombre del subsistema seguido de la palabra '*Connections*' (sin espacios). Por ejemplo, para definir las conexiones que se utilizarán en el subsistema **SistGest** de debe implementar la clase *SistGestConnections* y esta tiene que heredar de *ConnectionsStorage*.

En el constructor de esta clase se definen las conexiones y sus parámetros. Siempre debe llamarse al constructor de la clase padre y después definir los parámetros de conexión sobre la variable *connections_config*. Para esto se trata a *connections_config* como una matriz asociativa donde la primera llave es el nombre de la conexión y las llaves secundarias son los parámetros de conexión.

A continuación se muestra un ejemplo de este fichero.

```
<?php

class InquestConnections extends ConnectionsStorage
{
    public function __construct()
    {
        // Se invoca el constructor de la clase padre
        parent::__construct();

        $this->connections_config['sistgest'] = array();
        $this->connections_config['sistgest']['type'] = 'PostgreSQL';
        $this->connections_config['sistgest']['host'] = 'localhost';
        $this->connections_config['sistgest']['user'] = 'postgres';
        $this->connections_config['sistgest']['port'] = '5432';
        $this->connections_config['sistgest']['pass'] = 'postgres';
        $this->connections_config['sistgest']['dbase'] = 'sistgest';
    }
}

?>
```

1.4.3 Configuración del acceso a datos

Hasta el momento se han visto dos archivos en los cuales se establecen los parámetros de conexión que se utilizan en la parte servidor. Siempre se sigue la misma filosofía, una matriz asociativa donde la primera llave es el nombre de la conexión y las llaves secundarias son los parámetros de conexión. En este momento definiremos cuáles son los parámetros que se necesitan establecer para cada tipo de conexión.

- Conexión con PostgreSQL:
 - type : Tipo de conexión. Tiene que ser 'PostgreSQL'
 - host: Servidor al que se conecta
 - port: Puerto por el cuál se conecta
 - user: Nombre de usuario bajo el cual se conecta
 - pass: Contraseña
 - dbase: Nombre de la base de datos a la que se desea conectar

- Conexión con MySQL:
 - type: Tipo de conexión. Tiene que ser 'MySQL'
 - host: Servidor al que se conecta
 - user: Nombre de usuario bajo el cual se conecta
 - pass: Contraseña
 - dbase: Nombre de la base de datos a la que se desea conectar

- Conexión con Microsoft SQL Server:
 - type: Tipo de conexión. Tiene que ser 'MsSQL'
 - host: Servidor al que se conecta
 - user: Nombre de usuario bajo el cual se conecta
 - pass: Contraseña
 - dbase: Nombre de la base de datos a la que se desea conectar

- Conexión con Microsoft Access
 - type: Tipo de conexión. Tiene que ser 'Access'
 - driver: Driver que se utiliza para establecer la conexión. Puede ser empleado 'DRIVER={Microsoft Access Driver (*.mdb)}; DBQ=%s'
 - location: Dirección física de la base de datos
 - user: Nombre de usuario bajo el cual se conecta
 - pass: Contraseña

2. La parte cliente de la arquitectura

La arquitectura basa su funcionamiento en la parte cliente sobre el Framework de JavaScript Ext en su versión 4.0.3. Con el objetivo de agilizar el desarrollo de las aplicaciones y estandarizar el trabajo se propone una filosofía de trabajo para la parte cliente.

Se establece como regla que un módulo realiza la mayor parte de su trabajo en un panel principal, el cual se muestra en el área de trabajo y no es posible tener dos paneles principales de distintos módulos abiertos a la vez en el área de trabajo. Otra regla que se establece es que la parte cliente de un módulo sólo puede llamar a una función en la parte servidor de él mismo.

Bajo esta filosofía entran a jugar tres elementos, el objeto App, el Módulo en la parte cliente y la Funcionalidad en la parte cliente.

2.1 El objeto App

El objeto App (definido en el fichero App\Client\js\App.js) es la clase controladora de la aplicación en la parte cliente. Dicha clase se encarga de manipular los módulos que se cargan en la aplicación así como sus funcionalidades, de realizar las llamadas al servidor y construir la parte visual de la aplicación. Para ello cuenta internamente con:

- ✓ Un manejador de módulos: Este objeto se encarga de manejar los módulos que se cargan en la aplicación.
- ✓ Un controlador de la interface visual: Este objeto se encarga de construir la aplicación en el navegador. Internamente maneja:
 1. La barra de menú
 2. La barra de estados
 3. El banner de la aplicación
 4. El área de trabajo

Además cuenta con un objeto encargado de manejar los ítems de menú que son puestos por los módulos al iniciarse la aplicación.

- ✓ Un grupo de funciones para que los módulos interactúen con el objeto App. Existen funciones para registrar módulos y funcionalidades, para insertar ítems de menú, para realizar la comunicación con el servidor entre otras.

2.1.1 Funciones para interactuar con el cliente

El objeto App brinda una serie de métodos para que los módulos y funcionalidades puedan interactuar con la aplicación en forma general. Estos son:

- ✓ *RegisterModule (module_name, module)*: Esta función registra un módulo (o controlador de un módulo en la clase cliente) bajo un nombre. Este nombre tiene que ser el mismo de la carpeta donde está definido el módulo.
- ✓ *GetModule (module_name)*: Esta función retorna una instancia del módulo cuyo nombre se pasa por parámetro.
- ✓ *RegisterFunction (function_name, FnInstance)*: Esta función registra una funcionalidad. El primer parámetro es el nombre de la funcionalidad y el segundo una instancia de la misma.
- ✓ *InsertMenuItem (menu, MenuConfig)*: Esta función inserta un ítem de menú en el menú principal de la aplicación. El primer parámetro es el nombre del menú y el segundo es un objeto de configuración de un ítem de menú según Ext Js. Es necesario que tenga definido el atributo 'handler', que es la función que maneja el evento clic sobre el ítem.
- ✓ *InsertSubItemMenu (menu, menu_item_name, SubMenuConfig)*: Esta función inserta un ítem en un submenú dentro de un menú en la barra de menú de la aplicación. El primer parámetro es el nombre del menú en el que se colocará el ítem de menú cuyo nombre es el segundo parámetro y el tercer parámetro es un objeto de configuración de un ítem de menú según Ext Js. Es necesario que tenga definido el atributo 'handler', que es la función que maneja el evento clic sobre el ítem.

- ✓ *ShowMainPanel (filterObj)*: Esta función indica al objeto App que debe mostrar el panel principal que corresponde al módulo actualmente activo.
- ✓ *InfoMessage (title, message)*: Esta función muestra una pequeña notificación al usuario en la parte superior del navegador que se desvanece en un segundo. El primer parámetro es el título del mensaje y el segundo es el mensaje que se mostrará. Se recomienda que sean mensajes cortos.
- ✓ *ShowMsgBox (Ext_MsgBox)*: Esta función muestra un mensaje de los que tiene definido Ext. El parámetro que se le pasa es una instancia de este mensaje.
- ✓ *HideMsgBox* : Esta función oculta el mensaje que fue mostrado con ShowMsgBox.
- ✓ *GetDesktopHeigth*: Esta función retorna el alto del área de trabajo
- ✓ *GetDesktopWidth*: Esta función retorna el ancho del área de trabajo
- ✓ *RegisterValue (Key, Value)*: Esta función registra una variable bajo un nombre determinado. Su mayor uso es para compartir variables entre módulos.
- ✓ *GetRegisteredValue (Key)*: Esta función devuelve el valor registrado bajo la llave pasada como parámetro.
- ✓ *DeleteRegisteredValue (Key)*: Esta función elimina el valor registrado bajo la llave pasada como parámetro.
- ✓ *ClearRegistry*: Esta función elimina todos los valores registrados.

2.1.2 Funciones para la comunicación con el servidor

Como se explicó previamente, un módulo sólo puede invocar funciones implementadas en el servidor dentro del mismo módulo. Para realizar esta comunicación el objeto App brinda una serie de funciones destinada principalmente a tres acciones, construir un objeto de tipo JJsonStore para almacenar datos pedidos al servidor, realizar peticiones al servidor para ejecutar acciones y cargar ficheros hacia el servidor.

Toda llamada al servidor se tiene que realizar a una función de una clase. El nombre de la función a invocar tiene que estar compuesto por el nombre del fichero que contiene la clase, punto, el nombre de la clase contenida en el fichero y que contiene la función, punto, el nombre de la función a invocar.

Ejemplo: 'Productos.ManejadorDeProductos.ListarProductos'. El elemento antes del primer punto (*Productos*) indica en qué fichero está la clase definida en el elemento después del segundo punto (*ManejadorDeProductos*) y el tercero (*ListarProductos*) la función que se desea invocar. Más claro, Invocar la función *ListarProductos* de la clase *ManejadorDeProductos* que está definida en el fichero *Productos.php*.

Se incluye el nombre del fichero dentro de la llamada por dos razones. La primera es que un módulo puede tener muchas funciones en la parte servidor y por un problema de organización y comodidad necesitar más de un fichero. La segunda razón es no obligar a poner un nombre específico, así el programador puede escoger el nombre que desee para el fichero.

También se separa el nombre de la clase del nombre del fichero para no atar al programador a que el mismo nombre de la clase sea el mismo que el del fichero, lo cual es una buena práctica, y como segunda razón es que un fichero puede tener definida más de una clase.

