TEMA 5: GENERACIÓN DINÁMICA DE PÁGINAS WEB

Contenido

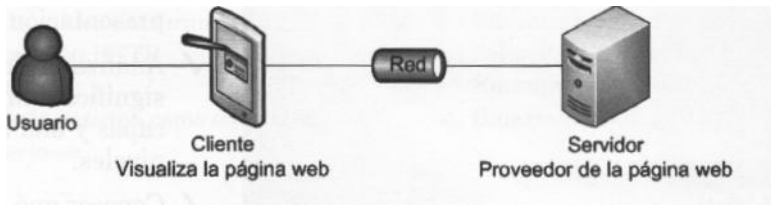
1.- INTRODUCCIÓN

La generación dinámica de páginas es uno de los grandes avances dentro del desarrollo de aplicaciones en entorno web. La gran ventaja de esta característica es permitir la independencia entre el diseño de páginas web y la lógica del negocio. La creación de páginas web ha evolucionado de manera paralela al desarrollo de la Web. Las primeras páginas web, escritas íntegramente en código HTML, coinciden con la denominada Web 1.0 la cual está dominada por los contenidos estáticos. En esta primera etapa se utiliza el lenguaje de marcado HTML el cual mezcla el diseño y el contenido de una página web a través de un conjunto de etiquetas. Las etiquetas permiten marcar tres tipos de elementos dentro de una página web:

- Características de formato de un elemento de la página (p. ej. para negrita en textos).
- Elementos estructurales de la página web (p. ej. <head> o <body>).
- Identificar elementos de hipervínculo. (p. ej. <a href>).

Esta primera etapa de la Web consolida una nueva labor dentro de la informática: los denominados webmasters. Los "maestros de la Web" son los encargados de la creación y mantenimiento de las nuevas Webs en los servidores web y son los poseedores del poder de publicación y actualización de contenidos en la Red.

El éxito de las páginas web trae consigo la introducción de los primeros elementos de interacción dentro de una página web: los formularios. Un formulario está constituido por cajas de texto, elementos de opción y de selección y botones que piden información al usuario desde el equipo cliente. A través de un evento (POST o GET) dicha información es enviada al servidor con el fin de realizar un conjunto de tareas que darán un resultado el cual es mostrado al usuario a través de una nueva página web. Ejemplos de este tipo de dinamismo en las páginas es el registro de un usuario en un servicio web (Hotmail, Gmail, etc.) a través de un formulario presentado por medio de una página web. La estrategia dinámica permite observar más claramente los dos componentes involucrados en la creación de páginas web: de un lado el servidor o proveedor de la página web y por otro el cliente o visualizador de la página web.



En esta arquitectura la lógica del negocio está totalmente separada de la página web, sin embargo, el diseño sigue estando atado a etiquetas HTML, a código estático y al servidor que es quien provee la página. La siguiente evolución en la creación de páginas web dinámicas es el código embebido por el cual una página ya no solo contiene código HTML, sino también posee instrucciones de lenguajes como JSP o PHP que se embeben en el código dando inteligencia al lado del servidor. Este tipo de código se denomina embebido porque está sumergido en el código

HTML, es decir, se mezcla con él. El éxito del código embebido es que este se ejecuta en el servidor y además, que dicha ejecución se realiza siempre que una página con código embebido es pedida. Esto quiere decir, que gracias al código embebido el servidor puede dar distintas respuestas con la misma página. El código embebido podrá leer cuál ha sido la selección en el momento en el que se ejecuto la página o las preferencias del usuario y de acuerdo a dichos parámetros arrojará resultados distintos y de hecho, la página web mostrará valores distintos. La última etapa en el dinamismo de las páginas web está dada por la inteligencia no solo en el lado del servidor sino también en el lado del cliente, por ejemplo, AJAX en el cual las páginas tienen la potestad de cambiar de acuerdo a la petición del usuario no solo desde el lado servidor sino también desde el lado cliente.

Todas estas evoluciones han permitido que la nueva Web sea más dinámica y que los contenidos no sean publicados únicamente por expertos webmasters sino por cualquier usuario que tenga mínimos conocimientos de informática y posea una interfaz adecuada de edición de código web. Además, se ha logrado la independencia entre el diseño de la página (menús, imágenes, colores de la página) y la lógica del negocio, es decir, los procesos que deben ejecutarse para lograr la respuesta deseada por el usuario final se ejecutan de manera independiente de la forma en la que se muestra la página. Esto ha permitido una especialización dentro de los profesionales involucrados en la creación de páginas web, ya que permite que los diseñadores se dediquen exclusivamente a cambiar el formato de los títulos y demás aspectos estructurales de la página web sin interferir en la labor de los desarrolladores de funcionalidad que se encargan de analizar, diseñar y construir los programas de acuerdo al tipo de servicio requerido. Esto permite que haya profesionales que se dedique a escribir los contenidos, otros al diseño de la página y otros a la inteligencia del negocio. Tres roles diferentes fusionados para crear un mismo elemento visual, una página web. Ejemplo de esta separación de roles puede ser la edición digital de un diario: los periodistas se encargan de los contenidos, los diseñadores de la página web se encargan de organizar la forma en la cual se presentarán las noticias y los informáticos se encargarán de que las funcionalidades de la página web como son los elementos interactivos, vídeos, RSS o similares se ejecuten correctamente. Uno de los tipos de herramientas que permiten administrar esta división de roles son los gestores de contenido o CMS, como por ejemplo, Joomla, donde se separa la imagen del contenido.

Pero, ¿cómo se puede reflejar esa independencia del diseño de página y la lógica del negocio a la hora de construir una aplicación o servicio web? La siguiente sección nos presenta las diferentes estrategias para dividir la lógica del negocio del diseño de las páginas web.

2.- MECANISMOS DE SEPARACIÓN DE LA LÓGICA DE NEGOCIO

Como ya hemos dicho anteriormente, la creación de páginas web dinámicas permite la separación entre el diseño de la página y la lógica del negocio. Esta separación es una decisión de tipo arquitectónico, que tiene repercusiones tanto a nivel del modelo físico como del modelo lógico en la construcción de la aplicación web.

A nivel del modelo físico, los sistemas web actuales están guiados por las arquitecturas

multinivel (del inglés, multitier). Este tipo de arquitectura propone distribuir la infraestructura, es decir los elementos de hardware en los que se ejecutarán los procesos y que constituyen el sistema, en niveles. Así, cada tipo de elemento que tenga un rol distinto representará una capa diferente. Este tipo de arquitecturas las explicaremos en la sección 2.1.

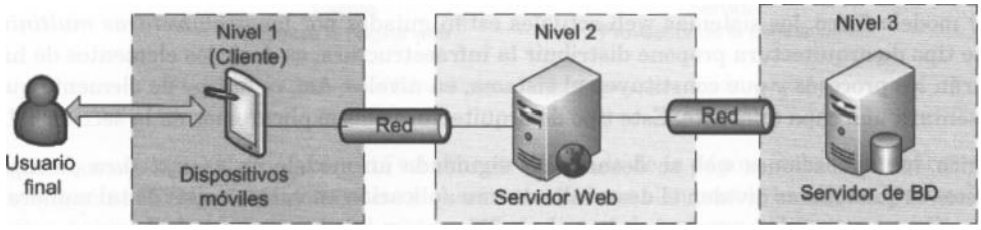
A nivel lógico, las aplicaciones web se desarrollan siguiendo un modelo de arquitectura de capas (del inglés, multilayer). Estas arquitecturas dividen el desarrollo de una aplicación en varias capas de tal manera que el trabajo de cada capa puede ser asignado a un equipo de trabajo diferente y solo basta con que el equipo conozca el modo de comunicarse con las otras capas que componen el sistema desarrollado. De manera concreta, estas arquitecturas dividen la forma en la que se organiza el código de la aplicación sin tener en cuenta su distribución física, es decir podemos decidir tener toda la aplicación ejecutándose sobre una misma máquina o dividirlo en diferentes equipos sin afectar nuestra visión de la arquitectura.

Es importante resaltar la diferencia entre nivel y capa, ya que muchas veces las dos palabras son utilizadas de manera indistinta. Sin embargo, veremos cómo un **nivel** corresponde a la forma física en la que se organiza una aplicación mientras que una **capa** hace referencia a las distintas partes en que se divide una aplicación.

2.1.- MODELOS FÍSICOS DE SEPARACIÓN: ARQUITECTURAS MULTINIVEL

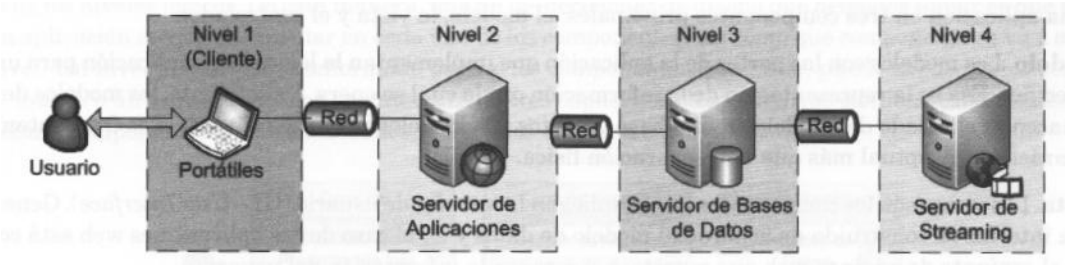
Cuando hablamos del modelo físico arquitectónico o de la arquitectura física, nos referimos a la forma en la cual se distribuye la infraestructura, es decir, los elementos de hardware en los que se ejecutarán los procesos y que constituyen el sistema. De manera genérica, los sistemas web actuales están guiados por las arquitecturas multinivel (del inglés, multitier). Este modelo de arquitectura deriva de la arquitectura Cliente-Servidor y proponen distribuir, los elementos arquitectónicos de infraestructura en niveles. Así, cada tipo de elemento que tenga un rol distinto representará una capa diferente. En el caso web, la mayoría de los sistemas presentan una arquitectura de dos o más niveles. Una arquitectura de dos niveles significa que las funciones estarán divididas en dos equipos de cómputo diferentes y equivale al modelo cliente-servidor tradicional. Ejemplo de una arquitectura de dos capas es un modelo en el cual el nivel uno representa al cliente quien a través de algún dispositivo de acceso a Internet (portátil, teléfono móvil, tablet, PDA) se comunica con el nivel dos en el cual se encuentra el equipo servidor que ejecutará los procesos y devolverá la respuesta al cliente.

Si la arquitectura física del sistema es de tres niveles quiere decir que tenemos otro elemento hardware involucrado en el funcionamiento de nuestro sistema. Por lo general, este nuevo elemento desdobla las funciones del servidor es decir, el nuevo nivel casi siempre representa a un nuevo servidor que descarga funcionalidad y carga de trabajo al servidor único. La siguiente figura representa una arquitectura de tres niveles.



En este caso el servidor único de la arquitectura en dos niveles se desdobla en dos equipos: un servidor web (nivel 2) encargado de ofrecer las prestaciones para aceptar y contestar peticiones web y un servidor de base de datos (nivel 3) encargado de almacenar y gestionar el acceso a la información.

Las arquitecturas multinivel pueden tener tantos niveles como equipos involucrados en el manejo de la información y el procesamiento existan. La siguiente figura muestra otra arquitectura multinivel de 4 niveles. Se puede ver que aunque tiene los mismos niveles del ejemplo anterior, los roles de cada nivel no son el mismo para cada ejemplo.



En el primer caso, existen dos equipos diferentes, el servidor web y el servidor de BD encargados de la gestión y el procesamiento de las peticiones recibidas de parte del cliente. Este caso es otro ejemplo de cómo la división en niveles corresponde a una división funcional, de manera que un equipo es destinado a una función particular y así crea un nuevo nivel en la arquitectura. Además, este ejemplo nos sirve para ilustrar la diferencia funcional entre un servidor web y un servidor de aplicaciones. Para aclarar los conceptos hagamos un poco de historia. Cuando se crearon los primeros servidores de páginas web (web servers), como Apache, su única misión era recuperar páginas web estáticas de su disco duro y enviárselas al cliente. Para cualquier otro tipo de información que debiera generarse de manera dinámica (respuestas a búsquedas, etc.) el servidor tenía que ceder el control a algún tipo de código externo mediante CGI el cual ejecutaba un script (programa). Con el paso del tiempo el uso de servidores web se generalizó y se hizo necesario incrementar los servicios ofrecidos ya que llamar a un interpretador para que ejecutara otro programa suponía una demanda muy fuerte sobre el equipo que mantenía el servidor de páginas (servidor web). Finalmente la evolución ha llevado a crear un nuevo término: servidor de aplicaciones (Application server). Hoy en día, se puede decir que todos los servidores web actuales son también servidores de aplicaciones, ya que incluyen alguna tecnología (CGI, PHP, JSP, etc.) que permite realizar aplicaciones que generan contenido dinámico o aplicaciones de servidor. Dependiendo de la funcionalidad, se incrementa la complejidad del sistema, ya sea en la forma de requerimientos del sistema (memoria, procesadores), carga administrativa (configuración, tiempo de desarrollo) o alguna otra.

Teniendo en cuenta que la tendencia de que cualquier servidor web sea un servidor de aplicaciones, el segundo caso muestra un ejemplo de arquitectura de 4 niveles con este rol.

2.2.- MODELOS DE SEPARACIÓN LÓGICOS

En contraste con los modelos físicos, cuando hablamos del modelo arquitectónico lógico o de la arquitectura lógica del sistema, nos referimos a la forma, en la cual elegimos dividir el software para obtener el mejor rendimiento del sistema. En los modelos lógicos se distribuyen los componentes software del sistema. Y así como pueden existir sistemas físicos distribuidos también pueden existir sistemas lógicos distribuidos.

Uno de los mecanismos más usado a la hora de construir páginas web es el uso de las arquitecturas de capas (del inglés, multilayer). Estas arquitecturas dividen el desarrollo de una aplicación en varios niveles lógicos. Estos niveles son formas de organizar el código. Hay que aclarar que de ninguna manera esta división en capas implica ejecutar cada una de las capas en diferentes equipos de cómputo. La construcción de software, y específicamente de páginas web, por capas trae como principal ventaja la organización y agrupamiento por funcionalidad. Esta agrupación ofrece reusabilidad, facilidad de mantenimiento y ciclos de desarrollo más cortos. Todas estas características redundan en una reducción de los costos de desarrollo y de mantenimiento de la aplicación. Otras ventajas del modelo son:

- Desarrollos paralelos en cada capa.
- Aplicaciones más robustas debido al encapsulamiento.
- Mantenimiento y soporte más sencillo. Es más fácil cambiar un componente que modificar una aplicación monolítica.
- Mayor flexibilidad. Podemos añadir nuevos módulos para dotar al sistema de nueva funcionalidad.
- Alta escalabilidad. Podemos manejar muchas peticiones con el mismo rendimiento simplemente añadiendo más hardware. El crecimiento es casi lineal y no es necesario añadir más código para conseguir esta escalabilidad.

Las arquitecturas en capas son una de las formas de implementación de un patrón arquitectónico denominado *Modelo-Vista-Controlador* (MVC).

2.2.1.- EL ESQUEMA MODELO-VISTA-CONTROLADOR (MCV)

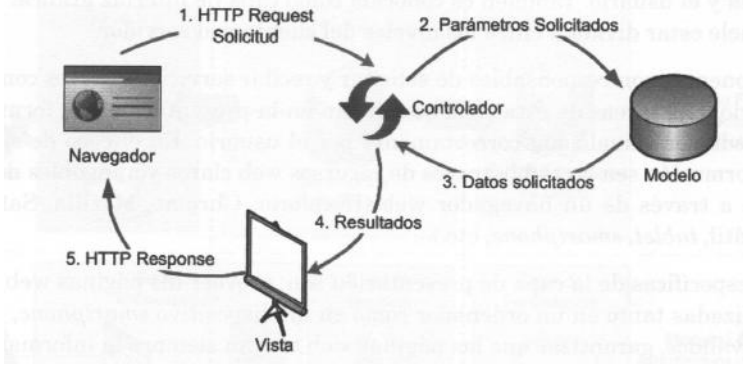
Este es uno de los esquemas más seguidos e implementados a la hora de construir aplicaciones, no solo a nivel web, sino también aplicaciones de sobremesa. Una arquitectura siguiendo el esquema Modelo-Vista-Controlador busca separar una aplicación en tres componentes principales: el modelo, la vista y el controlador.

- **Modelo.** Los modelos son las partes de la aplicación que implementan la lógica de la

aplicación para un dominio específico. Esa es la representación de la información con la cual se opera. Usualmente, los modelos devuelven y almacenan el estado del modelo en una base de datos. En modelos pequeños el modelo es frecuentemente una separación conceptual más que una separación física.

- **Vista.** Las vistas son los componentes que despliegan la interfaz de usuario (UI - User Interface). Generalmente, esta interfaz es construida de acuerdo al modelo de datos y en el caso de las aplicaciones web está constituido por el conjunto de páginas web que muestran y recogen la información del usuario.
- **Controlador.** Los controladores son los componentes que manejan la interacción con el usuario, trabajan con el modelo y seleccionan cuál es la vista correcta a desplegar para mostrar la información.

El patrón MVC nos ayuda a crear aplicaciones que separan los diferentes aspectos de la aplicación, la lógica de entradas, la lógica de negocio y la lógica de interfaz, generando un acoplamiento pequeño entre cada una de estas partes. Así el patrón especifica que cada una de estas lógicas debe corresponder con un elemento. La lógica de interfaz pertenecerá a la vista, la lógica de entradas pertenecerá al controlador y la lógica del negocio estará reflejada en el modelo. Esta separación ayuda al creador de una aplicación web a manejar la complejidad del desarrollo ya que le permite centrarse en solo un aspecto del sistema a la vez. El bajo acoplamiento de estos elementos promueve el desarrollo en paralelo. La siguiente figura muestra gráficamente este esquema en el entorno de una aplicación web.

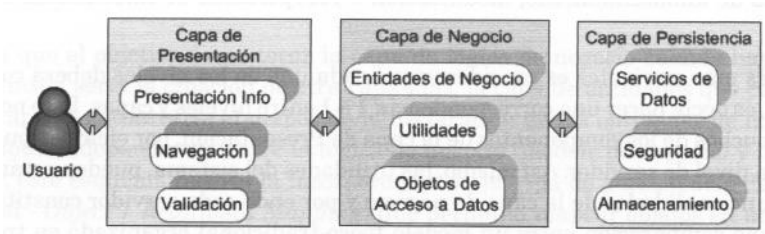


Muchas de las soluciones que se ofrecen actualmente siguen este modelo o alguna de sus variaciones. De hecho, las arquitecturas en capas constituyen una derivación de este esquema.

2.2.2.- LA ARQUITECTURA DE 3 CAPAS

Dentro de las arquitecturas multicapa, la arquitectura de 3 capas es una de las más utilizadas para el desarrollo de sitios web. Esta arquitectura divide la creación de una aplicación en tres niveles: capa de presentación, capa de la lógica de negocio y capa de persistencia (almacenamiento de datos). Está basado en el concepto de que todos los niveles de una aplicación son una colección de componentes que se proporcionan servicios entre sí o a otros niveles adyacentes. El modelo de tres capas está destinado a ayudar a construir componentes de software a partir de la

división de los niveles lógicos. De esta manera, una de las decisiones de diseño que debemos tomar es que parte de la lógica de la aplicación se va a encapsular en cada uno de los componentes, así como que componente se va a encapsular en cada nivel. Un nivel puede estar conformado por varios componentes, por tanto, puede suplir varios servicios. La siguiente figura muestra un esquema de una arquitectura de tres capas. Los elementos que contiene cada una de las capas corresponden a componentes o funcionalidades manejados en dicha capa.



A continuación realizaremos una breve explicación de cada una de las capas de esta arquitectura.

Capa de presentación

Esta capa es la más cercana al usuario presentándole una interfaz gráfica del recurso solicitado y capturando la interacción entre el sistema y el usuario. También es conocida como capa de interfaz gráfica. Esta capa se comunica con la capa de negocio y suele estar dividida entre los niveles del cliente y el servidor.

En esta capa, los componentes son responsables de solicitar y recibir servicios de otros componentes de la misma capa o de la capa de negocio. Las tareas de esta capa se centran en la presentación y el formateo de la información enviada para que esta pueda ser visualizada correctamente por el usuario. En el caso de sistemas web, esta capa debe garantizar que la información sea accesible través de recursos web claros y amigables de tal manera que dicha información sea accesible a través de un navegador web (lExplorer, Chrome, Mozilla, Safari, etc.) en cualquier dispositivo de acceso (portátil, *tablet*, *smartphone*, etc.).

Algunas de las tareas específicas de la capa de presentación son: proveer las páginas web, garantizar que dichas páginas puedan ser visualizadas tanto en un ordenador como en un dispositivo *smartphone*, gestionar que todos los enlaces del sitio web sean válidos, garantizar que las páginas web tengan siempre la información actualizada, etc.

Capa de negocio

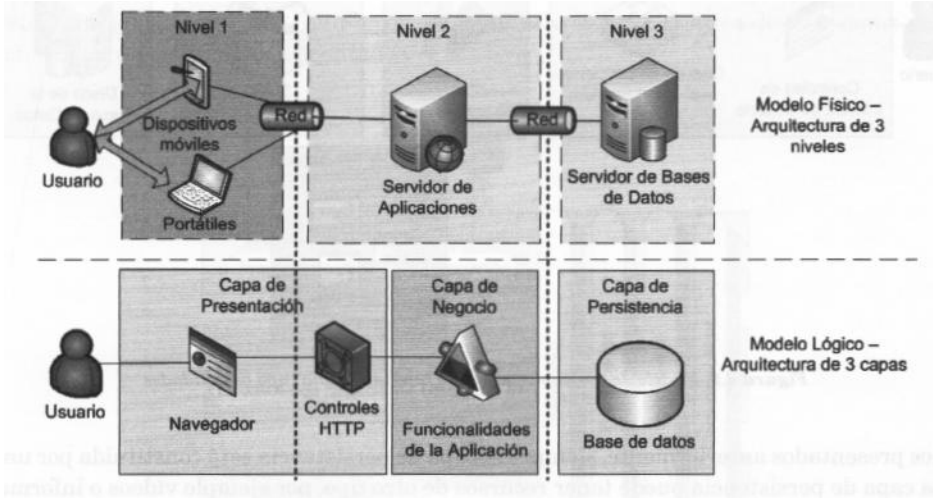
También denominada capa de lógica, capa de aplicación, capa media o capa de lógica de negocio. Es la capa que gestiona las funcionalidades del sistema o aplicación web ("lógica o reglas de negocio'). Habitualmente esta capa recibe las peticiones del usuario y desde ella se envían las respuestas apropiadas tras el procesamiento de la información proporcionada por el cliente. Se denomina capa de negocio porque es aquí donde se establecen todas las reglas que deben cumplirse para que las funciones de la aplicación web sean ejecutadas correctamente. La implementación de dichas reglas se hace a través de desarrollos software los cuales se ejecutan cuando llega la

petición de un usuario. Al igual que la capa de presentación, la lógica de negocio puede ser programada tanto en el entorno cliente como en el entorno servidor. Algunas de las tareas concretas de los componentes de esta capa son: coordinar la aplicación web, procesar los comandos, realizar la toma de decisiones, realizar los cálculos respectivos. Además, esta capa es la encargada de mover y procesar los datos entre sus dos capas adyacentes. Típicamente, la capa del negocio está conformada por uno o más módulos que se ejecutan en un equipo de cómputo denominado servidor de aplicaciones.

Capa de persistencia

Esta capa es la encargada del acceso a los datos. Aquí se encuentra toda la codificación necesaria tanto para el acceso a los datos como para el manejo y almacenamiento de los mismos. Normalmente está formada por un conjunto de componentes que se encargan del acceso a los datos (p. ej. clases de conexión a la base de datos) los cuales se ejecutan en los servidores de aplicaciones y por uno o más sistemas gestores de bases de datos (SGBD) que se ejecutan en el nivel de datos (servidores de bases de datos) los cuales se encargan de todo el proceso de administración de datos, ejecutando las solicitudes de almacenamiento, modificación o recuperación de información dadas desde la capa de negocio. Los SGBD.

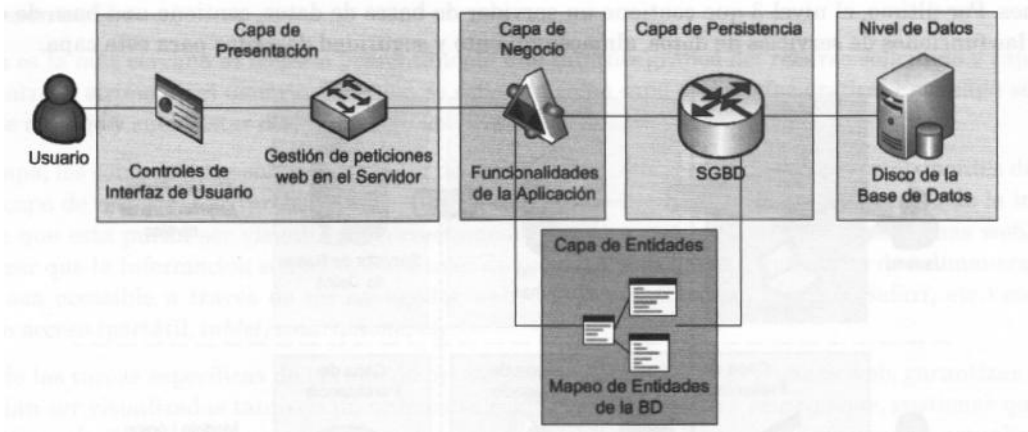
Una de las confusiones más corrientes es considerar que cada uno de los niveles deberá corresponder a una capa dentro de la arquitectura, es decir, hacer una correspondencia 1 a 1 entre niveles y capas. Esto no es necesariamente así. Como hemos explicado, muchos de los componentes de la capa de Presentación, por ejemplo, pueden estar distribuidos a nivel de cliente y otros a nivel de servidor. Así mismo, las utilidades del sistema, pueden están distribuidos en varios servidores, es decir, las funcionalidades de la capa de negocio y por ende cada servidor constituirá un nivel diferente. La Figura 4.8 muestra una comparación entre un modelo físico tradicional organizado en tres niveles y un modelo lógico de tres capas. En este caso, el nivel 1 del modelo físico se encarga de algunas funcionalidades de navegación y presentación a través del browser. El nivel dos, correspondiente al servidor de aplicaciones contiene todas las funcionalidades de la capa de negocio y algunas de la capa de presentación, como por ejemplo la validación o el control
de conexiones. Por último, el nivel 3 que contiene un servidor de bases de datos, contiene una base de datos que se encarga de las funciones de servicios de datos, almacenamiento y seguridad descritos para esta capa.



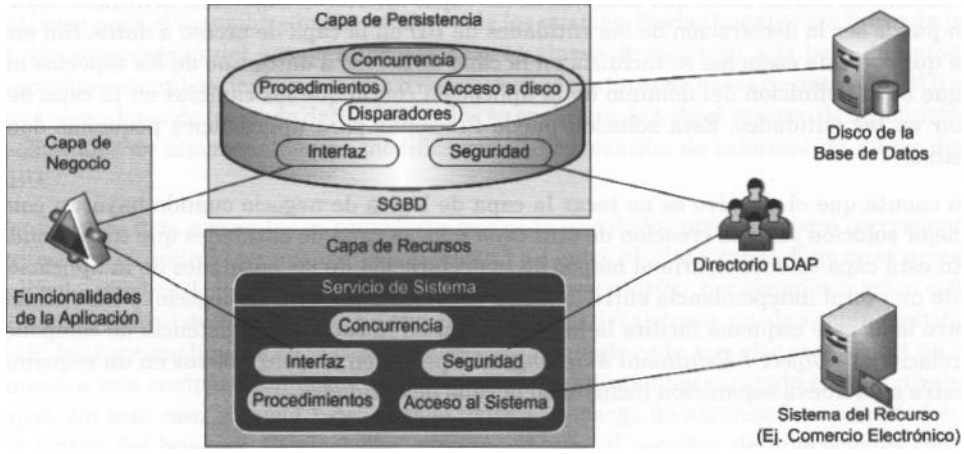
2.2.3.- ARQUITECTURAS MULTI-CAPA

Las nuevas tendencias en el desarrollo web así como el uso de la programación orientada a objetos crean nuevas formas de separación de la lógica del negocio.

Con el modelo de tres capas si se desea cambiar la interfaz gráfica de una aplicación web, este cambio solo afectará a los componentes de la capa de presentación sin afectar al resto de componentes de la aplicación. Sin embargo, en el momento de analizar el sistema y programarlo, los componentes de la capa lógica necesitan referenciar instancias de las clases de dominio (las que representan las entidades del negocio y que posteriormente se plasmarán en entidades en la base de datos) y los componentes de la capa de acceso a datos también tienen que referenciarlas para poder "rellenar" tales instancias con los datos que obtienen de las capas inferiores. Siguiendo la arquitectura de 3 capas, una posible solución puede ser la declaración de las entidades de BD en la capa de acceso a datos. Sin embargo, hay que tener en cuenta que con este esquema se incluiría en la capa de acceso a datos uno de los aspectos más importantes del desarrollo que es la definición del dominio de la aplicación con lo que los cambios en la capa de acceso a datos pueden impactar en las entidades. Esta solución puede funcionar para aplicaciones pequeñas donde los cambios pueden controlarse.
Teniendo en cuenta que el objetivo es no tocar la capa de lógica de negocio cuando haya un cambio en el nivel de datos, una mejor solución sería la creación de otra capa más, la capa de entidades que corresponde al dominio de la aplicación. En esta capa se encontraría el mapeo de la declaración de las entidades de la aplicación. Además este esquema permite una total independencia entre la lógica de negocio (modelo de negocio) y las entidades (modelo del dominio). Por otro lado, este esquema facilita la incorporación en la capa de persistencia de componentes tipo ORM (Mapeo objeto/relacional - *Object / Relational Mapping)* que permiten mapear objetos en un esquema relacional. La siguiente figura muestra esta nueva separación incluyendo la capa de entidades.