Bajo esta filosofía de llamadas al servidor y para realizar las acciones anteriormente vistas se definen las funciones siguientes:

- ✓ *BuildJsonStore* (*ServerRequest*, *JsonConfigObj*): Esta función construye un objeto de tipo *Ext.data.JsonStore*. El primer parámetro es la función que debe ser invocada y el segundo es el objeto de configuración con el que se crea un *Store* de tipo *JsonStore* en *Ext* pero sin la URL. Si se quiere pasar parámetros a la función que se invoca en el servidor se le pone el campo *params* al parámetro *JsonConfigObj*. El campo *params* debe ser un objeto.
- ✓ *PerformServerRequest* (*ServerRequest*, *Params*, *CallbackFn*, *OnError*, *AppHandleError*): Esta función realiza una petición al servidor utilizando la

implementación de Ajax que contiene Ext. El primer parámetro es la función que se invoca en el servidor. El segundo parámetro es un objeto que contiene los parámetros que se le pasan a la función que se invoca. El tercer parámetro es la función que se invoca una vez que retorne la respuesta del servidor, esta función recibe como parámetro el objeto que se retorna del servidor. El tercer parámetro es opcional y es una función para el tratamiento de errores en caso de un error en la parte del servidor, si no es pasada como parámetro el objeto App realiza su propio tratamiento. El último parámetro que también es opcional es para indicar si la aplicación maneja el error o no. Esta función se utiliza para llamadas asíncronas al servidor.

- ✓ *PerformSyncServerRequest* (*ServerRequest*, *Params*, *AppHandleError*): Esta función se utiliza para realizar llamadas síncronas al servidor. El primer parámetro es la función que se invoca en el servidor. El segundo parámetro es un objeto que contiene los parámetros que se le pasan a la función que se invoca. El tercer parámetro es opcional e indica si la aplicación maneja el error en caso de fallar en el servidor. Esta función devuelve el objeto que fue retornado desde el servidor en caso satisfactorio. En caso contrario, o sea, si ocurrió un error retorna **false**.
- ✓ *UploadFile* (*ServerRequest*, *ExtBasicForm*, *CallbackFn*, *ErrorFn*): Esta función permite cargar ficheros para el servidor. El primer parámetro es la función que se invoca en el servidor. El segundo parámetro es el formulario contenido en un objeto Ext.form.Panel de Ext Js. El tercer parámetro es la función que se invoca una vez que se complete la petición al servidor de forma satisfactoria. Si ocurrió algún error se ejecuta la función pasada en el cuarto parámetro (es opcional).

2.2 Definición de módulos en la parte cliente

La definición de la parte cliente de un módulo no es más que una clase que tiene que implementar obligatoriamente tres funciones:

1. *Init*: Función en la que se inicializa el módulo. Generalmente es utilizada para construir el ítem de menú que activa el módulo y realizar inicializaciones necesarias para el trabajo del módulo.

2. *BuildMainPanel*: Esta función se encarga de construir el panel que va a ser mostrado en el área de trabajo de la aplicación.
3. *Free*: Esta función se encarga de “liberar” la memoria utilizada por el módulo. Ejemplo de esto puede ser la destrucción de las variables de tipo Ext.data.Store u otras variables creadas en durante el proceso de trabajo con el módulo.

Una vez terminada la definición de la clase del módulo es necesario registrar una instancia de la misma en el objeto App mediante la invocación de la función App.RegisterModule, que recibe como parámetros el nombre del módulo y una instancia de la clase anteriormente definida.

Ejemplo de la definición y registro de un módulo:

```
// Definición de la clase que define el módulo para la administración de los usuarios
function AdministrarUsuarios() {
    // Variables para almacenar los usuarios y roles registrados en la base de datos
    this.__data_store = null;
    this.__rols = null;

    // Función para inicializar el módulo
    this.Init = function() {

        // Configuración del ítem de menú que se encargará de activar el módulo y mostrar
        // el panel principal del módulo en el área de trabajo de la aplicación
        var __menu_item_config = {
            text: 'Usuarios',
            id: 'users_menu_id',
            iconCls: 'user',
            handler: Ext.Function.bind(this.ShowMainWindow, this)
        };
        // Insertar el ítem de menú bajo el menú Administrar
        App.InsertMenuItem('Administrar', __menu_item_config);
    }
}
```

```

// Función para construir el panel principal del módulo que será mostrado en el área de
trabajo de la aplicación
this.BuildMainPanel = function(filterObj)
{
    this.__data_store = App.BuildJsonStore('UsersManager.UsersManager.LoadData',
    {...});

    this.__rols = App.BuildJsonStore('UsersManager.UsersManager.LoadRols',{...});

    this.__rols.load();

    var _grid = new Ext.grid.GridPanel({...});

    // Construcción del panel que será mostrado en el área de trabajo.
    var _panel = new Ext.Panel(
    {
        title : 'Gestionar Usuarios',
        border : true,
        frame : true,
        layout : 'fit',
        height : App.GetDesktopHeigth(),
        width : '100%',
        items : [_grid],
        listeners : {afterrender : function()
            {
                this.__data_store.load({params:{start:0, limit:25}});
            }, scope : this
        }
    });

    // Se retorna el panel.
    return _panel;
}

// Función para liberar la memoria utilizada en almacenar los datos traídos de la base de
datos de los usuarios y los roles
this.Free = function()
{

```

```

        this.__data_store.removeAll(true);
        delete this.__data_store;
        this.__data_store = null;

        this.__rols.removeAll(true);
        delete this.__rols;
        this.__rols = null;
    }

    // Función que se invoca al oprimir el botón del ítem de menú
    this.ShowMainWindow = function()
    {
        // Función que indica al objeto App que debe mostrar el panel principal del módulo
        // en el área de trabajo
        App.ShowMainPanel(null);
    }
}

// Registro del módulo
App.RegisterModule('Usuarios', new AdministrarUsuarios ());

```

2.3 Definición de funcionalidades en la parte cliente

Con el objetivo de mostrarle al usuario sólo lo que él tiene acceso a ver en cuanto a operaciones que se pueden realizar con un módulo se introduce el concepto de funcionalidad. Para este caso específico es necesario definir dos elementos. Primero, mediante qué componente visual se realiza la acción que define la funcionalidad. Por ejemplo, si para adicionar un producto necesariamente hay que oprimir un botón que muestra una ventana donde se recogen los datos del nuevo producto. El primer elemento al que se hace referencia es al botón en sí. El segundo elemento son las funciones que realiza el módulo tras oprimir el botón. Según el ejemplo citado anteriormente sería la acción de mostrar un formulario donde se introduzcan los datos del nuevo producto. Concretando lo anteriormente expuesto decimos que para definir una funcionalidad se define:

- ✓ Una clase para construir el componente que desencadena las acciones que requiere dicha funcionalidad. Esta clase tiene que implementar tres funciones:

1. *Render*: Función para dibujar el componente que desencadena las acciones de la funcionalidad.
2. *Enable*: Función para habilitar el componente.
3. *Disable*: Función para deshabilitar el componente.

Estas dos últimas funcionalidades tienen el objetivo de habilitar o deshabilitar los componentes de las funcionalidades según la lógica del negocio de la aplicación en cuanto a la interacción del usuario con ella.

- ✓ La extensión de las funciones del módulo mediante el recurso de JavaScript *prototype*.

A continuación se muestra un ejemplo de la definición de una funcionalidad.

// Definición de la clase que detalla la funcionalidad adicionar del módulo de los usuarios

```
function ModificarEmpleado()
{
    // Variable para almacenar el componente visual que desencadena la funcionalidad del
    módulo
    this._btn = null;

    // Función para dibujar el component visual
    this.Render = function(Panel) {
        this._btn = Ext.create('Ext.Button', {
            text : 'Modificar',
            id : 'mod_us_btn_id',
            iconCls : 'mod',
            handler : Ext.Function.bind(this.Owner. OnModificar,this.Owner)
        });

        var tbar = Ext.getCmp('users_tbar_id');
        tbar.add(this._btn);
    }
}
```

```

// Función para habilitar el componente visual
this.Enable = function() {
    this._btn.enable();
}

// Función para deshabilitar el componente visual
this.Disable = function() {
    this._btn.disable();
}
}

// Se registra la funcionalidad para adicionar el usuario
App.RegisterFunction('ModificarEmpleado', new ModificarEmpleado());

// Extensión de las funciones de la clase “AdministrarUsuarios” mediante el recurso de JavaScript
// prototype.
AdministrarEmpleados.prototype.OnModificar = function()
{
    var _mod_panel = new Ext.form.Panel({ ... });

    var _mod_win = new Ext.Window( {
        title : 'Modificar los datos de un empleado,
        ...
        items : [_mod_panel],
        buttons:[ { text:'Modificar',
                    handler : this. ModificarEmpleado
                },{ text: 'Cerrar',
                    handler: function(){_mod_win.close();}
                }]
    });

    _mod_win.show();
}

AdministrarEmpelados.prototype.ModificarEmpleado = function()
{
    var _form = Ext.getCmp('_mod_employee_panel_id').getForm();
    if(_form.isValid())
    {
        ...
    }
}

```

```

function Callback(response)
{ ...
    this.Disable('ModificarEmpleado');
}

App.PerformServerRequest('Empleados.Empleados.ModificarEmpeado',
    _form.getValues(), Ext.Function.bind(Callback,this) );
}
}

```

Veamos algunos detalles del ejemplo que se mostró. La primera parte define una clase con tres métodos como se especificó, Render, Enable y Disable. Lo primero que necesitamos notar es que en la función Render se crea el botón que desencadena la funcionalidad. Nótese que el campo handler del objeto de configuración con que se crea el botón hace referencia a this.Owner. *Esta es la forma en la que se accede directamente al módulo que contiene la funcionalidad y una vez que se acceda al módulo se accede a todos sus campos y funciones.*

Hay una línea intermedia donde se registra la funcionalidad usando la función 'App.RegisterFunction'. Como se ve el primer parámetro es el nombre de la funcionalidad y la segunda es una instancia de la misma. Cuando sea necesario habilitar esta funcionalidad en el módulo simplemente se invoca this.Enable(Funcionalidad) y para deshabilitarla this.Disable(funcionalidad). Un ejemplo del uso de esto se evidencia en la función Callback dentro de la función ModificarUsuario en donde se invoca this.Disable('ModificarUsuario').

Nota: Téngase en cuenta que la funcionalidad se deshabilita en el módulo

La otra parte del fichero es la extensión de las funciones del objeto AdministrarEmpleados mediante el recurso de JavaScript prototype.

De esta forma se define una funcionalidad en un módulo en la parte cliente.

3. La parte servidor de la arquitectura

En la parte servidor de la arquitectura se encuentra un grupo de clases que acomodan el trabajo al programador siempre siguiendo una filosofía de trabajo. Para orientar a objeto la programación se decidió que desde el cliente el usuario invoca funciones de una clase. Dicha clase hereda de la clase abstracta *Module* y toda función que se invoca desde el cliente tiene que implementar una función de validación además de la función que se invoca. El objetivo de esta regla es darle al programador un espacio donde pueda realizar la validación tanto de los datos que son enviados al servidor como velar por el correcto funcionamiento de la lógica del negocio de la aplicación.

Tanto la función que se invoca del servidor como la función de validación reciben como parámetro una referencia a una matriz asociativa donde la llave es el nombre del parámetro enviado desde el cliente y el valor asociado a dicha llave es el valor de la variable propiamente dicha.

3.1 La clase abstracta *Module*

Como se ha aclarado anteriormente, toda función que se invoca desde el cliente tiene que ser una función de una clase que herede de la clase abstracta *Module*. Esta clase tiene una función general donde se invoca la función de validación de cualquier función que se invoca del servidor. El nombre de la función de validación de cualquier función que se invoca desde el servidor tiene que tener antepuesto la palabra *Validate*. Por ejemplo, si desde el cliente se invoca la función *ModificarEmpleado*, la clase que implementa esta función también tiene que implementar la función *ValidateModificarEmpleado*.

También cuenta con una función (*RegisterError*) para registrar los errores que el programador considere que deben ser retornados y mostrados en la parte cliente de la aplicación. Esta función recibe dos parámetros, el primero es el tipo del error y el segundo es la descripción del error.

Por último es necesario destacar que el constructor de esta clase base recibe como parámetro el nombre de la conexión por defecto con la que se trabajará en las funciones que se invoquen desde el cliente así como las funciones de validación. El nombre de esta

conexión tiene que coincidir con uno de los nombres dados en el fichero de configuración de las conexiones. El objeto que maneja la base de datos y que será visto más adelante se almacena en la variable protegida `$_dbase`. De esta manera el programador sólo se preocupa por trabajar con el manejador de la base de datos y se despreocupa del momento en que realiza la conexión a la misma o la retira.

A continuación se muestra un ejemplo de la definición de una clase para administrar los usuarios:

```
<?PHP
class Empleados extends Module
{
    // Constructor de la clase donde se invoca el constructor de la clase padre pasándole como
    // parámetro el nombre de la conexión por defecto
    public function __construct()
    {
        parent::__construct('recursos_humanos');
    }

    public function ValidateCargarEmpleados(&$params){ ... }

    public function CargarEmpleados(&$params){ ... }

    public function ValidateAdicionarEmpleado(&$params){ ... }

    public function AdicionarEmpleado(&$params){ ... }

    public function ValidateEliminarEmpleado(&$params){ ... }

    public function EliminarEmpleado(&$params){ ... }
    ...
}
?>
```

En el Anexo 2 se muestra el código de un módulo implementado.

3.2 El acceso a base de datos

Teniendo en cuenta que la arquitectura está orientada mayormente a aplicaciones de gestión se hace imprescindible el acceso a base de datos. Por esta causa se presentan un grupo de clases bajo una filosofía para acomodar al programador su trabajo en este sentido.

El acceso a datos está diseñado de forma tal que es posible realizar la migración de un sistema gestor de base de datos a otro sin realizar grandes modificaciones en el código escrito por el programador. Esto es posible cambiando únicamente el tipo de conexión y los parámetros en el fichero de configuración de las conexiones. Es importante aclarar que el programador debe tener en cuenta las particularidades del lenguaje SQL en cada sistema gestor de base de datos pues la sintaxis puede variar ligeramente así como las funciones del sistema que se invocan en una consulta.

En el Anexo 2 se muestra el código de un módulo implementado utilizando algunos objetos que se explican a continuación.

3.2.1 El manejador de conexiones

La primera clase que entra en juego es ***ConnectionsManager***. Esta no es más que una clase con métodos estáticos para establecer una conexión a la base de datos según los parámetros establecidos en el fichero de configuración de las conexiones a base de datos. Es necesario aclarar que si el programador no utiliza otra conexión que la establecida en el constructor de la clase por defecto, no es necesario acceder a dicha clase.

Esta clase tiene dos métodos estáticos principales:

- ✓ *GetDatabase(\$connectionName)*: Con este método se obtiene el acceso a una base de datos previamente definidos los parámetros de conexión en el fichero de configuración. El parámetro *\$connectionName* no es más que el nombre de la conexión que se desea establecer.

- ✓ *GetConnectionOfType(\$connectionType, \$connectionData)*: Esta función es para obtener el acceso a una base de datos pasando directamente el tipo de conexión en su primer parámetro y la matriz asociativa con los parámetros de conexión como segundo parámetro.

3.2.2 El manejador de base de datos

Una vez que se pide el establecimiento de una conexión lo que se devuelve es un objeto de tipo ***DataBase***. Esta clase pretende encapsular el trabajo con una base de datos. A través de él es posible realizar acciones como:

- ✓ Acceder a una tabla de la base de datos
- ✓ Comenzar y abortar o ejecutar una transacción
- ✓ Realizar una consulta de selección

Los métodos que brinda la clase de tipo *DataBase* son:

- ✓ *Free()*: Libera los recursos de la conexión a base de datos.
- ✓ *GetTable(\$tableName)*: Retorna un objeto de tipo *Table* que encapsula el trabajo con una tabla. El parámetro que se le pasa es el nombre de la tabla.
- ✓ *BeginTransaction()*: Inicia una transacción.
- ✓ *Commit()*: Ejecuta la transacción iniciada.
- ✓ *Rollback()*: Aborta la transacción iniciada.
- ✓ *Select(\$statement, \$limit, \$offset)*: Realiza una selección a la base de datos y retorna un objeto de tipo *Selection*. Este objeto encapsula el trabajo con una selección hecha a la base de datos. El primer parámetro es el texto SQL de la consulta, el segundo parámetro es opcional e indica la cantidad a retornar y el tercer parámetro opcional también indica a partir de qué elemento recuperar los elementos encuestados.

- ✓ *GetQueryIterator(\$statement)*: Construye un objeto de tipo *ResultIterator* para iterar por las filas que retorna una consulta.

Los métodos hasta ahora mencionados relacionan otros tres objetos: *Table*, *Selection* y *ResultIterator*.

3.2.3 El manejador de tablas

Cuando se accede a una tabla a través de un objeto de tipo *DataBase*, lo que se devuelve es un objeto de tipo *Table*. Este objeto *Table* pretende encapsular el trabajo con una tabla y brinda funciones básicas para:

- ✓ Realizar actualizaciones
- ✓ Realizar inserciones
- ✓ Realizar eliminaciones
- ✓ Limpiar la tabla
- ✓ Obtener la cantidad de elemento de la tabla
- ✓ Obtener el nombre de las columnas de la tabla
- ✓ Obtener un valor de una columna dada una condición o no
- ✓ Obtener el primero ó último valor de una columna dada una condición o no.
- ✓ Obtener todos los registros de la tabla o sólo un rango.

A continuación se mostrarán los métodos que pueden ser aplicados a un objeto de tipo *Table*.

Nota: Algunas funciones muestran un ejemplo de su uso. Para ello asumamos que se está trabajando sobre una tabla que se llama empleados y tiene como campos el carnet de identidad (ci) el nombre, la edad y el sexo.

- ✓ *Update(\$valuesList, \$condition)*: Realiza la actualización de campos de una tabla. El primer parámetro es una matriz asociativa donde las llaves son los nombres de los campos a actualizar y el valor asociado a dicha llave es el valor con que se va a actualizar el campo. El segundo parámetro que es opcional es el texto SQL de la condición bajo la cual se va a actualizar. Retorna la cantidad de elementos actualizados y -1 en lugar de error.

Ejemplo:

```
$empleado->Update(array('ci'=> ``$_ci``, 'nombre'=> ``$_nombre``,  
'edad'=> $_edad, 'sexo'=> ``$_sexo``), "ci = '$_ci_antiguo'");
```

Generaría una sentencia SQL como:

```
"UPDATE empleados SET ci = '$_ci', nombre = '$_nombre', edad = $_edad,  
sexo = '$_sexo' WHERE ci = '$_ci_antiguo';";
```

- ✓ ***Insert()***: Realiza la inserción de los parámetros que se le pasen a la función. Recibe una lista de parámetros los cuales serán insertados en ese mismo orden. Retorna la cantidad de elementos insertados y -1 en lugar de error.

Ejemplo:

```
$empleado->Insert("``$_ci``, ``$_nombre``, $_edad, ``$_sexo``");
```

Generaría una sentencia SQL como:

```
"INSERT INTO empleados VALUES ('$_ci', '$_nombre', $_edad, '$_sexo');";
```

- ✓ ***InsertValues(\$values)***: Realiza la inserción de elementos en la tabla. El parámetro que recibe es una matriz asociativa, donde las llaves son los nombres de las columnas de la tabla y los valores asociados a la tabla son los valores que serán insertados.

Ejemplo:

```
$empleado->InsertValues(array('ci'=> ``$_ci``, 'nombre'=>  
``$_nombre``, 'edad'=> $_edad, 'sexo'=> ``$_sexo``));
```

Generaría una sentencia SQL como:

```
"INSERT INTO empleados (ci, nombre, edad, sexo)  
VALUES ('$_ci', '$_nombre', $_edad, '$_sexo');";
```

- ✓ ***DeleteWhere(\$condition)***: Elimina registros de una tabla dada una condición. Esta condición es el parámetro que recibe la función y es el texto SQL de la misma.