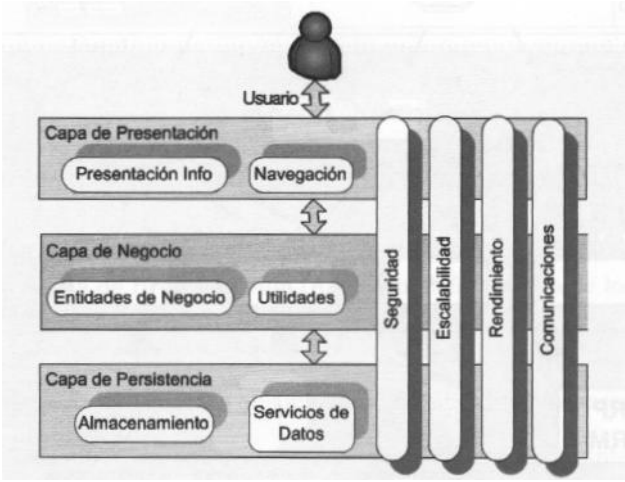
En los ejemplos presentados anteriormente, siempre la capa de persistencia está constituida por una base de datos. Sin embargo, esta capa de persistencia puede tener recursos de otro tipo, por ejemplo vídeos o información particular tratada de manera independiente. En estos casos, esta capa cambia su nombre a capa de recursos. En esta nueva capa, los recursos pueden ser ya establecidos como recursos de persistencia traducidos en sistemas de gestión de bases de datos o nuevos recursos que deberán ser implementados. Por ello, a la hora de diseñar la arquitectura del sistema habrá que tener en cuenta que ese nuevo recurso de la capa de recursos deberá cubrir funciones claves para garantizar que la capa del negocio se pueda conectar con él.



La figura anterior muestra la conexión de la capa de negocio con dos capas: una de persistencia y otra de recursos. En la primera capa representamos un SGBD con sus componentes incluidos dentro del mismo, mientras que la segunda capa representa un servicio no creado donde los componentes ya incluidos dentro de un sistema preestablecido, como un SGBD, deberán ser implementados. En realidad, no es que esas funciones no existan, la cuestión es que cuando hablamos de un SGBD estas funciones ya van inmersas en dicho sistema y por ello son transparentes a nosotros. Pero, si nos conectamos a un sistema que definimos nosotros mismos, estos componentes deberán ser explícitos.

Por último, es importante tener en cuenta que existen un conjunto de aspectos estructurales cuya funcionalidad no se implementen en una sola capa, sino que por el contrario, tienen que estar distribuidos por todas las capas, es decir, son aspectos transversales a toda la

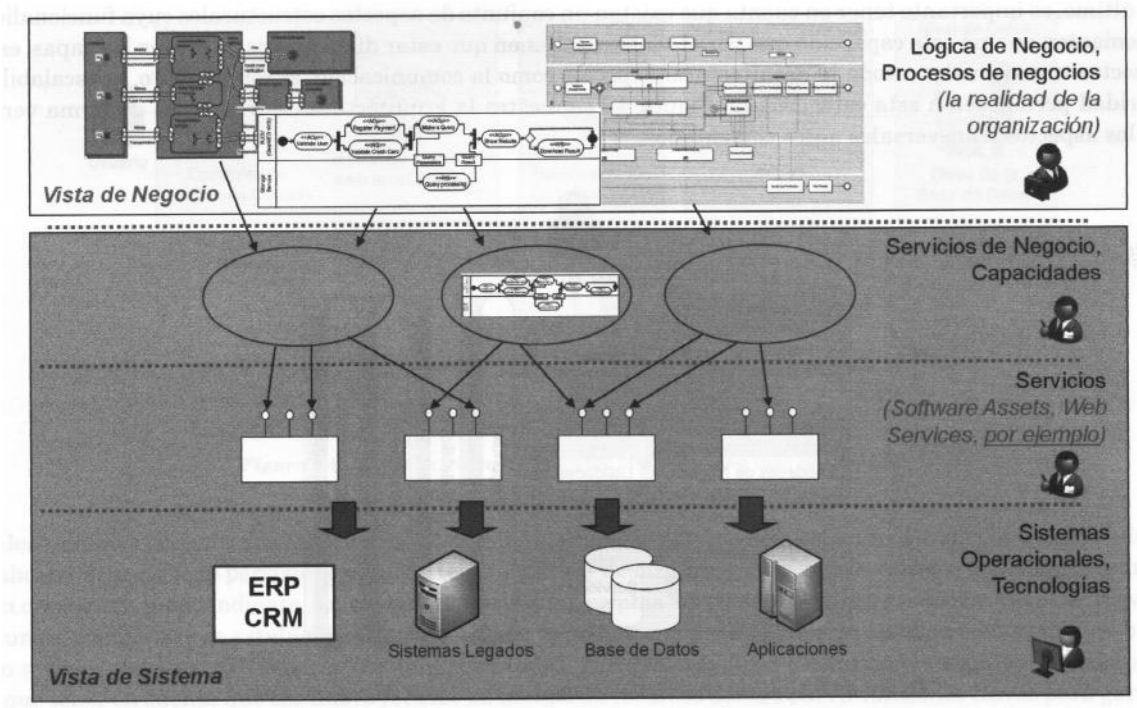
arquitectura. Aspectos como la comunicación, el rendimiento, la escalabilidad y la seguridad pertenecen a esta categoría. La siguiente figura muestra la arquitectura de tres capas de forma vertical y agrega los aspectos transversales antes comentados.



2.2.4.- LA ARQUITECTURA ORIENTADA A SERVICIOS (SOA)

El nivel de cooperación que presentan las organizaciones requiere que las aplicaciones de software desarrolladas por dichas organizaciones interactúen unas con otras. El problema es que algunas de ellas no se ejecutan en la misma plataforma o están desarrolladas con marcos de trabajo diferentes. La solución fue presentada como la arquitectura orientada a servicios (SOA, Service Oriented Architecture), que brinda entre otras cosas una forma estándar de publicar y utilizar servicios, conocidos comúnmente como servicios web (web services). De esta manera una aplicación es vista como un conjunto de servicios. Los servicios web intercambian mensajes en formato XML utilizando protocolos de transporte como HTTP. Los servicios web, básicamente, establecen un lenguaje común mediante el cual distintos sistemas pueden comunicarse entre sí y, de esta forma, facilitan la construcción de sistemas distribuidos heterogéneos. De esta manera, una organización expone sus competencias (funcionalidades o capacidades) para que sean utilizados por la misma organización o por otras organizaciones.

SOA permite emular el comportamiento de los negocios en el mundo real. En una arquitectura orientada a servicios, los usuarios finales, mediante la utilización de hardware y/o software liviano, como un navegador web, pueden acceder a lo que se denomina el nivel de clientes o aplicaciones que básicamente está constituida por la capa de presentación y consumen los servicios publicados por una organización. Este tipo de servicios son en general sitios o portales en la Web. Las aplicaciones de otras organizaciones también pueden utilizar los servicios publicados por una organización concreta; esta acción requiere de acuerdos comerciales y credenciales para autenticar y autorizar a quienes consumen los servicios. El proceso de orientación a servicios puede dividirse en cuatro capas. Estas capas se pueden ver en la siguiente figura



Podemos observar como en la orientación a servicios, la lógica del negocio no es vista como una capa a la hora de realizar la aplicación sino como una capa de análisis en la que se determinan los procesos de negocios. Es decir, la lógica del negocio se refiere a la realidad de la organización, a los procesos que guían la manera en la que se hacen las cosas en el mundo real, a los procesos que especifican los requisitos y proponen soluciones al negocio. Esta labor es realizada por los analistas del negocio o los arquitectos del negocio. Las dos capas de servicios, servicios de negocio y servicios, se centran en comprender cuál es el objetivo del negocio y verifican qué servicios son requeridos para suplir los requerimientos. Además, en estas capas se encargan de crear las estrategias de integración y reutilización de los servicios. Esta labor es realizada por analistas de software o arquitectos de servicios.

Finalmente, la capa de los sistemas operacionales es implementada por analistas de sistemas o desarrolladores de servicios, los cuales entienden claramente cuáles son los pasos para el desarrollo e integración de servicios proponiendo soluciones técnicas óptimas que permitan la implementación tanto de los servicios como de la solución. Es en esta capa donde se desarrollan los servicios, que se denominan servicios web. Estos servicios están soportados bajo protocolos como SOAP (Protocolo de Acceso a Objetos Simples - Simple Object Access Protocol) que es el protocolo de comunicación entre aplicaciones y servicios web a través de mensajes por medio de Internet, UDDI (Universal Discovery Description and Integration) que es un modelo de directorios para servicios web y WSDL (Lenguaje de Descripción de Servicios Web - Web Services Description Language) el cual es un protocolo basado en XML que describe los accesos a un servicio web, es decir, describe las interfaces del servicio web y cómo utilizarlas. Una vez construidos los servicios web, estos podrán ser utilizados y llamados por las páginas web, por ello muchos de los lenguajes de construcción de páginas web dinámicas como PHP o ASP.Net soportan la llamada a servicios web.

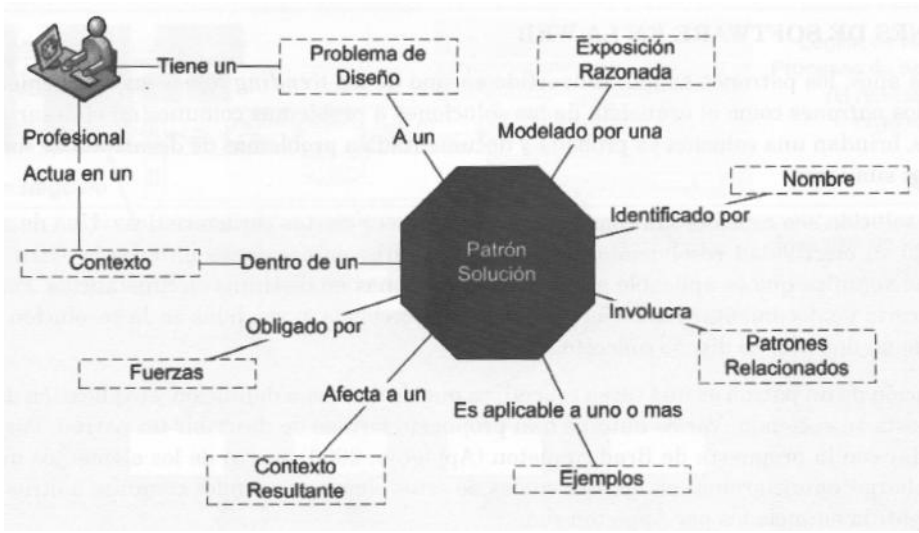
3.- PATRONES DE SOFTWARE EN LA WEB

En los últimos años, los patrones se han convertido en uno de los trending topics en la ingeniería del software. Podemos definir los patrones como el esqueleto de las soluciones a problemas comunes en el desarrollo de software. En otras palabras, brindan una solución ya probada y documentada a problemas de desarrollo de software que están sujetos a contextos similares.

Para que una solución sea considerada como patrón debe poseer ciertas características. Una de ellas es que debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser reutilizable, lo que significa que es aplicable a diferentes problemas en distintas circunstancias. Por ello, un patrón es una forma literaria de documentar las mejores prácticas y lecciones aprendidas en la resolución de un problema complejo dentro de un dominio de diseño concreto.

La documentación de un patrón es una tarea crucial, ya que de su buena definición y explicación dependerá de que la solución propuesta se entienda. Varios autores han propuesto formas de describir un patrón. Por su completitud, nos vamos a quedar con la propuesta de Brad Appleton (Appleton, 2000) acerca de los elementos que debería tener un patrón, sin embargo, enunciaremos en negrita cuales de estos elementos son los comunes a otras propuestas. Los elementos de un patrón enunciados por Appleton son:

- **Nombre.** Es un nombre descriptivo y único que ayuda a identificar y referenciar al patrón.
- **Problema.** Descripción resumida en una o dos frases que nos describe la intención del patrón, es decir, las metas y objetivos que queremos alcanzar.
- **Contexto.** Problema recurrente en el que es aplicable el patrón. Suelen usarse ejemplos del estado inicial del sistema antes de que el patrón sea aplicado.
- **Fuerzas.** Descripción de las fuerzas, los objetivos y restricciones relevantes para ese patrón, y de cómo éstas interaccionan entre ellas o con las metas que deseamos alcanzar. Además puede incluir un escenario concreto que sirva de motivación para el patrón. La noción de fuerza generaliza los tipos de criterios que justifican al patrón.
- **Solución.** Es el corazón del patrón. Está formado por un conjunto de instrucciones que describen cómo construir la solución del problema. Esta descripción puede ir acompañada de dibujos, diagramas o esquemas explicativos de dicho patrón.
- **Ejemplos.** Casos prácticos, pueden ser visuales, que ayudan al lector a entender el uso y la aplicabilidad del patrón.
- **Contexto resultante.** Indica el estado del sistema después de aplicar el patrón, incluyendo sus consecuencias (positivas y negativas).
- **Exposición razonada.** Expone cómo funciona el patrón y por qué es útil. Mientras que la solución muestra la estructura visible del patrón, la exposición explica sus mecanismos subyacentes.
- **Patrones relacionados.** Patrones que se pueden combinar con este, o es posible aplicar a partir del contexto resultante, o representan soluciones alternativas.



El éxito de los patrones ha generado que existan una gran cantidad de tipos. Entre las categorías tradicionales se encuentran:

- **Patrones de arquitectura.** Son aquellos que expresan un esquema organizativo o estructural fundamental para sistemas de software. Entre los patrones arquitectónicos se encuentran, MVC, las arquitecturas por niveles y las arquitecturas por capas descritas en los apartados anteriores.
- **Patrones de diseño.** Estos son los tipos de patrones más conocidos y son los que expresan esquemas para definir estructuras de diseño (o sus relaciones) con las que construir sistemas de software. Los patrones de diseño casi siempre resuelven problemas de diseño de código. A su vez, este tipo de patrones se divide en tres categorías:
    - **De Creación.** El objetivo de estos patrones es abstraer el proceso de instanciación y ocultar los detalles de cómo los objetos son creados o inicializados.
    - **Estructurales.** Los patrones estructurales describen como las clases y objetos pueden ser combinados para formar grandes estructuras y proporcionar nuevas funcionalidades. Estos objetos adicionados pueden ser incluso objetos simples u objetos compuestos.
    - **De Comportamiento.** Los patrones de comportamiento nos ayudan a definir la comunicación e iteración entre los objetos de un sistema. El propósito de este patrón es reducir el acoplamiento entre los objetos.
- **Dialectos.** Patrones de bajo nivel específicos para un lenguaje de programación o entorno concreto.

Además de los patrones ya vistos actualmente existen otros patrones como los siguientes:

- **Patrones de interacción.** Son patrones que nos permiten el diseño de interfaces web.
- **Patrones de análisis.** Describen un conjunto de prácticas destinadas a elaborar modelos de los conceptos principales de la aplicación que se va a construir (Fowler, 1996). Estos

patrones apoyan el trabajo de modelado, pues no siempre tienen experiencia al respecto y, en la mayoría de los casos, construyen sus modelos sin referencia alguna. Los patrones de análisis se diferencian de los de diseño en que se centran en aspectos sociales, de organización *y* económicos, los cuales son primordiales en el análisis de requisitos *y* la aceptación y usabilidad del sistema final.

- **Patrones de dominio.** Se caracterizan por dar soluciones a un dominio específico. Sirven como referencia conceptual del dominio del problema, ayudándonos a identificar cuáles son los requisitos o características más relevantes de ese dominio.