Ejemplo:

```
$empleado->DeleteWhere("edad > $_edad");
```

Generaría una sentencia SQL como:

```
"DELETE FROM empleados WHERE edad > $_edad;"
```

- ✓ ***Clean()***: Elimina todos los registros de la tabla.

Ejemplo:

```
$empleado->Clean();
```

Generaría una sentencia SQL como:

```
"DELETE FROM empleados;"
```

- ✓ ***GetRowCount()***: Retorna la cantidad de registros que tiene almacenada la tabla.
- ✓ ***GetFields()***: Retorna los nombres de los campos de la tabla.
- ✓ ***Contains(\$Key_Values)***: Retorna la cantidad de registros que cumplen una condición. El parámetro que se le pasa es una matriz asociativa donde las llaves son los nombres de los campos que intervienen en la condición y los valores asociados son los valores que deben cumplir los campos de los registros.

Ejemplo:

```
$empleado->Contains(array('nombre' => "'$_nombre'", 'edad' => $_edad));
```

Generaría una sentencia SQL como:

```
"SELECT count(*) as count FROM empleados WHERE nombre = '$_nombre' AND edad = $_edad;"
```

- ✓ ***GetValueBy(\$fieldName, \$conditions, \$OrderBy, \$OrderType)***: Esta función retorna el valor de un campo dada una condición. El primer parámetro es el nombre del campo que se desea retornar. El segundo parámetro es la condición para la

búsqueda. Esta condición es una matriz asociativa donde la llave es el nombre del campo y el valor asociado a la es el valor que debe tener el campo. El tercer parámetro que es opcional indica el campo (string) o los campos (array) por los cuales ordenar para seleccionar el campo. El cuarto parámetro, opcional también indica el tipo de ordenación, ascendente (por defecto) o descendente.

Ejemplo:

```
$empleado->GetValueBy('ci', array('nombre' => "$_nombre", 'edad' => $_edad));
```

Generaría una sentencia SQL como:

```
"SELECT ci FROM empleados WHERE nombre = "$_nombre" AND edad = $_edad LIMIT 1;"
```

- ✓ *GetValueWhere(\$fieldName, \$whereCondition, \$OrderBy, \$OrderType)*: Esta función retorna el valor de un campo dada una condición. El primer parámetro es el nombre del campo que se desea retornar. El segundo parámetro es el texto SQL de la condición para la búsqueda. El tercer parámetro que es opcional indica el campo (string) o los campos (array) por los cuales ordenar para seleccionar el campo. El cuarto parámetro, opcional también indica el tipo de ordenación, ascendente (por defecto) o descendente.

Ejemplo:

```
$empleado->GetValueWhere('ci', "edad <= $_edad AND sexo = '$_sexo', 'nombre');
```

Generaría una sentencia SQL como:

```
"SELECT ci FROM empleados WHERE edad <= $_edad AND sexo = '$_sexo' ORDER BY nombre LIMIT 1;"
```

- ✓ *GetFirstValue(\$fieldName, \$whereCondition)*: Retorna el primer valor de un campo dada una condición. El primer parámetro es el nombre del campo que se desea obtener. El segundo parámetro (opcional) es el texto SQL de la condición para

buscar el campo que se desea. Hay que notar que el primer valor no indica el primero que fue insertado, sino el menor.

Ejemplo:

```
$empleado->GetFirstValue('nombre', "edad <= $_edad);
```

Generaría una sentencia SQL como:

```
"SELECT nombre FROM empleados WHERE edad <= $_edad  
ORDER BY nombre ASC LIMIT 1;"
```

- ✓ *GetLastValue(\$fieldName, \$whereCondition)*: Retorna el último valor de un campo dada una condición. El primer parámetro es el nombre del campo que se desea obtener. El segundo parámetro (opcional) es el texto SQL de la condición para buscar el campo que se desea. Hay que notar que el último valor no indica el último que fue insertado, sino el mayor.

Ejemplo:

```
$empleado->GetLasrValue('ci', "edad > $_edad AND sexo = '$_sexo');
```

Generaría una sentencia SQL como:

```
"SELECT ci FROM empleados WHERE edad > $_edad AND sexo = '$_sexo'  
ORDER BY ci DESC LIMIT 1;"
```

- ✓ *GetAll()*: Retorna un arreglo de matrices asociativas que contiene todos los registros de la tabla.
- ✓ *GetRange(\$Limit, \$OffSet, \$whereCondition)*: Retorna un rango de registros de la tabla. El primer parámetro es la cantidad de registros. El segundo parámetro (opcional) el desplazamiento, o sea, a partir de qué número. El tercer parámetro (opcional) el texto SQL de la condición que deben cumplir los registros que se retornan.
- ✓ *getTable_name()*: Retorna el nombre de la tabla.

3.2.4 El manejador de selecciones

Como se vio anteriormente, también es posible realizar una consulta a la base de datos. Cuando esto sucede es retornado un objeto de tipo *Selection*. Este objeto pretende encapsular el trabajo con una selección realizada a la base de datos y brinda funciones básicas para:

- ✓ Obtener la cantidad de registros retornados
- ✓ Obtener la lista de campos retornados
- ✓ Obtener el tipo de un campo determinado
- ✓ Obtener en una matriz asociativa todos los elementos resultantes de la consulta
- ✓ Obtener una fila de la consulta
- ✓ Obtener un objeto de tipo *QueryIterator* que permite iterar por las filas retornadas de una consulta.

Los métodos que pueden ser aplicados a este tipo de objeto son:

- ✓ *GetNumRows()*: Retorna la cantidad de elementos retornados en la selección.
- ✓ *GetFieldType(\$field)*: Retorna el tipo de dato de un campo retornado en la consulta. El parámetro que recibe es el nombre del parámetro.
- ✓ *GetFieldsList()*: Retorna la lista de campos que devuelve la selección.
- ✓ *Free()*: Libera los recursos empleados por el sistema para la selección.
- ✓ *GetRow(\$rowIndex)*: Retorna un registro de la selección dado el índice del registro.
- ✓ *GetAll()*: Retorna un arreglo de matrices asociativas con todos los registros obtenidos en la selección.
- ✓ *GetIterator()*: Retorna un objeto de tipo *ResultIterator* que permite iterar por cada registro retornado en la selección.
- ✓ *getSqlStatement()*: Retorna el texto SQL de la consulta.

3.2.5 El 'iterador' de selecciones

En ocasiones es necesario insertar datos para una base de datos que provienen de otra base de datos. Con el objetivo de facilitar al desarrollador procesos de este tipo se implementa un objeto de tipo ResultIterator que permite iterar por los registros retornados de una selección. Cuenta básicamente con tres métodos:

- ✓ *Reset()*: Reinicia el proceso de iteración.
- ✓ *Next()*: Se mueve hacia el próximo elemento y retorna true en caso de ser posible, falso en caso contrario.
- ✓ *GetCurrent()*: Obtiene el elemento actual en la iteración.

A continuación se muestra un ejemplo de cómo utilizar este objeto:

```
...
$_selection = $this->_dbase->Select("SELECT * FROM empleados WHERE edad > 25");
if(is_null($_seleccion)) return false;
$_iterator = $_selection-> GetIterator();
$_iterator->Reset();
while($_iterator->Next()) {
    $_row = $_iterator->GetCurrent();
    ...
}
```

3.3 Clase para validación de datos

Se implementó la clase Validator que tiene solamente métodos estáticos para validar la entrada de los datos y convertir las entradas de tipo texto al tipo que realmente debe ser. El objetivo de esta clase es proveer al programador de una herramienta para validaciones básicas de los datos que son enviados desde el cliente.

En el Anexo 2 se muestra el código de un módulo implementado utilizando las funciones de validación de esta clase.

Los métodos que implementa esta clase son:

- ✓ *IsInt(&\$value, \$min_value, \$max_value)*: Comprueba si un texto es la representación de un número entero y si dicho número está en un rango. El primer parámetro es una referencia al texto. El segundo parámetro (opcional) es el valor mínimo y el tercer parámetro (opcional) es el valor máximo.
- ✓ *ToInt(&\$value, \$min_value, \$max_value)*: Convierte un texto en un número entero. El primer parámetro es una referencia al texto. El segundo parámetro (opcional) es el valor mínimo que puede tomar y el tercer parámetro (opcional) es el valor máximo que puede tomar. En caso satisfactorio retorna true y false en caso contrario. El resultado final es asignado al primer parámetro.
- ✓ *IsFloat(&\$value, \$min_value, \$max_value)*: Comprueba si un texto es la representación de un número con coma y si dicho número está en un rango. El primer parámetro es una referencia al texto. El segundo parámetro (opcional) es el valor mínimo y el tercer parámetro (opcional) es el valor máximo.
- ✓ *ToFloat(&\$value, \$min_value, \$max_value)*: Convierte un texto en un número con coma. El primer parámetro es una referencia al texto. El segundo parámetro (opcional) es el valor mínimo que puede tomar y el tercer parámetro (opcional) es el valor máximo que puede tomar. En caso satisfactorio retorna true y false en caso contrario. El resultado final es asignado al primer parámetro.
- ✓ *IsDate(&\$value, \$min_value, \$max_value)*: Comprueba si un texto es la representación de una fecha y si dicha fecha está en un rango. El primer parámetro es una referencia al texto. El segundo parámetro (opcional) es el valor mínimo y el tercer parámetro (opcional) es el valor máximo.
- ✓ *ToDate(&\$value, \$date_format, \$min_value, \$max_value)*: Convierte un texto en una fecha dado un formato. El primer parámetro es una referencia al texto. El segundo parámetro es el formato final en que se desea la fecha. El tercer parámetro (opcional) es el valor mínimo que puede tomar y el cuarto parámetro

(opcional) es el valor máximo que puede tomar. En caso satisfactorio retorna true y false en caso contrario. El resultado final es asignado al primer parámetro.

- ✓ *CheckStringSize(&\$value, \$min_size, \$max_size)*: Valida la cantidad de caracteres de un texto. El primer parámetro es una referencia al texto. El segundo parámetro es la longitud mínima que debe tener y el tercero es la longitud máxima. En caso de que no interese el tamaño mínimo o máximo, se le pasa -1.
- ✓ *IsJSON(&\$str_value)*: Valida que un texto tiene el formato de intercambio de datos Notación de Objetos de JavaScript (JavaScript Object Notation (JSON)) correcto.
- ✓ *ToJSON(&\$str_value, \$as_array)*: Convierte un texto con formato de intercambio de datos Notación de Objetos de JavaScript (JSON) en su correspondiente objeto. El segundo parámetro (opcional) indica si lo convierte a una matriz asociativa, si es true lo convierte en una matriz, si es false (por defecto) lo convierte en un objeto. Retorna true en caso de ser satisfactorio y false en caso contrario. El resultado final es asignado al primer parámetro.
- ✓ *ToCleanSQL(&\$str_value)*: Trata de limpiar una cadena de inyecciones de SQL. El resultado final es asignado al parámetro.
- ✓ *ExistsParams(\$names_array, &\$params)*: Valida que la lista de nombres que se pasan como primer argumento existan como nombres de parámetros que se pasan en el segundo argumento.
- ✓ *PassByPost(\$names_array, &\$params)*: Valida que la lista de nombres que se pasan como primer argumento existan como nombres de parámetros pasados por POST desde el cliente y que se pasan en el segundo argumento de la función.
- ✓ *PassByGet(\$names_array, &\$params)*: Valida que la lista de nombres que se pasan como primer argumento existan como nombres de parámetros pasados por GET desde el cliente y que se pasan en el segundo argumento de la función.

3.4 El manejo de errores

Un elemento importante en todo sistema es el manejo de errores. Este detalle no escapa de la arquitectura que se presenta. Para ello, en la parte servidor, la clase abstracta `Module`, de la cuál tienen que heredar todos los módulos que se desarrollen en el sistema, implementa el método `RegisterError` que ya se mencionó anteriormente. La arquitectura tiene un mecanismo de forma tal que los errores que son registrados por el usuario se muestran en el cliente.

En la parte cliente de la aplicación, el objeto `App` tiene una forma de manejar los errores que es mostrando un mensaje. En caso de que el programador desee tratar de forma diferente un error, puede realizarlo también.

4. La seguridad de la aplicación

La arquitectura ha tenido en cuenta la seguridad de una aplicación en siete aspectos. Primeramente obliga al programador a implementar una función de validación de los datos. Es responsabilidad del programador la correcta implementación o no de la misma. Segundo, brinda una serie de funciones básicas para validar los parámetros que llegan desde el cliente, que pueden ser usadas o no por los programadores. Tercero, garantiza, junto con el subsistema de seguridad, que un usuario pueda acceder a los módulos a los que él tiene acceso solamente y ejecutar las funciones que se le permitió. Cuarto, todas las peticiones al servidor se realizan a través de una sola página, ocultando así la estructura del funcionamiento de la aplicación en la parte servidor. Quinto, garantiza que no se pueda entrar al sistema mediante el uso de inyecciones de SQL. Sexto, tiene en cuenta la cantidad de intentos de entrar a un sistema desde la página principal, cuando esta cantidad pasa de cinco siempre niega el acceso. De esta forma se garantiza que sea complejo entrar a un subsistema utilizando un robot con diccionarios de usuarios y contraseñas. Séptimo, cuando la aplicación corre en modo de Ejecución los mensajes de errores internos en el servidor, ya sea de base de datos o de otro tipo se ocultan y muestra un solo error, ocultando así al usuario o al posible atacante el tipo de sistema gestor de base de datos que se utiliza.

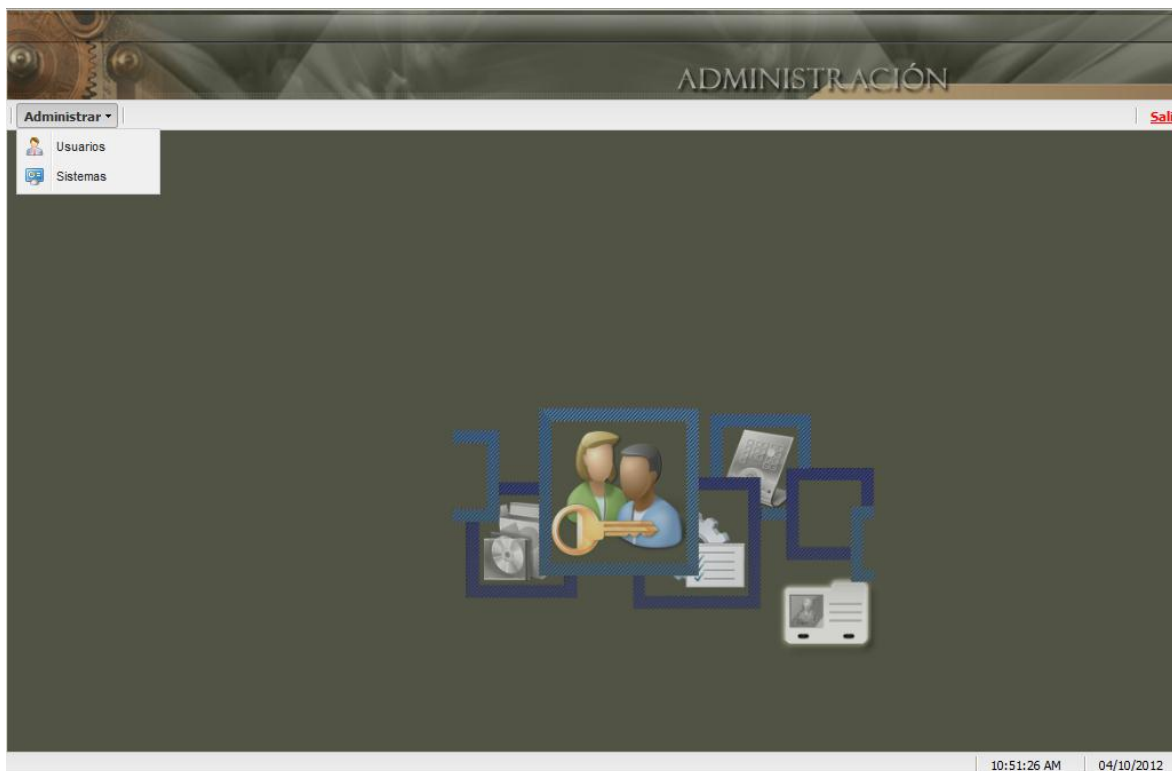
5. Comunicación Cliente – Servidor

La comunicación cliente - servidor se realiza mayormente mediante la tecnología AJAX que incorpora el Framework de JavaScript Ext. Todas las llamadas se realizan a una sola página que procesa las peticiones del cliente y envía las respuestas solicitadas. De esta manera se oculta la estructura y funcionamiento de la aplicación en la parte servidor.

Todas las peticiones se realizan a través del objeto App. Este se encarga de construir la URL a la que se le realiza la petición, enviarla y tratarla una vez que esta es respondida. Si ha ocurrido algún error es manejado por el objeto App pero también se da la posibilidad al programador de manejar el error.

6. Subsistema de Administración

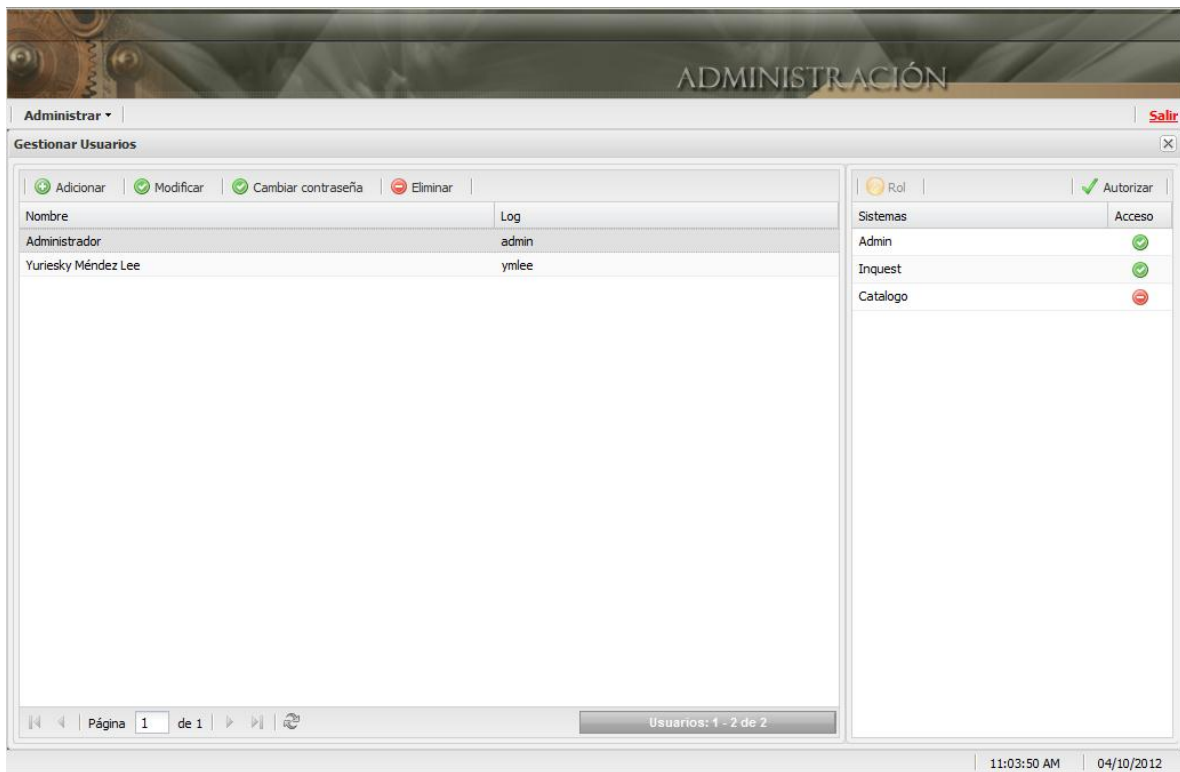
La arquitectura implementa un subsistema para la administración de los usuarios y los subsistemas que tiene la aplicación general.