La siguiente tabla recopila patrones expuestos por diversos autores y que están relacionados con la construcción de la página web.

Navegación	
Nombre del patrón	**Descripción**
Index Navigation (Garzotto et al., 1999)	Proporcionar un acceso rápido a un conjunto de conceptos para que los usuarios que estén interesados en uno o varios de ellos puedan realizar su elección.
Guided Tour Navigation (Garzotto et al., 1999)	El usuario debe poder acceder de manera secuencial a un grupo de conceptos relacionados.
Organización	
Nombre del patrón	**Descripción**
Hierarchical Organization (van Duyne et al., 2002)	Organizar la información en una jerarquía de categorías puede ayudar al estudiante a encontrar las cosas (p. ej. siguiendo la estructura de un curso, capítulos, lecciones u organizarlo por contenidos).
Task-based Organization (van Duyne et al., 2002)	Completar un conjunto de tareas relacionadas de una manera rápida y sencilla enlazando dichas tareas según la secuencia en que se deben realizar.
PRESENTACION	
Nombre del patrón	**Descripción**
Navigation Bar (van Duyne et al., 2002)	El usuario debe encontrar siempre visible y de manera consistente las herramientas de ayuda a la navegación.

Define and Run Presentation (Cybulski y Linden, 1999)	El usuario debe percibir los elementos multimedia como una composición estética, donde un conjunto de elementos son mostrados de manera secuencial en uno o varios canales.
Personalización	
Nombre del patrón	**Descripción**
Structure Personalization (Rossi et al., 2001)	El usuario debe acceder solo a la información que le interesa.
Content Personalization (Rossi et al., 2001)	El usuario debe recibir los contenidos de manera personalizada.
Usabilidad	
Nombre del patrón	**Descripción**
Sentido de localización	El usuario necesita saber su localización dentro del sitio web, indicándole dónde está, en qué contexto y la ruta de información seguida para llegar a ese punto.
Volver a un sitio seguro	Los usuarios pueden sentirse desorientados o perdidos cuando han realizado más de una parada en el sitio web. Es necesario proporcionar un modo para que los usuarios pongan marcas para volver cuando se sientan perdidos.
Logotipo del sitio arriba a la izquierda	Los usuarios necesitan saber cómo volver a un sitio seguro, p. ej. a la página principal. Según el estándar, el logotipo se debe situar arriba a la izquierda de cada página y al pinchar sobre él, siempre te lleva a la página principal (homepage).
Botón de vuelta atrás	Los usuarios suelen cometer errores, especialmente al realizar tareas en varios pasos. El usuario debería ser siempre capaz de volver al paso anterior, y deshacer las cosas que haya podido hacer.
Interáccion	
Nombre del patrón	**Descripción**
Wizard	Para guiar al usuario en la realización de una tarea que necesita tomar varias decisiones, mostrarle cuáles son los pasos que existen y cuáles han sido realizados a la vez que se le guía a lo largo de la tarea.

Stepping	Permitir a los usuarios ir al paso anterior y siguiente de una tarea para realizar posibles modificaciones.
Input Error	Informar al usuario de entradas de datos incorrectas, dónde se han producido y cómo resolverlas.
Outgoing Links	Mostrar al usuario los enlaces que le llevarán fuera del actual sitio web marcándolos con un icono después de su etiqueta.
Seguridad	
Nombre del patrón	**Descripción**
Authorization	Describir los tipos de acceso a los recursos del sistema, distinguiendo entre entidades activas (roles de usuario o ejecución de un proceso) y entidades pasivas.
Role-Based Access Control	Asignar los tipos de acceso a los usuarios según sus roles en el sistema.
Multilevel Security	Tratar con diferentes niveles de seguridad.

En la Red podrás encontrar mucha información sobre la implementación de patrones en entornos de desarrollo para la creación de aplicaciones web. Aquí dejamos unos cuantos sitios web que pueden ser útiles de acuerdo a la plataforma o lo que busques para la Web.

Patrón	URL
Patrones de)2EE	http://Java.sun.com/blueprints/corej2eepatternsl Patterns/index. HTML
Patrones de ASP.Net	http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=20559)
Patrones web de Yahoo	http://developer.yahoo.com/ypatternsl
Patrones para Android	http://www.androídpatterns.com/
Catálogo de patrones para la Web	http://patterntap.com/
Ruby on rails	http://www.rubyonrails.org.es/
Patrones en PHP	http://www.ibm.com/developerworks/library/os-php-designptrns/

4.- FRAMEWORKS

Para la implementación de las arquitecturas por capas o niveles, pueden utilizarse entornos ya desarrollados que nos aportan elementos para la implementación de las mismas, los **frameworks**.

Así un framework web podría definirse como un conjunto de componentes software (clases, interfaces, Apis, etc) orientadas hacia el desarrollo de aplicaciones web usando patrones de diseño y modelos de arquitecturas en capas o niveles.

Los frameworks actuales suelen incluir:

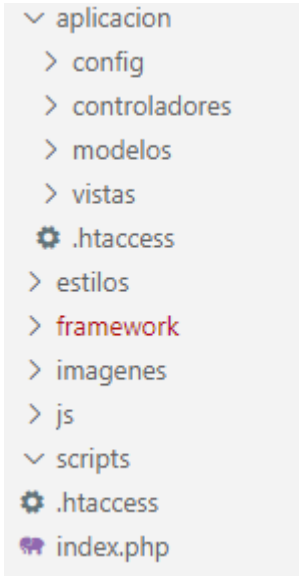
- ***Modelo-Vista-Controlador (MVC)***. Separa la lógica, del control de la misma y su visualización.
- ***Abstracción de URLs y sesiones***. No es necesario manipular directamente las URLs ni las sesiones, el framework ya se encarga de hacerlo.
- ***Acceso a datos***. Incluyen las herramientas e interfaces necesarias para integrarse con herramientas de acceso a datos, en BBDD, XML, etc.. Normalmente usan un mapeado Objeto-Relacional del tipo DataSet.
- ***Controladores específicos***. La mayoría de frameworks implementa una serie de controladores para gestionar eventos, como una introducción de datos mediante un formulario o el acceso a una página. Estos controladores suelen ser fácilmente adaptables a las necesidades de un proyecto concreto.
- ***Autentificación y control de acceso*** Incluyen mecanismos para la identificación de usuarios mediante login y password y permiten restringir el acceso a determinas páginas a determinados usuarios usando ACL (listas de control de acceso).
- ***Internacionalización***. Permiten definir la aplicación web en diferentes idiomas de forma fácil.

4.1.- FRAMEWORK PEDROSA

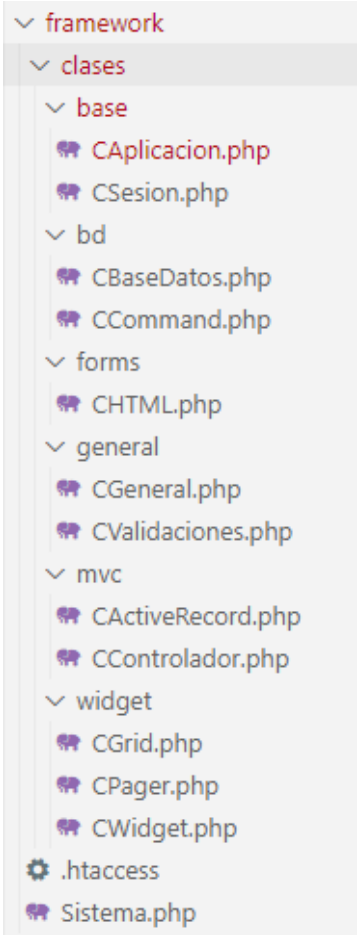
En este curso vamos a trabajar con un framework creado en cursos anteriores en clase y que se ampliará con diversa funcionalidad.

Este framework está basado en uno existente llamado Yii framework (http://www.yiiframework.com/).

Para crear nuestra aplicación usando este framework tendremos que copiar una serie de carpetas y archivos a nuestro proyecto. Estos archivos tendrán la siguiente estructura:



- Carpeta framework: Contendrá los ficheros del framework. Define las clases que dan funcionalidad general a nuestra aplicación. Normalmente no debe modificarse, salvo que se quiera incluir nueva funcionalidad al framework.
- Aplicación: Contendrá los ficheros propios de nuestra aplicación. Aquí definiremos los controladores, los modelos, las vistas, clases auxiliares, etc necesarios para implementar la lógica del programa deseada.
- Fichero index.php: Define el punto de acceso a la aplicación. Será el único fichero accesible mediante URL. En principio no debe cambiarse.
- Otras carpetas: Se pueden crear otras carpetas en las que añadir nuestro propio código. En ellas podremos introducir código javascript, hojas de estilos, imágenes, etc.



4.1.1.- LAS CLASES DEL FRAMEWORK

El framework incorpora una serie de clases. Estas clases gestionan el MVC, la aplicación, etc.

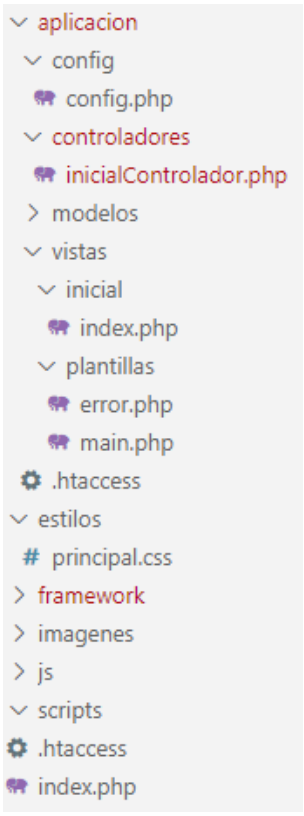
Las clases que nos encontramos en el framework son:

- Sistema.php: Esta clase define el objeto básico para acceder a la aplicación y a la funcionalidad del framework.
- CAplicacion: Es la clase que gestiona la aplicación. Incluye enlaces a los demás elementos y se encarga de gestionar el MVC.
- CHTML.php: Esta clase contiene una serie de métodos estáticos que nos permiten crear de forma más fácil las páginas html. Incorpora métodos para definir los formularios.
- CGeneral.php. Contiene algunos métodos estáticos usados por el resto del framework. Por ejemplo, métodos para convertir

fechas a formato mysql, métodos para escapar cadenas, etc.

- CValidaciones.php. Contiene métodos estáticos para validación de diferentes tipos de datos como fechas, email, números, etc
- CActiveRecord.php. Es la clase base usada para definir los modelos a usar en nuestra aplicación.
- CControlador.php. Es la clase base usada para definir los controladores a usar en nuestra aplicación.
- CBaseDatos.php y CCommand.php: Son las clases usadas para el acceso a bases de datos MySql.
- CWidget, CPager y CGrid: Clases que nos permiten dibujar elementos de interfaz.

4.1.2.- LA PRIMERA APLICACIÓN



Cuando nosotros creamos una aplicación lo primero es copiar toda la estructura del Framework en una carpeta. Con esto ya tenemos una aplicación totalmente operativa.

Posteriormente se irán definiendo los controladores, las vistas y los modelos que implementarán nuestra lógica de negocio y la interfaz de la aplicación.

Una vez copiado el framework tendremos la estructura que se ve al margen.

El archivo index.php contiene el código necesario para la puesta en marcha de la aplicación encargándose de procesar la petición realizada por el usuario.

El fichero config.php se encarga de indicar configuraciones generales para la aplicación. Inicialmente su contenido es:

```
<?php

    $config=array("CONTROLADOR"=> array("inicial"),
                  "RUTAS_INCLUDE"=>array("aplicacion/modelos"),
                  "URL_AMIGABLES"=>true,
                  "VARIABLES"=>array("autor"=>"Profesor",
                                     "direccion"=>"C/ Carrera -
Madre Carmen, 12"
                                    ),
                  "BD"=>array("hay"=>false,
                              "servidor"=>"localhost",
                              "usuario"=>"root",
                              "contra"=>"2daw",
                               "basedatos"=>"practica10")
```

```
                    );
```

La configuración se indica mediante un array con varias posiciones asociativas:

- CONTROLADOR: indica cual es el controlador/acción que se carga inicialmente. En nuestro caso es el controlador inicial.
- RUTAS_INCLUDE: indica las rutas en las que colocaremos clases que se cargarán de forma automática en la aplicación (con la autocarga de clases que lleva implementado el framework). Por defecto viene definida con la carpeta aplicación/modelos donde se colocarán los modelos que necesitemos.
- URL_AMIGABLES: indica si las url que se generan son normales o de tipo amigables
- VARIABLES: permite definir variables accesibles desde cualquier punto de la aplicación. En este caso define la propiedad autor. Se podrá acceder mediante Sistema::app()->autor
- BD: Define opciones para la creación automática de un objeto CBaseDatos en la aplicación.

Como se ha visto, el controlador por defecto es inicial. Todos los controladores se sitúan en la carpeta /aplicación/controladores. Un controlador no es más que una clase con una serie de métodos que implementan las acciones a realizar por la aplicación. El controlador *inicial* se corresponderá con el fichero *inicialControlador.php*. Como se ve el controlador xxxx se almacenaría en un fichero con nombre xxxxControlador.php. Si no se hace así el framework no será capaz de acceder al controlador.

El controlador inicial (inicialControlador.php) quedaría de esta forma

```
<?php
class inicialControlador extends CControlador
{
    public function accionIndex()
    {
        $this->barraUbi = [
            [
                "texto" => "Inicio",
                "enlace" => ["inicial"]
            ],
        ];

        $this->menuizq = [
            [
                "texto" => "Inicio",
                "enlace" => ["inicial"]
            ]
        ];
```

```
        $this->dibujaVista("index",[],
                           "Pagina principal");
    }


}
```

Se tiene la clase inicialControlador (igual que el fichero) que extiende de la clase base CControlador.

Las acciones se definen como métodos públicos de esta clase. Igual que con el controlador se sigue una convención para indicar las acciones: la acción yyyy debe definirse en un método llamado accionYyyy.

Todos los controladores tienen la acción por defecto index, que se llamará si se omite la acción al llamar al controlador.

Por último, y para hacer la prueba, he definido un sitio virtual llamado framework. Ahora, ya puedo acceder al sitio desde un navegador.

4.1.3.- LOS CONTROLADORES

Ya se ha visto que los controladores no son más que clases que heredan de CControlador, que se ubican en la carpeta /aplicación/controladores y que tienen una convención para el nombre tanto para la propia clase como para las acciones.

Así el controlador *artículos* se crea en un archivo llamado *articulosControlador.php* en la carpeta /aplicacion/controladores. La clase se debe llamar *articulosControlador*.

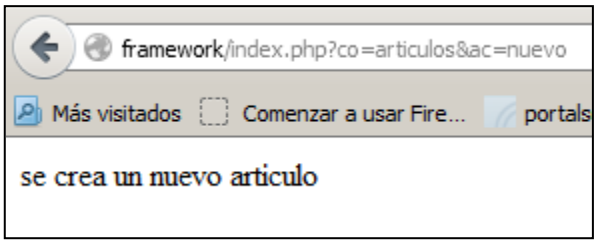
```
<?php

class articulosControlador extends CControlador
{
    public function accionIndex()
    {
        echo "accion por defecto de articulos";
    }

}
```

Cada operación que se desea gestionar se corresponde con una acción. Podríamos pensar que una acción se corresponde con una página html. Esto es, si yo quisiera hacer una página para añadir un nuevo artículo sin usar el framework definiría un fichero nuevoArticulo.php. En el caso del framework tengo que crear una acción en el controlador de artículos que se llame nuevo.

```
public function accionNuevo()
     {
          echo "se crea un nuevo articulo";
     }
```

Pedrosa es un framework con un único punto de acceso /index.php. Todas las peticiones se gestionan a través de este fichero. Esto permite centralizar aspectos como la seguridad y limitar los posibles problemas.

Una vez creado la acción/controlador, se podría llamar con la url apropiada: ***http://framework/index.php?co=articulos&ac=nuevo***



En esta uri se tienen dos parámetros:

- co: controlador
- ac: acción para el controlador

Si se omite la acción, se llama a la acción por defecto del controlador. Si se omite el controlador, se llama al controlador por defecto.

Actualmente se tiende a definir **url amigables**. Esto permite que los buscadores pueden almacenar mejor las referencias a las páginas dinámicas. En una url amigable se sustituyen los parámetros por "carpetas" en la dirección. Si hemos definido la opción URL_AMIGABLES a true, el framework generará las URL como amigables. En este caso se tendría

```
url normal
www.sitio.es/index.php?co=controlador&ac=accion&par1=valor1

url amigable equivalente
www.sitio.es/controlador/accion/par1=valor1
```

Si no se indica la acción o el controlador, ese elemento no aparecerá en la url correspondiente.

A nivel interno, se define la dirección como un array con dos posibles posiciones: el controlador y la acción. Si se omite el segundo, se consideraría la acción por defecto del controlador. Si se omiten los dos, se considera la acción por defecto del controlador por defecto

```php
$direccion=array("usuarios","ver");
//indica accion ver del controlador usuarios.

$direccion=array("usuarios");
//indica acción por defecto del controlador usuarios

$direccion=array();
//indica acción por defecto del controlador por defecto
```

CAplicacion aporta una serie de métodos que nos permiten trabajar con direcciones.

Se accede al objeto CAplicación usando la clase SISTEMA. Esta clase tiene el método estático app() que nos devuelve una referencia al objeto aplicación.

```php
Sistema::app()
```

El método **generaURL** (generaURL($direccion, $parametros)) devuelve una cadena que representa la URL equivalente a la dirección con los parámetros indicados. Como dirección se puede indicar tanto un array en el formato ya visto como una cadena que representa directamente la url (http://www.ejemplo.es)

```php
Sistema::app()->generaURL(array("usuario"));
// devolvería la cadena "/index.php?co=usuario
Sistema::app()->generaURL(array("usuario","modificar"),
```

```
                                array("id"=>21));
// devolvería la cadena "/index.php?co=usuario&ac=modificar&id=21
```

El valor devuelto por la función generaURL se puede usar para indicar en nuestra aplicación las direcciones. No se aconseja escribir ninguna dirección relativa a la aplicación indicando directamente la url.

```
// referencia a direcciones de la propia aplicación
<a href="<?php
      echo Sistema::app()->generaURL(array("usuario","modificar"),
                                array("id"=>21));?>">
                Modificar usuario</a>
<a href="<?php
     echo Sistema::app()->generaURL(array("usuario","ver"),
                                array("id"=>21));?>">
                Ver usuario</a>

// referencia a direcciones externas a la aplicacion
<a href="http://www.google.es">Acceso a google</a>
```

El método **irAPagina**($direccion, $parametros) permite ir a la página indicada por la dirección. Es equivalente a poner en la barra de direcciones la dirección correspondiente o a ejecutar "location: dirección". Como dirección se puede indicar tanto un array en el formato ya visto como una cadena que representa directamente la url (http://www.ejemplo.es)

```
//abriría la página
//index.php?co=usuarios&ac=ver&id=1
Sistema::app()->irAPagina(array("usuarios","ver"),
    array("id"=>1));
```

Por último y aunque no usa directamente un controlador, la clase CAplicacion incorpora el método **paginaError**($numero,$mensaje). Este método permite mostrar una página de error generando el código de error "número" y devolviendo éste al cliente. Usa para ello la plantilla error.

```
Sistema::app()->paginaError(404,"pagina no encontrada");
```

Cuando se crea el controlador es posible indicar la acción por defecto mediante la propiedad **accionDefecto**. El siguiente código muestra lo comentado anteriormente.

```
class articulosControlador extends CControlador
{
    public function __construct()
    {
        $this->accionDefecto="verTodos";
    }

    public function accionVerTodos()
```

```
    {
        echo "accion por defecto de articulos";
        ?>
     <a href="<?php echo
     Sistema::app()->generaURL(array("articulos","nuevo"));?>">
           Nuevo articulo
     </a><br />
     <b>LISTADO DE ARTICULOS</b>


     <?php
     }

     public function accionNuevo()
     {
           echo "se crea un nuevo articulo";
           //tras crear el articulo volvemos al listado
           Sistema::app()->irAPagina(array("articulos"));
     }
}
```

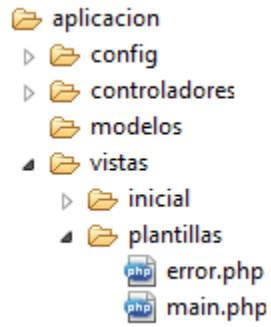
4.1.4.- LAS VISTAS

La interfaz de la aplicación puede generarse directamente en la acción del controlador. Esto acarrea problemas:

- Como definimos la interfaz de la aplicación. ¿De una forma común? ¿una a una en cada acción?
- Si queremos cambiar la interfaz una vez creada, hay que modificar todas las acciones que la usan.
- Se mezcla la lógica de negocio con la interfaz.

Para resolver los problemas anteriores se usan las vistas. Una vista no es más que una forma de definir cómo se va a ver la página.

En el framework, una página se creará a partir de dos elementos: la plantilla (layout) y la vista.



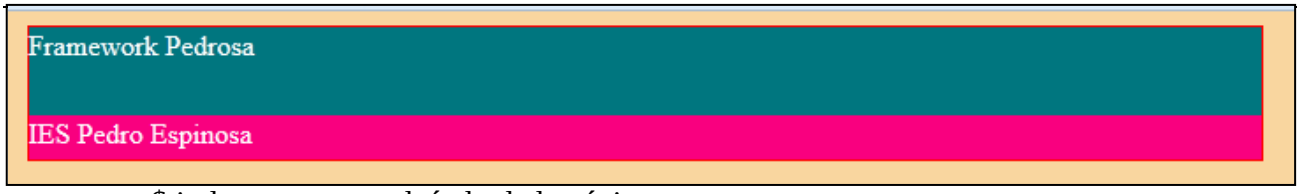
Una plantilla permite fijar un formato común de página para el sitio. En esta plantilla se definirá la cabecera, el pie, la zona donde irá el contenido, etc.

Si nos fijamos en la estructura de carpetas de la aplicación, ya aparecen dos plantillas dentro de la carpeta /aplicación/vistas/plantillas: error.php y main.php. La que nos interesa es main.php.

```
<!DOCTYPE html>
<html>
	<head>
		<title><?php echo $titulo;?></title>
		<!-- definiciones comunes a todo el sitio -->
		<link	type="text/css"	href="/estilos/principal.css"
rel="stylesheet"/>
	</head>
	<body>
		<div id="todo">
			<div id="cabecera">Framework Pedrosa
			</div>
			<div id="conte">
				<?php echo $contenido; ?>
			</div>
			<div id="pie">IES Pedro Espinosa</div>
		</div>
	</body>
</html>
```

El fichero main.php contiene código html que define como se mostrará la página. Como en cualquier página, podemos incluir enlaces a Javascript, enlaces a CSS, apartados propios, etc. En particular, en este fichero hay un enlace a estilo.

Además aparecen dos variables php **$titulo** y **$contenido**.



- $titulo se usa para el título de la página.
- $contenido se rellena de forma automática con la salida generada por la vista que hayamos llamado.

La plantilla se vería así

Cada controlador tiene una plantilla asignada que puede cambiarse mediante la propiedad **$plantilla**

```
class articulosControlador extends CControlador
{
     public function __construct()
     {
          $this->accionDefecto="verTodos";
          $this->plantilla="main";
     }
```

Una vista se crea como un fichero en una ubicación especial. Si tenemos un controlador, las vistas correspondientes a dicho controlador deben situarse en la carpeta /aplicación/vistas/controlador. Por ejemplo, tengo el controlador articulos y quiero crear la vista nueva, tendré que crear el fichero nueva.php en la carpeta /aplicación/vistas/articulos.

El siguiente paso, consiste en usar la vista. La clase CControlador incorpora dos métodos para ello:

- dibujaVistaParcial**(**$vista, $variables=array**()**,$devolver=false). Que mostraría la $vista sin incluir la plantilla.
- dibujaVista**(**$vista, $variables=array**()**,$titulo="aplicacion"). Que mostraría la vista incluyendo la plantilla. El tercer parámetro es el título que se asignará a la barra de títulos del navegador (variable $titulo en la plantilla).

Siguiendo con el ejemplo, podría crear la vista listar (/aplicación/vistas/artículos/listar.php)

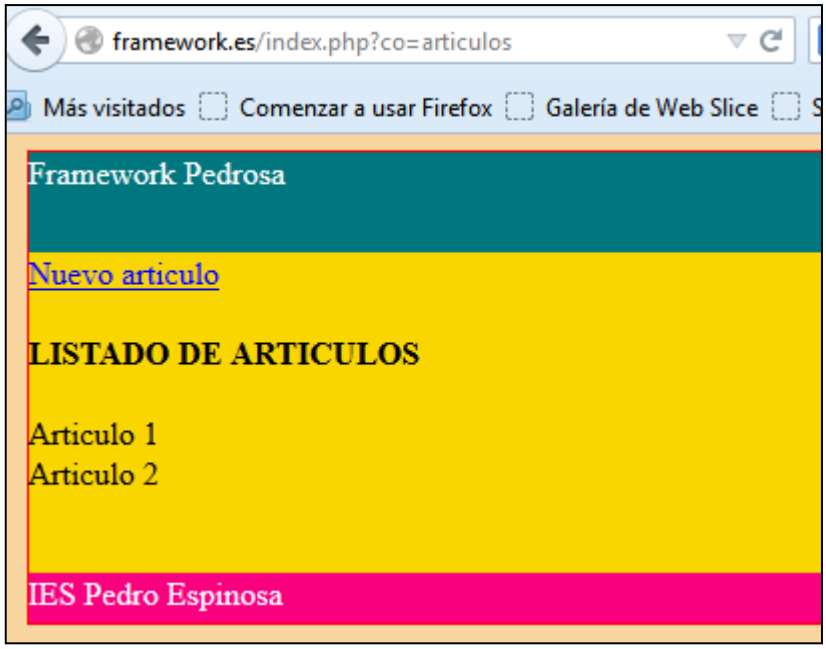
```
<a        href="<?php        echo        Sistema::app()->generaURL(
array("articulos","nuevo"));?>">
     Nuevo articulo
</a><br />
<br />

<b>LISTADO DE ARTICULOS</b>

<br /> <br />
```

```
Articulo 1 <br />
Articulo 2 <br />
<br />
<br />
```

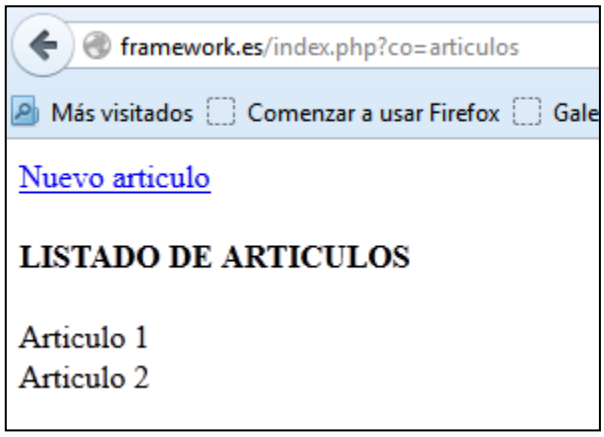
Para que se cargara la vista anterior en la acción verTodos haría:

```
public function accionVerTodos()
{
    $this->dibujaVista("listar");
    return;

}
```

El resultado sería



En este caso se dibuja la vista y la plantilla (método dibujaVista). Si yo quisiera sólo dibujar la vista sin la plantilla usaría el método dibujaVistaParcial. En este caso el resultado sería:



Otro punto a tener en cuenta con las vistas es la posibilidad de pasarle parámetros que se podrán usar dentro de la vista. Esto se consigue con el segundo parámetro de los métodos indicados. Como segundo parámetro se debe indicar un array en el que cada elemento es de la forma "nombreVariable"=> valor. Posteriormente se usa esa variable dentro de la vista como $nombreVariable.

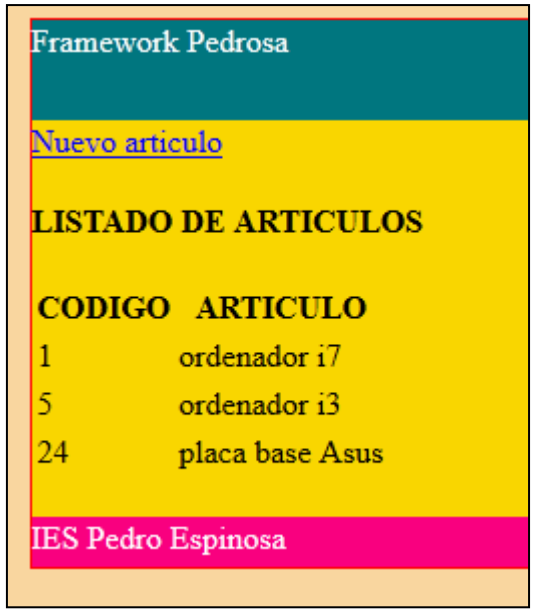
Por ejemplo puedo modificar la acción del controlador para definir una variable

```
public function accionVerTodos()
{

    $articulos=array(1=>"ordenador i7",
        5=>"ordenador i3",
        24=>"placa base Asus");

    //dibujo la vista pasándole un parámetro
    $this->dibujaVista("listar", array("arti"=>$articulos));
    return;
}
```

Dentro de la vista usaré la variable $arti (es el nombre asignado)

```
<a href="<?php echo Sistema::app()->
                    generaURL(array("articulos","nuevo"));?>">
     Nuevo articulo
</a><br />
<br />

<b>LISTADO DE ARTICULOS</b>

<br /> <br />
<table>
     <tr><th>CODIGO</th><th>ARTICULO</th></tr>
<?php
    foreach($arti as $clave=>$valor)
     {
          echo "<tr><td>$clave</td><td>$valor</td></tr>\n";
     }
     ?>
</table>
<br />
```

El resultado sería



Las vistas parciales son útiles cuando queremos dividir nuestra interfaz en partes pudiendo asignar cada parte a una vista. En el caso anterior podríamos tener la vista listar y para mostrar cada artículo una segunda vista (articulo) que muestra los datos del articulo de una forma determinada.