6.1 Administración de usuarios

La administración de usuarios permite crear usuarios, modificarlos, cambiar su contraseña, eliminarlos, asignarle permisos para acceder a subsistemas y darle permiso para trabajar con módulos y funciones de los subsistemas. De esta forma un usuario puede tener acceso a varios subsistemas e individualizar el acceso a módulos y funcionalidades.

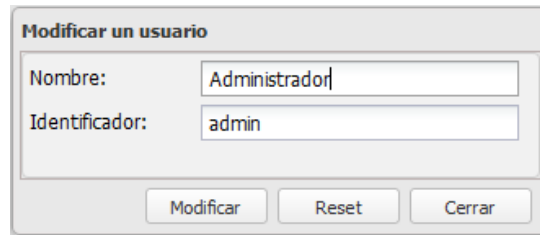
La ventana para la administración de usuarios tiene dos paneles, uno a la derecha y otro a la izquierda. El panel de la izquierda se muestran los usuarios que hay registrados en la aplicación en general. Dicho panel permite crear un nuevo usuario, modificar sus datos, modificar su contraseña o eliminarlo.



Para crear un usuario se oprime el botón Adicionar. Se mostrará una ventana que pedirá el nombre completo del usuario, el identificador y la contraseña con su confirmación.

The screenshot shows a dialog box titled 'Nuevo usuario'. It contains four text input fields labeled 'Nombre:', 'Identificador:', 'Contraseña:', and 'Confirmación:'. At the bottom of the dialog box, there are two buttons: 'Adicionar' and 'Cerrar'.

Para modificar los datos del usuario (nombre e identificador) se oprime el botón modificar. Se mostrará una ventana que pedirá el nuevo nombre y el nuevo identificador.

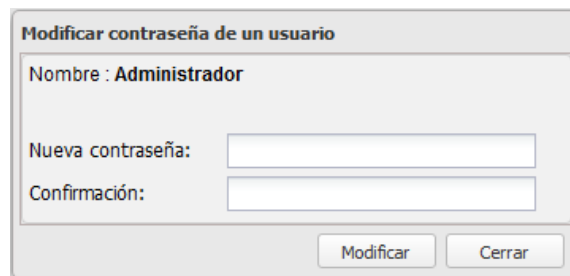


Modificar un usuario

Nombre:

Identificador:

Para cambiar la contraseña a un usuario se oprime el botón 'Cambiar contraseña'. Se mostrará una ventana que pedirá la nueva contraseña y su confirmación.



Modificar contraseña de un usuario

Nombre : Administrador

Nueva contraseña:

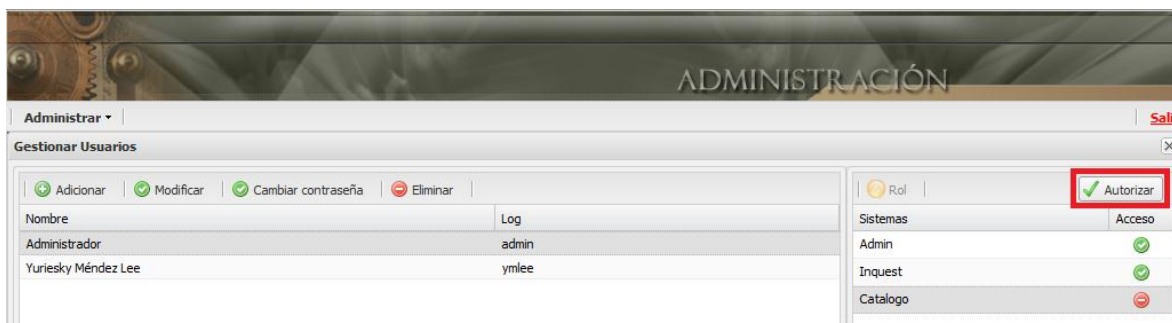
Confirmación:

Para eliminar un usuario se oprime el botón Eliminar. Se mostrará un mensaje de confirmación que en caso de aceptar se eliminará el usuario. Es necesario aclarar que un usuario no se puede eliminar a él mismo.

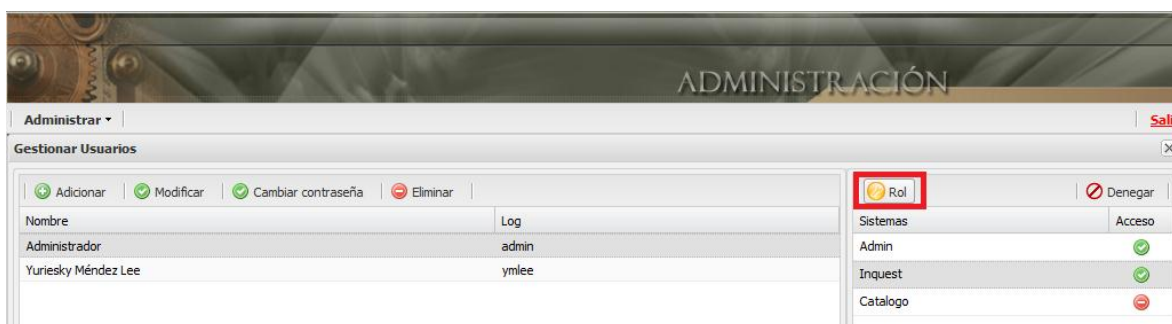
6.1.1 Permiso de acceso a subsistemas, módulos y funciones

Una vez que se seleccione un usuario, en el panel de la derecha se muestran los subsistemas que hay registrados en la aplicación así como el acceso que tiene el usuario a él. Si no tiene acceso al subsistema la columna Acceso mostrará un ícono rojo con un signo de menos. En caso de tener acceso se mostrará un ícono verde con el símbolo (✓).

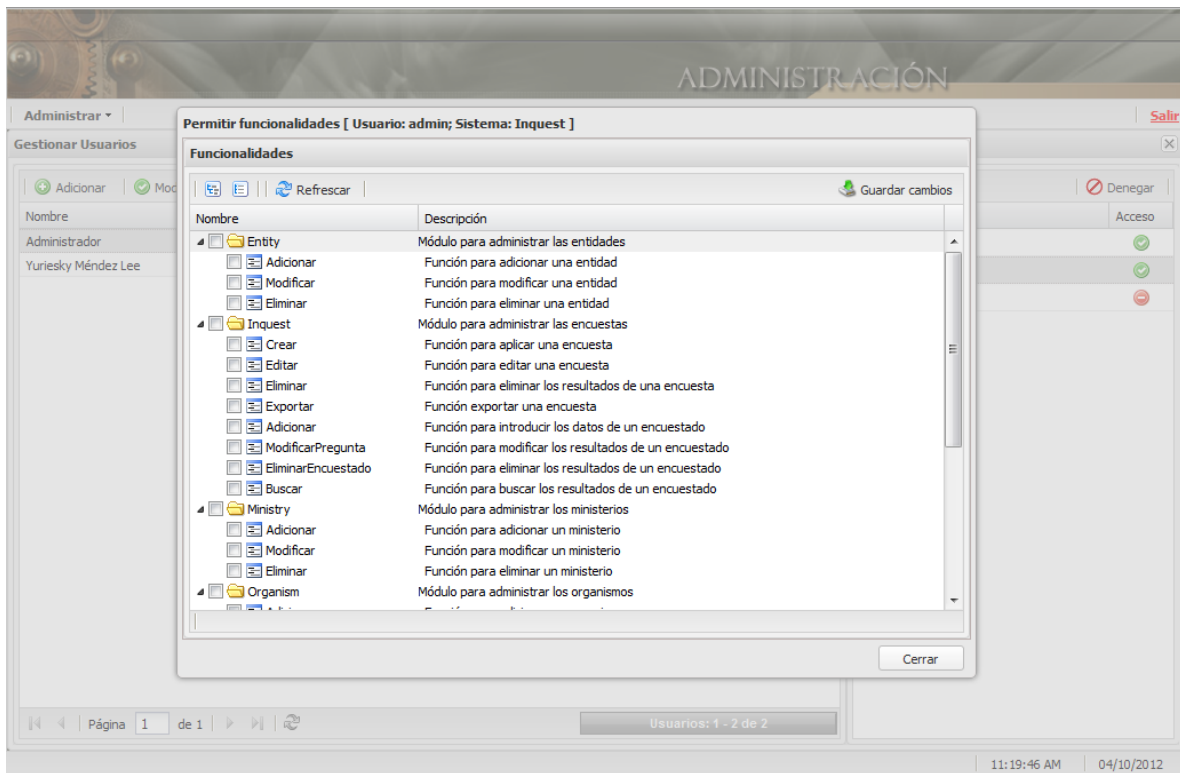
Para permitir o denegar el acceso de un usuario a un subsistema, seleccione el usuario, luego seleccione el subsistema y presione el botón que aparece encima de la columna Acceso en el panel derecho.



Una vez que el usuario tenga acceso a un subsistema se pasa a definir los módulos y las funcionalidades a las que el usuario tendrá acceso en el subsistema. Para ello se selecciona el usuario, el subsistema y se oprime el botón Rol que está encima de la columna sistemas en el panel derecho.



Aparecerá una ventana donde muestra en forma arbórea todos los módulos del subsistema y las funcionalidades de los módulos.



Los módulos son los que se representan con el ícono de una carpeta mientras que las funcionalidades son las que se representan con el ícono cuadrado. Para asignarle o quitarle permiso a un usuario sobre un módulo o funcionalidad solamente tiene que marcar o desmarcar. Para aplicar estos permisos se oprime el botón 'Guardar cambios' de la esquina superior derecha.