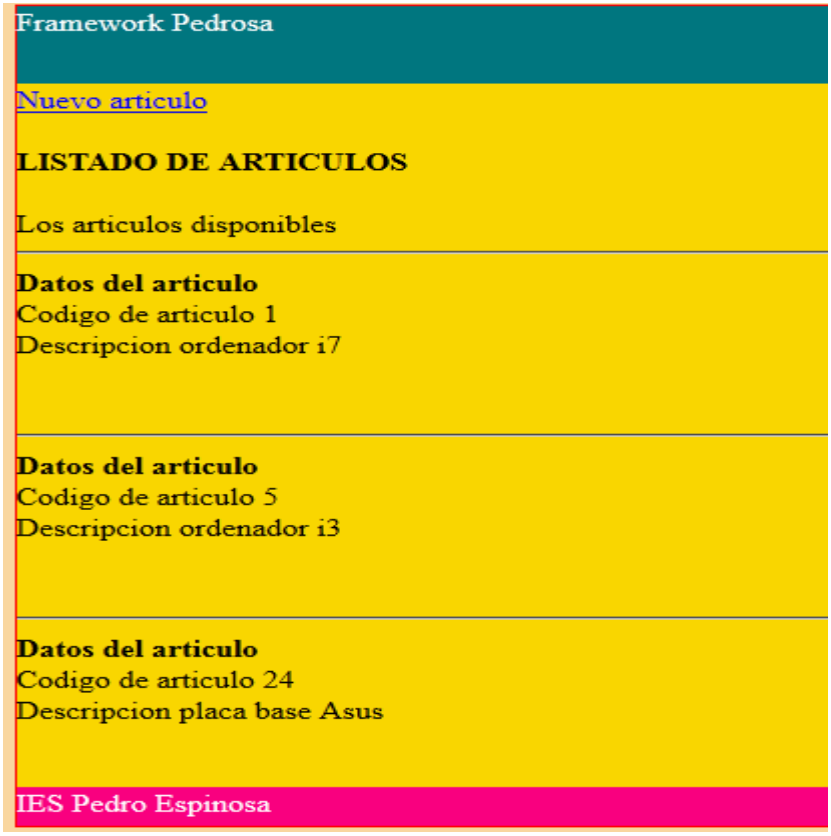
Como ejemplo vamos a crear la vista datosArticulo

```
<br />
<hr />
<b>Datos del articulo</b><br />
Codigo de articulo <?php echo $art["cod"];?> <br />
Descripcion <?php echo $art["nombre"];?> <br />
<br />
```

La vista listar quedaría

```
<b>LISTADO DE ARTICULOS</b>

<br /> <br />
Los articulos disponibles
<?php
    foreach($arti as $clave=>$valor)
     {
          $this->dibujaVistaParcial("datosArticulo",
                    array("art"=>array("cod"=>$clave,
                                       "nombre"=>$valor)));
     }
     ?>
</table>
```

Y el resultado sería:



Es importante hacer notar que la vista se carga internamente dentro de un método de la clase controladora desde la que se ha llamado. Es por eso que $this hace referencia a la clase controladora y por tanto todo lo que tengamos definida en ella (propiedades y métodos) es accesible en la vista.

Así en el caso anterior, dentro de la vista listar he ejecutado ***$this->dibujaVistaParcial("datosArticulo",…)*** En este caso $this hace referencia a la clase controladora Articulos que es la encargada de llamar a mostrar vista.

Dentro de la plantilla podremos usar cualquier variable que queramos siempre que se rellene antes de cargarla (normalmente se definiría en la vista). Esta variable debe ser definida como $this->var. Por ejemplo, en la plantilla puedo usar la variable $this->var. En todas las vistas que usan esa plantilla debo asignar un valor a dicha variable o tener en cuenta la no existencia (isset($this->var)

De igual forma es posible usar las propiedades dinámicas definidas en la aplicación. Éstas se indicaban en el fichero config.php, opción “VARIABLES”. Una vez establecidas se puede acceder a las mismas mediante Sistema::app()->miVar

Cuando se genera una vista se captura la salida estándar de la vista y se almacena en la variable $contenido. Esto se consigue a través de una api especial “Funciones del control de salida”. Esta librería guarda la salida estándar (lo que enviamos mediante echo o directamente el código html) en un buffer interno. Posteriormente se puede guardar en una variable. Las funciones más útiles serían:

- ob_start(): inicia la captura de la salida.
- ob_get_contents(): devuelve la salida capturada como una cadena de caracteres.
- ob_end_clean: borra la salida capturada hasta el momento y finaliza el proceso de

captura.

4.1.5.- LOS MODELOS

Un modelo representa un objeto del mundo real que queremos controlar en nuestro programa. Podría equipararse en cierto modo a un registro de una tabla de BD o a una clase que encapsula un objeto del mundo real.

Fundamentalmente un modelo representa un elemento para el que se conocen sus atributos con sus restricciones.

Por ejemplo, pensemos en un artículo del mundo real que queremos modelar en nuestra aplicación. Un articulo tiene nombre, fabricante, un precio, una fecha de fabricación, etc. Además sabemos una serie de restricciones como que el fabricante es una cadena de 50 caracteres o que la fecha de fabricación es posterior al año 1950. En nuestra aplicación crearemos un modelo que representa un articulo con atributos nombre, fabricante, precio, etc y restricciones las indicadas anteriormente. Este modelo nos permitirá crear nuevos artículos, modificar los datos de artículos existentes, etc.

Un modelo aporta mecanismos para la definición de los atributos, las restricciones y para la gestión del objeto. Además, al estar normalmente los datos almacenados en BD, dan soporte directo para la inserción, modificación, borrado y obtención de esa información.

Un modelo se crea extendiendo la clase base CActiveRecord. Estos modelos se sitúan en la carpeta aplicacion/modelos.

Vamos a poner un ejemplo. Quiero controlar artículos. Cada artículo tiene un código, un nombre, una descripción, un fabricante y una fecha de alta. Además sé que el código tiene que ser entero, la descripción una cadena de 60 caracteres, el nombre una cadena de 30 caracteres, el código de fabricante un entero mayor o igual a 0 y la fecha de alta del año 2000 o posterior.

Con todo lo anterior, creo el modelo artículos en el archivo artículos.php en la carpeta /aplicación/modelos con el siguiente contenido:

```
class Articulos extends CActiveRecord
{
    protected function fijarNombre()
    {
        return 'arti';
    }

    protected function fijarAtributos()
    {
        return array("cod_articulo","descripcion",
            "nombre","cod_fabricante",
            "fabricante_nombre","fecha_alta");
    }

    protected function fijarDescripciones()
    {
        return array("fecha_alta"=>"Fecha de alta",
```

```
            "cod_fabricante"=>"Fabricante",
            "nombre_fabricante"=>"Fabricante");
    }

    protected function fijarRestricciones()
    {
        Return
        array(array("ATRI"=>"cod_articulo,nombre",
            "TIPO"=>"REQUERIDO"),
            array("ATRI"=>"cod_articulo","TIPO"=>"ENTERO",
                "MIN"=>0),
            array("ATRI"=>"nombre","TIPO"=>"CADENA",
                "TAMANIO"=>30),
            array("ATRI"=>"descripcion",
                "TIPO"=>"CADENA", "TAMANIO"=>60),
            array("ATRI"=>"cod_fabricante","TIPO"=>"ENTERO",
                "MIN"=>0),
            array("ATRI"=>"fecha_alta", "TIPO"=>"FECHA"),
            array("ATRI"=>"fecha_alta",
                "TIPO"=>"FUNCION",
                "FUNCION"=>"validaFechaAlta"),
        );
    }

    protected function afterCreate()
    {
        $this->cod_articulo=0;
        $this->nombre="";
        $this->descripcion="";
        $this->cod_fabricante=1;
        $this->fabricante_nombre="SIN INDICAR";
    }

    public function validaFechaAlta()
    {
        $fecha1=DateTime::createFromFormat('d/m/Y',
            $this->fecha_alta);
        $fecha2=DateTime::createFromFormat('d/m/Y',
            '01/01/2000');
        if ($fecha1<$fecha2)
        {
            $this->setError("fecha_alta",
                "La fecha de alta debe ser posterior a
01/01/2000");
        }
    }

    public static function listaFabricantes($fabricante=null)
    {
        $fabricantes=array(1=>"Siemens",
            2=>"Intel",
            3=>"Apple");

        if ($fabricante===null)
            return $fabricantes;
            else
            {
                if (isset($fabricantes[$fabricante]))
                    return $fabricantes[$fabricante];
```

```
                    else
                        return false;
                }
            }

}
```

Lo primero que se observa es que el modelo hereda de CActiveRecord. Esta clase es la que realmente gestiona el modelo. El modelo en sí sólo define los elementos propios como los atributos, las restricciones, métodos particulares, etc.

Siguiendo con el ejemplo, lo primero es redefinir el método **fijarNombre**. Este método devuelve el nombre que se asignará al modelo. Este nombre es usado dentro del framework en otros componentes (en la clase CHTML cuando se dibujan los elementos del formulario).

Después se definen los atributos del modelo mediante **fijarAtributos**. Los atributos en un modelo se implementan internamente mediante propiedades creadas dinámicamente (__get, __set) de lo que se encarga CActiveRecord. En este método se devuelve un array con la lista de atributos que tendrá el modelo.

A los atributos se les asigna una descripción de forma automática que coincide con el nombre del campo. Si deseamos definirle una descripción propia se usa el método **fijarDescripciones** devolviendo un array asociativo con cada elemento de la forma “atributo” => “descripción”.

Una vez indicados los atributos, el siguiente paso es indicar las restricciones sobre los mismos. Estas restricciones incluyen desde la definición de los tipos de datos a la comprobación de los valores. Para indicar las restricciones se redefine el método **fijarRestricciones**.

El método fijarRestricciones devuelve un array con las restricciones a aplicar en el modelo para uno o varios campos. Un campo puede tener diferentes restricciones. Las restricciones podrán ser

- REQUERIDO: El campo al que se refiere no puede evaluarse a false, es decir, si es cadena tiene que ser distinto de ‘’, si es entero distinto de 0, si es array distinto de array vacio, etc
- CADENA: El campo al que se refiere debe ser de tipo cadena y con una longitud máxima determinada.
- ENTERO: El campo al que se refiere debe estar entre unos valores mínimos y máximos.
- REAL: El campo al que se refiere debe estar entre unos valores mínimos y máximos.
- FECHA: el campo al que se refiere debe ser una fecha válida.
- HORA: el campo al que se refiere debe ser una hora válida.
- RANGO: el campo al que se refiere debe tener un valor igual a alguno de los definidos.
- EMAIL: El campo al que se refiere debe ser una dirección de correo válida.
- FUNCION: Para validar el campo se ejecuta la función que se indica.

Para cada restricción se creará un array con los siguientes elementos

- ATRI: nombre del campo o campos a los que se aplica la restricción. Si hay varios se indican separados por comas.
- TIPO: Cadena que representa el tipo de restricción. Los posibles valores son REQUERIDO, CADENA, ENTERO, REAL, FECHA, HORA, RANGO, FUNCION (se aplicará una función definida por el usuario), EMAIL. Por defecto CADENA
- TAMANIO: Longitud máxima del campo. Usado con restricciones de tipo CADENA. Por defecto 30.
- MIN: valor mínimo (solo para restricciones de tipo ENTERO y REAL). Por defecto -100000.
- MAX: valor máximo (solo para restricciones de tipo ENTERO y REAL). Por defecto 100000.
- DEFECTO: valor por defecto (para restricciones de tipo ENTERO, REAL, RANGO, CADENA, FECHA, HORA y EMAIL). Si se omite se asignan los siguientes valores: para ENTERO y REAL 0, para CADENA "", para RANGO el primer valor del RANGO, para FECha "01/01/2000", para HORA "00:00:00" y para EMAIL "aaa@aa.com"
- RANGO: Array con los posibles valores del atributo (solo para restricciones de tipo RANGO).
- MENSAJE: Mensaje que se asignará en caso de que al validar la restricción no sea correcta. Cada tipo tiene su mensaje por defecto.
- FUNCION: Cadena que representa la función personalizada que permite validar el campo. Por defecto ".

Siguiendo con el ejemplo se puede ver como en el método se definen 7 restricciones. La primera obliga a que cod_articulo y nombre no estén vacíos (restricción requerido). La segunda restricción hace que el cod_articulo sea entero con valor mínimo de 0 (restricción entero). La tercera y cuarta definen campos como cadena indicando el tamaño máximo (restricción cadena). La restricción FECHA hace que el campo fecha_alta deba ser una fecha válida. La última restricción es de tipo FUNCION. Esta restricción obliga a que se defina la función validaFechaAlta.

La clase CActiveRecord define una serie de métodos adicionales como **setError**($atributo, $mensaje). Este método asigna el error $mensaje al atributo $atributo. Este método lo hemos usado dentro de la función validaFechaAlta para indicar que la validación ha fallado. Este método es público por lo que puede usarse para indicar errores obtenidos fuera de la clase.

Para realizar las validaciones más usuales se usan unos métodos estáticos definidos en la clase CValidaciones (/framework/clases/general). Nos encontramos en esta clase con validaDNI (para validar dni), validaEntero, validaReal, validaCodPostal, validaEmail, validaLista, validaFecha y validaHora,

Otro método que se redefine es **afterCreate**. Este método se usa para asignar a los atributos del modelo los valores por defecto que deseemos.

Otros métodos interesantes de CActiveRecord son:

- Método getNombre(). Devuelve el nombre asignado al modelo.
- Método errorAtributo($atributo): Devuelve el/los mensajes de error asignados actualmente

al atributo o null en caso de que en la última validación no hubiese error sobre el atributo.

- Método getErrores(). Devuelve los errores que hay. Cada error es de la forma "descripción_campo: mensaje_de_error".
- Método setValores($arrayValores=array()). Este método permite asignar al modelo los valores indicados en el array. Se asignarán aquellos valores del array cuyo índice asociativo coincida con un campo del modelo.
- Método publico validar(). Se encarga de validar todas las restricciones definidas. Devuelve verdadero si cumple todas las restricciones y falso en cualquier otro caso. Siempre se validan todas las restricciones.

Una vez que ya tenemos el modelo, el siguiente paso es usarlo en combinación con las vistas y con los controladores.

En el controlador crearía la acción nuevo en la que usaría el modelo.

```
public function accionNuevo()
{
    //creo un nuevo objeto articulo
    $articulo=new Articulos();

    //nombre del modelo= nombre del array por post
    $nombre=$articulo->getNombre();
    if (isset($_POST[$nombre]))
    {
        //asigno un codigo de articulo por defecto
        $articulo->cod_articulo=5;
        //asigno  los valores al articulo a partir de lo recogido
del formulario
        $articulo->setValores($_POST[$nombre]);

        //compruebo si son validos los datos del articulo
        if ($articulo->validar())
        {  //son validos los datos del articulo

            //almaceno el articulo en la base de datos


            //redirecciono a la página de listado de todos los
articulos
            Sistema::app()->irAPagina(
                array("articulos",
                    "verTodos"));
                exit;
        }
        else
        {  //no es valido, vuelvo a mostrar los valores
            $this->dibujaVista("nuevo",
                array("modelo"=>$articulo),
                "Crear articulo");
                exit;
        }
    }
```

```
    //muestro la vista inicialmente
    $this->dibujaVista("nuevo",
        array("modelo"=>$articulo),
        "Crear articulo");

}
```

La vista nuevo sería

```
<H1>ALTA DE UN ARTICULO</H1>
<br />

<?php
     echo CHTML::iniciarForm();

     echo CHTML::modeloLabel($modelo, "nombre");
     echo CHTML::modeloText($modelo,
                    "nombre",array("maxlength"=>30,"size"=>31));
     echo CHTML::modeloError($modelo, "nombre");
     echo "<br>";
     echo CHTML::modeloLabel($modelo, "descripcion");
     echo CHTML::modeloText($modelo,
                "descripcion", array("maxlength"=>60,"size"=>61));
     echo CHTML::modeloError($modelo, "descripcion");
     echo "<br>";
     echo CHTML::modeloLabel($modelo, "cod_fabricante");
     echo CHTML::modeloListaDropDown($modelo,
                           "cod_fabricante",
                           Articulos::listaFabricantes(),
                           array("linea"=>false));
     echo CHTML::modeloError($modelo, "cod_fabricante");
     echo "<br>";
     echo CHTML::modeloLabel($modelo, "fecha_alta");
     echo CHTML::modeloText($modelo,
                           "fecha_alta",
                           array("maxlength"=>10,"size"=>11));
     echo CHTML::modeloError($modelo, "fecha_alta");
     echo "<br>";

     echo CHTML::campoBotonSubmit("Crear");

     echo CHTML::finalizarForm();
```

El resultado sería:

Para generar los elementos del formulario se pueden indicar directamente las etiquetas o usar la clase CHTML que aporta un conjunto de funciones para esto.

4.1.6.- SOPORTE DE BASE DE DATOS

El framework Pedrosa incorpora soporte para base de datos a través de los componentes CBaseDatos/CCommand. Podemos tanto crear nuestro propio enlace a una base de datos como utilizar el que crea por defecto al arrancar la aplicación.

En este caso hay que tener en cuenta que habrá que configurar la aplicación para indicar los parámetros de acceso a la base de datos modificando el fichero **config.php**

```
$config=array("CONTROLADOR"=> array("inicial"),
    "RUTAS_INCLUDE"=>array("aplicacion/modelos"),
    "BD"=>array("hay"=>true,
        "servidor"=>"localhost",
        "usuario"=>"root",
        "contra"=>"root",
        "basedatos"=>"relacion6"));
```

Como se ve se define la posición “BD” que será un array con valores:

- hay: booleano que indica si se quiere o no soporte de bd automático.
- servidor: nombre o dirección ip del servidor.
- usuario: nombre de usuario con el que conectaremos a la BD.
- contra: contraseña usada para conectarse a la BD.
- basedatos: nombre de la base de datos a usar.

Una vez configurado se podrá acceder a la conexión a la BD mediante:

```
Sistema::app()->BD();
```

De esta forma podríamos ejecutar cualquier sentencia SQL de la forma:

```
$sentencia="select * from articulos";

//ejecuto la sentencia
$consulta=Sistema::App()->BD()->crearConsulta($sentencia);

//devuelvo las filas
$filas=$consulta->filas();
```

De igual forma es posible crear nuestro propio enlace a la base de datos usando directamente la clase CBaseDatos.

```
     //creo el enlace a la base de datos
$enlace=new CBaseDatos("localhost","root","root","relacion6");
$sentencia="select * from articulos";

//ejecuto la sentencia
$consulta=$enlace->crearConsulta($sentencia);

//devuelvo las filas
```

```
$filas=$consulta->filas();
```

Otro elemento interesante y muy relacionado con las Bases de Datos son los modelos. Un modelo representa un objeto con su comportamiento. Normalmente debe guardarse de forma permanente por lo general en una base de datos. Debido a que esto es muy común, los modelos traen incorporado soporte para base de datos. (nota es posible trabajar con un modelo sin que se obtenga/guarde la información en BD).

Si queremos trabajar en un modelo con bases de datos se debe modificar el modelo. Por ejemplo si tuvieramos el modelo Articulos nos quedaría:

```
class Articulos extends CActiveRecord
{
    protected function fijarNombre()
    {
        return 'arti';
    }

    protected function fijarTabla()
    {
        return "articulos";
    }

    protected function fijarId()
    {
        return "codArticulo";
    }
```

El método fijarTabla establece el nombre de la tabla en la BD relacionada con el modelo. Si no se indica ninguna tabla no habrá soporte de BD.

El método fijarId establece el campo que se corresponde con la clave principal de la tabla, en este caso codArticulo.

Para que el modelo y la tabla de BD mantengan la misma información hay que asegurarse que los campos del modelo se llamen igual a los de la tabla. En el modelo podremos tener más campos que los de la tabla.

Una vez establecidos esos elementos en el modelo podemos usar una serie de métodos para acceder a la información. Lo normal es crear una instancia del modelo. En este caso se considera que el elemento es nuevo. Si lo que queremos es cargar la información de un determinado elemento se usa el método **buscarPorId($valor,$opciones=array())**. Este método devuelve true si encuentra un elemento en la tabla cuyo id corresponda con el valor indicado. Además si lo encuentra, carga los valores del registro en el modelo.

```
//creo una instancia del modelo articulos
```

```
$articulo=new Articulos();

//busco un registro en la tabla cuyo id sea 1
if (!$articulo->buscarPorId(1))
    echo "articulo no encontrado";
```

Otro método usado para cargar el modelo a partir de la base de datos sería **buscarPor($opciones=array())**, al que se le indica las opciones de búsqueda en la base de datos. Igual que buscarPorId, devolvería true si encuentra un registro y carga los datos del mismo en el modelo.

```
if (!$articulo->buscarPor(array("where"=>"nombre='ORDENADOR'")))
{
    echo "No se ha encontrado el articulo";
}
```

Una vez que tenemos el modelo cargado a partir de la base de datos podremos trabajar con él igual que se ha hecho en apartados anteriores: validar, asignar a propiedades, etc.

Cuando cargamos el modelo desde la base de datos se pueden indicar operaciones a realizar con los datos cargados. Esto se consigue con el método **afterBuscar** que se debe redefinir en el modelo. Un uso muy común es transformar la fecha en formato MySQL 'YYYY-MM-DD' al formato que usamos normalmente 'DD/MM/YYYY'.

```
protected function afterBuscar() {
    $fecha= $this->fecha_alta;
    $fecha= CGeneral::fechaMysqlANormal($fecha);
    $this -> fecha_alta = $fecha;
}
```

Algunas veces nos interesa que al buscar un elemento en la base de datos, se recojan además de los campos de la tabla otros que se obtienen de otro tabla (al estilo de una vista que devuelve datos de varias tablas). Esto se puede hacer al redefinir el método buscarPorId()

```
public function buscarPorId($valor,$opciones=array())
{
    return parent::buscarPorId($valor,
        array("select"=>
            "t.*,".
            "t.nombre as nombre_articulo"));
}
```

Otra operación normal, es la de guardar los datos del modelo en la base de datos. Esto se consigue llamando al método **guardar()** que devuelve true si se realiza correctamente. Tras guardar

los datos se vuelve a cargar el registro desde la base de datos.

Con todo lo indicado la acción para articulo nuevo podría ser

```
public function accionNuevo()
{
    //creo un nuevo objeto articulo
    $articulo=new Articulos();

    //nombre del modelo= nombre del array por post
    $nombre=$articulo->getNombre();
    if (isset($_POST[$nombre]))
    {
        //asigno  los valores al articulo a partir de lo
        //recogido del formulario
        $articulo->setValores($_POST[$nombre]);

        //compruebo si son valido los datos del articulo
        if ($articulo->validar())
        {  //son validos los datos del articulo

            //almaceno el articulo en la base de datos
            if (!$articulo->guardar())
            {
                $this->dibujaVista("nuevo",
                    array("modelo"=>$articulo),
                    "Crear articulo");
                    exit;
            }

            //redirecciono a la página de listado de todos
            //los articulos
            Sistema::app()->irAPagina(array("articulos",
                "verTodos"));
            exit;
        }
        else
        {  //no es valido, vuelvo a mostrar los valores
            $this->dibujaVista("nuevo",
                array("modelo"=>$articulo),
                "Crear articulo");
                exit;
        }
    }

    //muestro la vista inicalmente
    $this->dibujaVista("nuevo",array("modelo"=>$articulo),
        "Crear articulo");

}
```

Para que el método de guardar se realice correctamente se debe haber indicado en el modelo las sentencias de inserción y actualización mediante la redefinición de los métodos **fijarSentenciaInsert()** y **fijarSentenciaUpdate()**. Para el ejemplo de artículos podríamos haber

hecho:

```
protected function fijarSentenciaInsert()
{
    $nombre=CGeneral::addSlashes($this->nombre);
    $descripcion=CGeneral::addSlashes($this->descripcion);
    $fabricante=CGeneral::addSlashes($this->fabricante);
    $fecha=CGeneral::fechaNormalAMysql($this->fechaAlta);

    return "insert into articulos (".
        "   nombre, descripcion, fabricante, fechaAlta ".
        "         )   values  (  ".
        "   '$nombre', '$descripcion, ".
        "   '$fabricante', '$fecha'".
        "           ) " ;
}

protected function fijarSentenciaUpdate()
{
    $nombre=CGeneral::addSlashes($this->nombre);
    $descripcion=CGeneral::addSlashes($this->descripcion);
    $fabricante=CGeneral::addSlashes($this->fabricante);
    $fecha=CGeneral::fechaNormalAMysql($this->fechaAlta);

    return "update articulos set ".
        "     nombre='$nombre', ".
        "     descripcion='$descripcion', ".
        "     fabricante='$fabricante', ".
        "     fechaAlta='$fecha'".
        "    where codArticulo={$this->codArticulo} ";
}
```

Hay que indicar que en la sentencia insert se considera que la clave principal es de tipo autonumérica y que se obtiene de forma automática.

Además de lo indicado anteriormente es muy normal el obtener filas de la tabla. Para ello hay dos métodos: **buscarTodosNRegistros($opciones)** y **buscarTodos($opciones).** El primero nos devolvería un entero con el numero de filas para la consulta con las $opciones indicadas mientras el segundo nos devolvería las filas de la consulta.

```
$numero=$articulo->buscarTodosNRegistros();
$filas=$articulo->buscarTodos();
```

A estos métodos se le pueden pasar un array con las cláusulas para la sentencia select que se genera. El array asociativo tendrá las posiciones asociativas select, from, where, group, having, order y limit.

```
$filas=$articulo->buscarTodos(
    array("select"=>"   t.codArticulo,   t.nombre,   count(*)   as
n_filas",
        "from"=>" inner join fabricantes f".
```

```
        " on (f.cod_fabricante=t.cod_fabricante)",
        "group"=>"1,2",
        "having"=>"count(*)>1",
        "order"=>"t.nombre asc",
        "where"=>"nombre like '%h%",
        "limit"=>"10,5"));
```

Cualquiera de las posiciones puede omitirse quedando la misma con valores por defecto (select, from) o simplemente no aplicarse (group, having, order, where, limit).

De forma automática, siempre se incorpora la tabla en el from con el alias t. Por ejemplo, si la tabla se llama artículos la clausula from quedaría de la forma “from artículos t”. En la posición from se puede indicar como se enlaza la tabla con cualquier otra que queramos.

Por último es posible ejecutar cualquier sentencia sql mediante el método ejecutarSentencia($sentencia).

```
$sentencia="select current_date";
$articulo->ejecutarSentencia($sentencia);
```

4.1.7.- LA INTERFAZ : LA CLASE CHTMLY LOS WIDGETS

En HTML todos los elementos se escriben como etiquetas (<a>, <input>,
, etc). Dentro de la clase CHTML encontramos los métodos básicos para escribir las etiquetas y los elementos más comunes. A estos métodos se pasa como parámetro un array (atributosHTML) que contiene valores a convertir en atributos para la etiqueta.

El array opciones será de la forma "atributo_html"=>valor. Por ejemplo, para el siguiente array:

```
$atributos=array("name"=>"ele1",  "id"=>"id_ele1",  "maxlength"=>30,
     "disabled"=>true,  "type"=>"text");
```

Tendríamos la siguiente cadena html:

```
type="text" name="ele1" id="id_ele1" maxlength="30"
disabled="disabled"
```

Algunos de los atributos son especiales, es decir, el valor es igual a la etiqueta como por ejemplo disabled que debería escribirse como disabled="disabled". Los atributos especiales son: async, autofocus, autoplay, checked, controls, declare, default, defer, disabled, formnovalidate, hidden, ismap, loop, multiple, muted, nohref, noresize, novalidate, open, readonly, required, reversed, scoped, seamless, selected, typemasmatch.

Los elementos de formulario como checkbox, text, etc se pueden usar directamente para un campo determinado de un modelo. En este caso se usa la información correspondiente del modelo para determinar atributos html entre los que están "name" e "id". Esto permite posteriormente generar envíos de formularios en los que se recoja los valores como arrays.

Por ejemplo, dado el modelo Usuarios y los campos nombre, dirección, edad. Para trabajar correctamente al realizar la recogida del formulario, los inputs para los distintos campos tendrán respectivamente los siguientes id y name:

- Campo nombre del modelo Usuarios: atributo name -> Usuarios[nombre], atributo id -> usuarios_name.
- Dirección: name -> Usuarios[direccion], id ->usuarios_direccion
- Edad: name -> Usuarios[edad], id -> usuarios_edad

En esta clase nos encontramos con:

- Propiedad estática errorCSS. Contiene el nombre de la clase que se asignará a la etiqueta si se indica como propiedad.

- Propiedad estática cerrarEtiquetasUnicas. Por defecto vale true e indica si las etiquetas únicas como br o hr se deben cerrar (
) o no (
).

- Propiedad estática prefijoID. Cadena que representa el prefijo para los id autogenerados.

- Método **generaID**(): Devuelve una etiqueta de ID única en la página en la que estamos.

- Método estático **dibujaEtiqueta**($etiqueta, $atributosHTML=array(), $contenido=null, $cerrarEtiqueta=true). Este método devuelve una cadena html correspondiente a la etiqueta indicada con las opciones. Si se indica un contenido se agrega y si además cerrarEtiqueta vale true se pone una etiqueta de cierre. Si no se tiene contenido se considera que la etiqueta es única. Por ejemplo:

```
CHTML::dibujaEtiqueta("br");
//devuelve <br > o <br /> si cerrarEtiquetasUnicas vale true

CHTML::dibujaEtiqueta("input",array("id"=>"nombre","type"=>"text",
"readonly"=>1));
//devuelve <input id="nombre" type="text" readonly="readonly" >
//o <input .... /> si cerrarEtiquetasUnicas vale true

CHTML::dibujaEtiqueta("textarea",    array("cols"=>80),"datos    de
usuario");
//devuelve <textarea cols="80">datos de usuario</textarea>

CHTML::dibujaEtiqueta("textarea",    array("cols"=>80),"datos    de
usuario",false);
//devuelve <textarea cols="80">datos de usuario
//sin la etiqueta de cierre
```

  Este método es el básico para dibujar cualquier etiqueta como input, link, img, etc

- Método estático **dibujaEtiquetaCierre**($etiqueta). Este método devuelve una cadena html correspondiente a la etiqueta de cierre correspondiente.

```
CHTML::dibujarEtiquetaCierre("textarea");
//devuelve </textarea>
```

- Método estático **boton**($etiqueta, atributosHTML=array()). Genera un botón con la etiqueta indicada y atributos indicados (etiqueta input).