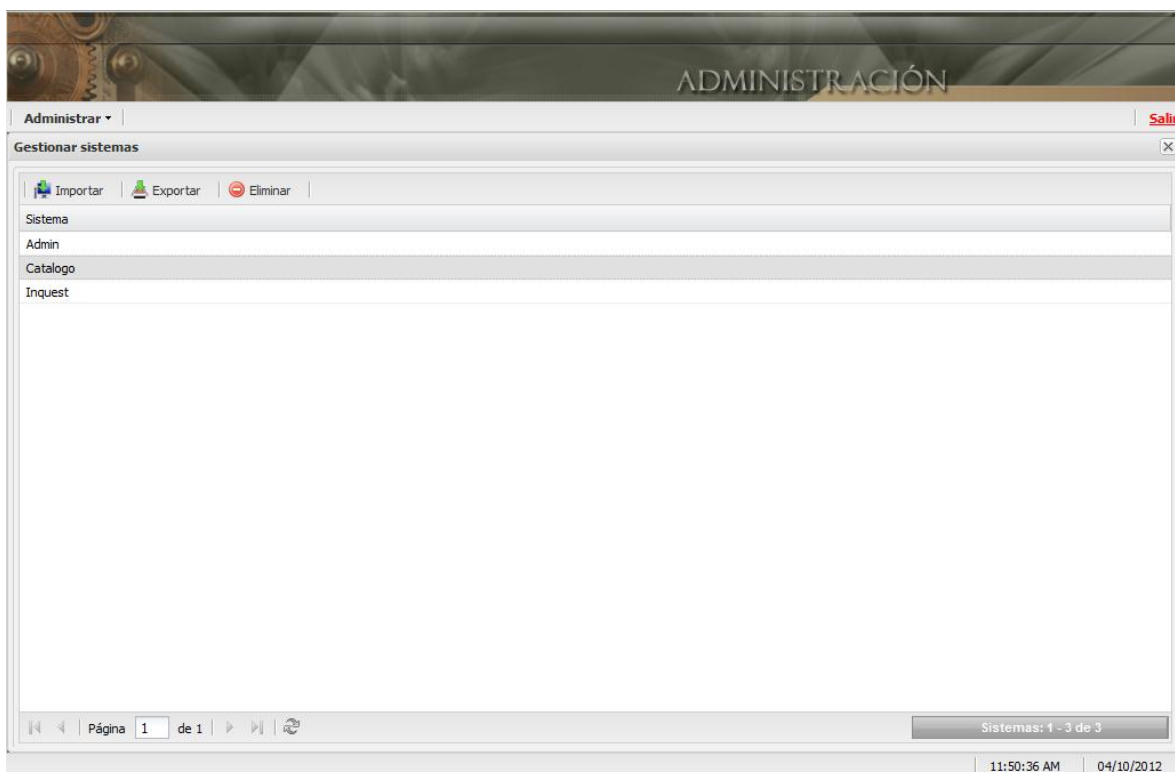
El proceso de selección sigue algunas reglas:

- ✓ Si se habilita un módulo, se habilitan todas las funcionalidades.
- ✓ Si se habilita una funcionalidad en un módulo que no está habilitado, se habilita también el módulo, pero no el resto de las funcionalidades.
- ✓ Si se deshabilita un módulo se deshabilitan todas las funcionalidades habilitadas en él.

- ✓ Es posible tener acceso a un módulo y a ninguna de sus funcionalidades. De esta forma el usuario solo podrá ver el panel principal del módulo sin realizar operación alguna. Para lograr esto se habilita una funcionalidad de un módulo deshabilitado, con ello se habilita el módulo también y luego se deshabilita la misma funcionalidad. De esta forma queda habilitado el módulo solamente.

6.2 Administración de subsistemas

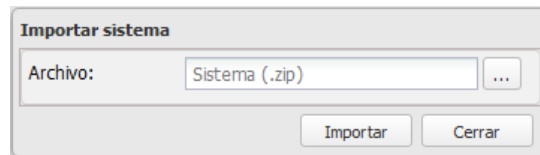
La ventana para la administración de los subsistemas contiene un panel con los subsistemas registrados en la aplicación en general y opciones para importar uno nuevo, exportar o eliminar.



La opción de importar permite incorporar otro subsistema a la aplicación de forma automática. La importación de un nuevo subsistema se realiza mediante un archivo compactado .zip. Internamente se descarga el fichero, se descomprime en la carpeta de los subsistemas y se registra en la base de datos el nuevo subsistema así como sus funcionalidades y sus módulos.

Nota: La base de datos que utilice el nuevo subsistema tiene que ser restaurada de forma manual.

Para importar un nuevo subsistema se oprime el botón Importar. Aparece una ventana donde se pide que seleccione el archivo a importar. Una vez seleccionado se oprime el botón Importar.



La opción Exportar permite crear un archivo compactado con extensión .zip que contiene el subsistema seleccionado. Para ello se selecciona el subsistema a exportar y se oprime el botón Exportar. La aplicación abrirá una nueva página cuyo contenido es el archivo compactado. Es responsabilidad del usuario realizar la salva del mismo.

Finalmente, para eliminar un subsistema se oprime el botón Eliminar. Se mostrará un mensaje de confirmación que en caso de aceptar, se eliminará el subsistema de la base de datos y físicamente.

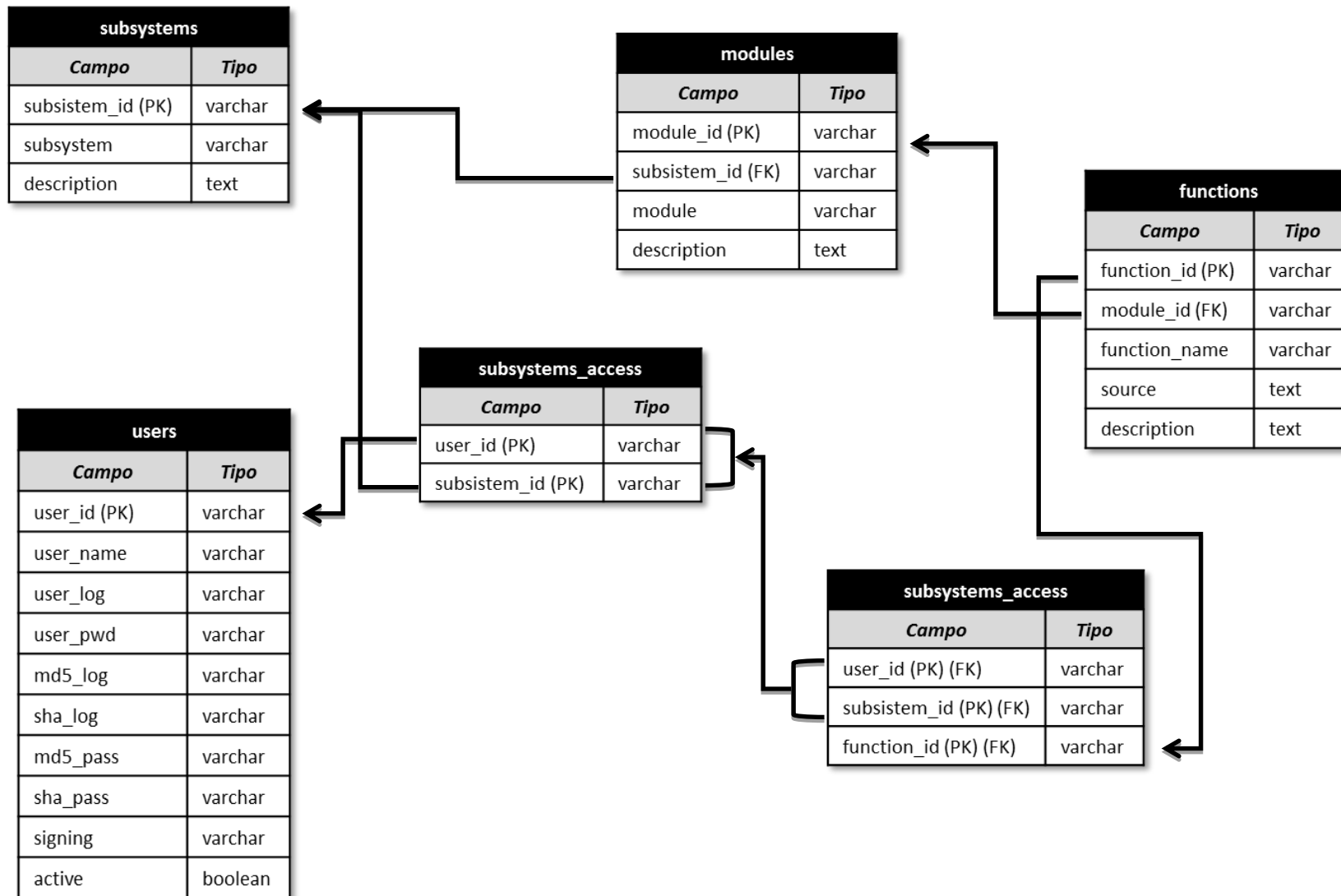
7. Proceso de desarrollo e integración

A continuación se darán una serie de pasos a seguir para indicar al desarrollador la forma en que debe realizar el sistema con vista a la integración en la aplicación final.

1. Crear la carpeta del subsistema que comenzará a realizar.
2. Crear la carpeta de configuración con la estructura mencionada.
3. Crear los ficheros de configuración de la interface visual, (Config\Client\config.xml y Config\Client\Interface.conf.js) aunque sea con imágenes de prueba, y el fichero de configuración de las conexiones a base de datos (Config\Server\db_access).
4. Crear el primer módulo.
 - 4.1. Crear la carpeta con el nombre del módulo.
 - 4.2. Crear los directorios para la parte cliente (Client\js) y para la parte Servidor (Server).
 - 4.3. Crear el fichero principal del módulo en el cliente y comenzar su implementación.
 - 4.4. Crear el fichero php en la parte servidor que es utilizado desde el cliente y comenzar su implementación.
 - 4.5. Registrar el módulo en el fichero de configuración del subsistema (Config\Server\config.xml). O sea, adicionar el módulo que se está configurando.
5. Modificar la forma de ejecución de la aplicación a modo de prueba en el fichero App\Config\app_config.php y definir el subsistema que iniciará por defecto.
6. Todas las pruebas se realizan sobre la página main.php
7. Continuar la implementación de las funcionalidades del módulo y registrarlas en el fichero de configuración del subsistema (Config\Server\config.xml).
8. Continuar con la implementación de todos los módulos del subsistema.
9. Probar que funcione correctamente con la seguridad el sistema terminado.
 - 9.1. Actualice el fichero de configuración de la aplicación (App\Config\config.xml) con el subsistema de administración (Admin) y el subsistema que acaba de implementar.
 - 9.2. Cree una base de datos para la seguridad de la aplicación.

- 9.3. Establezca los parámetros de conexión para la base de datos de la seguridad tanto en la aplicación (App\Config\db_access.php) como en subsistema de administración (SubSystems\Admin\Config\Server\db_access.php)
- 9.4. Ejecute desde el navegador el archivo install.php para registrar los dos subsistemas en la base de datos de la seguridad.
- 9.5. Modificar el subsistema que cargará la aplicación por defecto para que cargue Admin.
- 9.6. Cargar la aplicación para entrar directo al subsistema de administración.
- 9.7. Crear un administrador si desea y los usuarios con que entrará al sistema que acaba de implementar. Asignar los permisos que usted considere.
- 9.8. Cambiar el modo de ejecución de la aplicación a modo de prueba.
- 9.9. Actualizar la página index.php para que se pueda seleccionar el subsistema.
- 9.10. Limpiar la caché del navegador y cargar la página index.php.
10. Una vez probado el subsistema con la seguridad y la administración se compacta a .zip para instalarlo donde está corriendo la aplicación final.

Anexo 1. Modelo de base de datos de las tablas que intervienen en la seguridad de la aplicación.



Anexo 2. Código fuente de la implementación de la parte servidor de un módulo

```
<?PHP
class ControladorAgencias extends Module
{

    public function __construct()
    {
        // Se invoca el constructor de la clase padre pasándole como parámetro
        // el nombre de la conexión a base de datos que se usa por defecto.
        parent::__construct('recursos');
    }

    public function ValidateCargarDatos(&$params)
    {
        return true;
    }

    public function CargarDatos(&$params)
    {
        $_data = array();
        $_count = 0;

        $_start = ($params['start']) ? $params['start'] : 0;
        $_limit = $params['limit'];

        // Obtener los datos de la table agencias
        $_table = $this->_dbase->GetTable('agencias');
        $_data = $_table->GetRange($_limit, $_start);
        if(is_null($_data))
            return false;

        $_count = $_table->GetRowCount();
        if($_count == -1)
            return false;

        $result = array( 'success' => true, 'results' => $_count, 'rows' => $_data);

        return $result;
    }

    public function ValidateAdicionar(&$params)
    {
        $agencia = $params['agencia'];
        $ag = $params['agencia'];
        $objeto_social = $params['objeto_social'];
    }
}
```

```

// Validar la cantidad de caracteres del nombre de la agencia
$agencia = trim($agencia);
$_valid = Validator::CheckStringSize($agencia, 3, 64);
if(!$_valid)
{
    $this->RegisterError('Parámetro incorrecto', "Nombre de la agencia incorrecto.");
    return false;
}

// Limpiar de inyecciones SQL
Validator::ToCleanSQL($agencia);
$params['agencia'] = $agencia;

// Validar la cantidad de caracteres del nombre del objeto social de la agencia
$objeto_social = trim($objeto_social);
$_valid = Validator::CheckStringSize($objeto_social, 4, 1024);
if(!$_valid)
{
    $this->RegisterError('Parámetro incorrecto', "Objeto social incorrecto.");
    return false;
}

// Limpiar de inyecciones SQL
Validator::ToCleanSQL($objeto_social);
$params['objeto_social'] = $objeto_social;

// Validar que el nombre de la agencia no existe
$agencias = $this->_dbase->GetTable('agencias');
$count = $agencias->Contains(array('agencia'=>"$agencia"));

// Si el resultado es menor que 0, ocurrió un error interno
if($count < 0)
    return false;

// Si el resultado es mayor que 0, ya existe la agencia
elseif ($count > 0)
{
    $this->RegisterError('Operación no válida', "La agencia '$ag' ya existe.");
    // En el mensaje se muestra la misma agencia que fue enviada desde el cliente para no
    // mostrar al usuario que se trató un posible inyección SQL

    return false;
}

return true;
}

```

```

public function Adicionar(&$params)
{
    // Optener los parámetros
    $agencia = $params['agencia'];
    $objeto_social = $params['objeto_social'];

    $agencias = $this->_dbase->GetTable('agencias');

    $_result = $agencias->InsertValues(array
    (
        'agencia' => "$agencia",
        'objeto_social' => "$objeto_social"
    ));

    if($_result <= 0)
        return false;

    return true;
}

public function ValidateModificar(&$params)
{
    $agencia = $params['agencia'];
    $ag = $params['agencia'];
    $nueva_agencia = $params['nueva_agencia'];
    $nag = $params['nueva_agencia'];
    $objeto_social = $params['objeto_social'];

    // Validar la cantidad de caracteres de la nueva agencia
    $nueva_agencia = trim($nueva_agencia);
    $_valid = Validator::CheckStringSize($nueva_agencia, 3, 16);
    if(!$_valid)
    {
        $this->RegisterError('Parámetro incorrecto', "El nombre de la nueva agencia es incorrecto.");
        return false;
    }

    Validator::ToCleanSQL($nueva_agencia);
    $params['nueva_agencia'] = $nueva_agencia;

    // Validar la cantidad de caracteres del objeto social
    $objeto_social = trim($objeto_social);
    $_valid = Validator::CheckStringSize($objeto_social, 4, 1024);
    if(!$_valid)
    {
        $this->RegisterError('Parámetro incorrecto', "Objeto social incorrecto.");
        return false;
    }
}

```

```

Validator::ToCleanSQL($objeto_social);
$params['objeto_social'] = $objeto_social;

Validator::ToCleanSQL($agencia);
if($agencia === $nueva_agencia)
    return true;

// Validar que la agencia antigua existe
$agencias = $this->_dbase->GetTable('agencias');
$count = $agencias->Contains(array('agencia'=>"$agencia"));

// Si el resultado es menor que 0, ocurrió un error interno
if($count < 0)
    return false;

// Si el resultado es igual a 0, no existe la agencia
elseif ($count == 0)
{
    $this->RegisterError('Operación no válida', "La agencia '$ag' no existe.");
    return false;
}

// Validar que el nombre de la nueva agencia no exista
$count = $agencias->Contains(array('agencia'=>"$nueva_agencia"));

// Si el resultado es menor que 0, ocurrió un error interno
if($count < 0) return false;

// Si el resultado es mayor que 0, ya existe la agencia
elseif ($count > 0)
{
    $this->RegisterError('Operación no válida', "La agencia '$ag' ya existe.");
    return false;
}

$params['agencia'] = $agencia;
$params['nueva_agencia'] = $nueva_agencia;
$params['objeto_social'] = $objeto_social;

return true;
}

public function Modificar(&$params)
{
    $agencia = $params['agencia'];
    $nueva_agencia = $params['nueva_agencia'];
    $objeto_social = $params['objeto_social'];

```



```

$agencias = $this->_dbase->GetTable('agencias');

$update_count = $agencias->Update(
    array(
        'agencia' => "$nueva_agencia",
        'objeto_social'=>"$objeto_social"
    )
    , "agencia = '$agencia'");

// Comprobando el proceso de actualización
if($update_count < 0)
    return false;

return true;
}

public function ValidateEliminar(&$params)
{
    $agencia = $params['agencia'];
    $ag = $params['agencia'];

    // Limpiar de inyecciones SQL
    Validator::ToCleanSQL($agencia);

    /// Validar que existe la agencia
    $agencias = $this->_dbase->GetTable('agencias');
    $count = $agencias->Contains(array('agencia'=>"$agencia"));

    // Si el resultado es menor que 0, ocurrió un error interno
    if($count < 0)
        return false;
    // Si el resultado es igual a 0, no existe la agencia
    elseif ($count == 0)
    {
        $this->RegisterError('Operación no válida', "La agencia '$ag' no existe.");
        return false;
    }

    // Obtengo el identificador de la agencia
    $id_agencia = $agencias->GetValueWhere('id_agencia', "agencia = '$agencia'");
    if(is_null($id_agencia)) return false;

    /// Validar que no existan empleados en la agencia
    $empleados = $this->_dbase->GetTable('empleados');
    $count = $empleados->Contains(array('id_agencia' => $id_agencia));

    // Si el resultado es menor que 0, ocurrió un error interno
    if($count < 0)
        return false;

```

```

        // Si el resultado es mayor que 0, la agencia tiene trabajadores
        elseif ($count > 0)
        {
            $this->RegisterError('Operación no válida', "La agencia '$ag' cuenta con trabajadores.
Imposible eliminarla.");
            return false;
        }

        // En el parámetro 'agencia' se pone el identificador de la agencia
        // para hacer más fácil la eliminación
        $params['agencia'] = $id_agencia;

        return true;
    }

    public function Eliminar(&$params)
    {
        $id_agencia = $params['agencia'];

        $agencias = $this->_dbase->GetTable('agencias');

        $_result = $agencias->DeleteWhere("id_agencia = '$id_agencia'");
        if($_result <= 0)
            return false;

        return true;
    }
}
?>

```