- Método estático **css**($texto, $medio=””). Genera la etiqueta apropiada para encerrar el texto css indicado para el medio correspondiente.

- Método estático **cssFichero**($url, $medio=””). Genera la etiqueta apropiada para cargar el fichero css indicado por la url para el medio correspondiente.

- Método estático **botonHtml**($etiqueta, atributosHTML=array()). Genera un botón html (etiqueta button) para los valores indicados.

- Método estático **imagen**($src, $alt, atributosHTML=array()). Genera una etiqueta de imagen para los valores datos (etiqueta img).

- Método estático **link**($texto, $url=”#”, $atributosHTML=array()). Este método devuelve una cadena correspondiente a una etiqueta link para la url indicada y los atributos html correspondientes.

- Método estático **linkHead**($rel=null, $type=null, $href=null, $media=null, atributosHTML=array()). Genera una etiqueta link para insertarla en la sección HEAD de la página. Los valores rel, type, href y media se añadirán como atributos html siempre que no sean null.

- Método estático **metaHead**($content, $name=null, $httpEquiv=null, atributosHTML=array()) Genera una etiqueta meta para insertarla en la sección HEAD. Los valores name y httpEquiv se añadirán a los atributos html solo si son distintos de null.

- Método estático **normalizaURL**($url). Este método devuelve una cadena normalizada para la url indicada. Se usará cuando necesitemos una url correcta. Según el valor de $url se devolverá:
  Si vale “”, se devuelve la dirección actual (#).
  Si es una cadena no vacía se devuelve tal cual.
  Si es un array, se considera de la forma (controlador, acción) y se devuelve el resultado de llamar al método generaURL de CAplicacion.

- Método estático **script**($texto, atributosHTML=array()). Genera una etiqueta script para el $texto javascript y los atributos indicados.

- Método estático **scriptFichero**($url, atributosHTML=array()). Genera una etiqueta script para la url y los atributos indicados.

Un elemento fundamental al crear las aplicaciones web es el formulario. Un formulario permitirá la recogida de valores que normalmente se guardarán en modelos. En CHTML se tienen dos conjuntos de funciones para generar elementos de formulario. El primero es genérico mientras que el segundo se orienta hacia un modelo determinado.

Métodos para dibujar formularios/objetos de formulario genéricos.

- Método protegido **dameIdDeNombre**($name). Devuelve una cadena que representa un ID obtenido a partir de $name. Se sustituyen [,], , por _.
- Método protegido **campoInput**($tipo,$nombre,$valor,$atributosHTML=array()). Este método devuelve una cadena que corresponde con una etiqueta input del tipo indicado y con los atributos correspondientes, generándose un id apropiado. Si en $atributosHTML se indica “id”=>false, no se genera ningún id para el input.
- Métodos estáticos **campoDate**, **campoEmail**, **campoFile**, **campoHidden, campoNumber, campoPassword, campoRange, campoText, campoTextarea, campoTime, campoUrl**. Todos reciben como parámetro ($nombre,$valor,$atributosHTML=array()) y se encargan de devolver una cadena correspondiente al input del tipo (date, email, file, hidden, ….) y con los valores indicados

- Método estático **campoCheckBox**($nombre,$checked=false,$atributosHTML). Devuelve una cadena correspondiente a un input de tipo checkbox con los valores indicados. Dentro de atributosHTML se pueden indicar valores especiales
    - value: valor que se enviará si se marca el checkbox.
    - uncheckValor: valor que se corresponde con lo que se enviaría en caso de que no se marcara la casilla. Se consigue añadiendo un campo oculto con su valor. Si vale false no se añade.
    - Etiqueta: Permite añadir un label al final para indicar el texto del checkbox.
- Método estático **campoListaCheckBox**($nombre, $seleccionado, $datos, $separador="
\n", $atributosHTML=array()). Devuelve una cadena correspondiente a una serie de checkbox, tantos como elementos haya en el array $datos, separados por $separador. Aparecerán seleccionados aquellos cuyo valor (el índice del array $datos) sea igual a $seleccionado (puede ser un solo valor –una cadena- o una serie de valores –un array-). Dentro de atributosHTML puede indicarse como valores especiales:
    - uncheckValor: valor que se corresponde con lo que se enviaría en caso de que no se marcara la casilla. Se consigue añadiendo un campo oculto con su valor. Si vale false no se añade.
- Método estático **campoRadioButton**($nombre, $checked=false, $atributosHTML). Devuelve una cadena correspondiente a un input de tipo radio con los valores indicados. Dentro de atributosHTML se pueden indicar valores especiales
    - value: valor que se enviará si se marca el checkbox.
    - uncheckValor: valor que se corresponde con lo que se enviaría en caso de que no se marcara la casilla. Se consigue añadiendo un campo oculto con su valor. Si vale false no se añade.
    - Etiqueta: Permite añadir un label al final para indicar el texto del checkbox.
- Método estático **campoListaRadioButton**($nombre, $seleccionado, $datos, $separador="
\n", $atributosHTML=array()). Devuelve una cadena correspondiente a una serie de radio, tantos como elementos haya en el array $datos, separados por $separador. Aparecerá seleccionado aquel cuyo valor (el índice del array $datos) sea igual a $seleccionado (una cadena). Dentro de atributosHTML puede indicarse como valor especial:
    - uncheckValor: valor que se corresponde con lo que se enviaría en caso de que no se marcara la casilla. Se consigue añadiendo un campo oculto con su valor. Si vale false no se añade.
- Método estático **campoListaDropDown**($nombre, $seleccionado, $datos, $atributosHTML=array()). Devuelve una cadena correspondiente a un select, con tantas opciones como elementos haya en el array $datos. Aparecerá seleccionado aquel cuyo valor (el índice del array $datos) sea igual a $seleccionado (una cadena). Dentro de atributosHTML puede indicarse como valor especial:
    - Linea: Permite añadir una opción inicial para valor por defecto o texto indicativo.

- Método estático **iniciarForm**($accion, $metodo =”post”, atributosHTML=array()). Genera la etiqueta de apertura de un formulario para la acción, el método y los atributos indicados.
- Método estático **finalizarForm**(). Devuelve una cadena que representa la etiqueta de cierre de form.

Para dibujar objetos de formularios a partir de un modelo se tiene los siguientes métodos.

- Método estático **modeloError**($modelo, $atributo, $atributosHTML=array()). Permite mostrar una caja (div) en la que aparecerán los errores correspondientes al atributo del modelo.

- Método estático **modeloErrorSumari**o($modelo, $atributosHTML=array()). Permite mostrar una caja (div) con todos los errores del modelo.

- Métodos estáticos **modeloDate, modeloEmail, modeloFile, modeloHidden, modeloNumber, modeloPassword, modeloRange, modeloText, modeloTextarea, modeloTime, modeloUrl**. Todos reciben como parámetro ($modelo, $atributor, $atributosHTML=array()) y se encargan de devolver una cadena con el input del tipo correspondiente (date, email, file, hidden, ….) para el campo atributo del modelo.
- Método estático **modeloCheckBox**($modelo, $atributo, $atributosHTML). Devuelve una cadena con un input de tipo checkbox para el campo $atributo del modelo $modelo. Dentro de atributosHTML se pueden indicar valores especiales
    - value: valor que se enviará si se marca el checkbox.
    - uncheckValor: valor que se corresponde con lo que se enviaría en caso de que no se marcara la casilla. Se consigue añadiendo un campo oculto con su valor. Si vale false no se añade.
    - Etiqueta: Permite añadir una label al final para indicar el texto del checkbox.
- Método estático **modeloListaCheckBox**($modelo, $atributo, $datos, $separador=”
\n”, $atributosHTML=array()). Devuelve una cadena correspondiente a una serie de checkbox, tantos como elementos haya en el array $datos, separados por $separador. Aparecerán seleccionados aquellos cuyo valor (el índice del array $datos) sea igual al valor del campo $atributo del modelo $modelo. Dentro de atributosHTML se pueden indicar valores especiales:
    - uncheckValor: valor que se corresponde con lo que se enviaría en caso de que no se marcara la casilla. Se consigue añadiendo un campo oculto con su valor. Si vale false no se añade.
- Método estático **modeloRadioButto**n($modelo, $atributo, $atributosHTML). Devuelve una cadena con un input de tipo radio para el campo $atributo del modelo $modelo. Dentro de atributosHTML se pueden indicar valores especiales

  - value: valor que se enviará si se marca el checkbox.
  - uncheckValor: valor que se corresponde con lo que se enviaría en caso de que no se marcara la casilla. Se consigue añadiendo un campo oculto con su valor. Si vale false no se añade.
  - Etiqueta: Permite añadir un label al final para indicar el texto del checkbox.

- Método estático **modeloListaRadioButton**($modelo, $atributo, $datos, $separador="
\n",$atributosHTML=array()). Devuelve una cadena correspondiente a una serie de radio, tantos como elementos haya en el array $datos, separados por $separador. Aparecerán seleccionado aquellos cuyo valor (el índice del array $datos) sea igual al valor del campo $atributo del modelo $modelo. Dentro de atributosHTML puede indicarse como valor especial:
  - uncheckValor: valor que se corresponde con lo que se enviaría en caso de que no se marcara la casilla. Se consigue añadiendo un campo oculto con su valor. Si vale false no se añade.
- Método estático **modeloListaDropDown**($modelo, $atributo, $datos, $atributosHTML=array()). Devuelve una cadena correspondiente a un select con tantas options como elementos haya en el array $datos. Aparecerá seleccionado aquel cuyo valor (el índice del array $datos) sea igual al valor del campo $atributo del modelo $modelo. Dentro de atributosHTML puede aparecer:
  - Linea: Permite añadir una opción inicial para valor por defecto o texto indicativo.

Al crear la interfaz de la aplicación se suele usar de forma muy común elementos como tablas, cajas, menus, etc. Los widgets son implementaciones de estos elementos que nos permiten mejorar la presentación de la aplicación y nos ahorran mucho trabajo de desarrollo.

En el framework Pedrosa, todos los widgets heredan de la clase CWidget. Esta clase tiene los siguientes métodos:

- dibujaApertura(). Devolvería una cadena que se corresponde con todo el código html necesario para dibujar la apertura del widget. (al estilo de un form seria la parte de iniciar el form <form….>)
- dibujaFin(). Devuelve una cadena con la parte del código html correspondiente al cierre del widget. (al estilo de un form sería la parte de </form>
- dibujate(). Devuelve una cadena con todo el código html para el widget. Es decir, es equivalente a llamar a dibujaApertura y dibujaFin consecutivamente.
- requisitos(). Método estático que devuelve una cadena con el código html necesario para que funcione el widget. Podría ser código javascript, estilo, etc que se sitúa en la parte del head.

Para generar la cadena se suele usar la API OB_ que nos permite capturar la salida estándar y almacenarla en una cadena.

El funcionamiento de los widget es simple. Si para que el widget funcione correctamente se requiere código adicional como javascript o css, en primer lugar se envía al navegador la cadena

que devuelve el método estático requisitos. Posteriormente se llama al constructor del widget y con la instancia generada, se llama al método apropiado para que se dibuje. Normalmente todas estas operaciones se realizan dentro de una vista.

A continuación muestro un código de ejemplo usando la clase CWidget. Es solo un ejemplo de funcionamiento ya que esta clase no puede instanciarse al ser abstracta:

```
//pongo los requisitos si fueran necesarios
echo CWidget::requisitos();

……

CHTML::dibujaEtiqueta("br");

//creo la instancia del widget
$objeto=new CWidget();


//dibujo el widget
echo $objeto->dibujate();


//si entre la apertura y el cierre
//puedo indicar código html

//abro el código html del widget
echo $objeto->dibujaApertura();

//código html
echo "esto va dentro ";

//cierro el código html del widget
echo $objeto->dibujaFin();
```

Dentro del framework se tienen implementados dos widgets: ***CGrid y CPager***.

CGrid permite dibujar la tabla html correspondiente a un array de filas.

Siguiendo con el ejemplo de los artículos podría dibujar una tabla usando el componente CGrid. Para ello:

En la acción index del controlador pondría el siguiente código:

```
public function accionIndex()
{
    $articulo=new Articulos();

    //establezco las opciones de filtrado
    $opciones=array();

    //filas a dibujar en el CGrid
    $filas=$articulo->buscarTodos($opciones);
    foreach ($filas as $clave => $valor)
    {
        $filas[$clave]["fecha_alta"]=
```

```
            CGeneral::fechaMysqlANormal(
                $filas[$clave]["fecha_alta"]);
            //botones


$cadena=CHTML::link(CHTML::imagen("/imagenes/24x24/ver.png"),
                Sistema::app()->generaURL(
                    array("articulos","consultar"),
                    array("id"=>$filas[$clave]
                        ["cod_articulo"])));
            $cadena.=CHTML::link(CHTML::imagen(
                '/imagenes/24x24/modificar.png'),
                Sistema::app()->generaURL(
                    array("articulos","modificar"),
                    array("id"=>$filas[$clave]
                        ["cod_articulo"])));
            $cadena.=CHTML::link(CHTML::imagen(
                '/imagenes/24x24/borrar.png'),
                Sistema::app()->generaURL(
                    array("articulos","borrar"),
                    array("id"=>$filas[$clave]
                        ["cod_articulo"])),
                array("onclick"=>"return
confirm('&iquest;Esta seguro de borrar el articulo?');"));

                $filas[$clave]["opciones"]=$cadena;
        }

        //definiciones de las cabeceras de las
        //columnas para el CGrid
        $cabecera=array(array("ETIQUETA"=>"NOMBRE",
            "CAMPO"=>"nombre"),
            array("CAMPO"=>"descripcion"),
            array("CAMPO"=>"fabricante",
                "ETIQUETA"=>"fabricante"),
            array("CAMPO"=>"fecha_alta",
                "ETIQUETA"=>"fecha alta",
                "ANCHO"=>"200px",
                "ALINEA"=>"cen"),
            array("CAMPO"=>"opciones",
                "ETIQUETA"=>" operaciones")
        );

        //llamo a la vista con las definiciones para
        //el CGrid
        $this->dibujaVista("index",
            array("filas"=>$filas,
                "cabe"=>$cabecera),
            "lista de articulos");
}
```

Como se observa se crea un array con las filas donde cada posición asociativa define una columna y un array con las cabeceras. En este array se define las características que va a tener cada columna. Estas características se establecen mediante un array asociativo con las siguientes posiciones:

- "CAMPO": Posición asociativa en el array de filas ($filas) que representa el contenido de la columna. (obligatoria)
- "ETIQUETA": descripción que se mostrara en la columna. Si no se indica, se establecerá el valor de “CAMPO”.
- "ANCHO": ancho para la columna. Por defecto ‘’.
- "VISIBLE": la columna es visible o no. Por defecto visible.
- "ALINEA": alineacion del texto. Valores izq, der, cen. Por defecto izq

Una vez que ya tenemos la definiciones apropiadas para las filas y cabeceras, se dibuja el componente en la vista:

```
<?php
     //se crea el objeto CWidget con las definiciones
     //apropiadas
     $tabla=new CGrid($cabe,$filas,
                           array("class"=>"tabla1"));

     //se dibuja la tabla
     echo $tabla->dibujate()


?>
```

Se crea el objeto CGrid y se llama al método dibújate. Por defecto se dibuja la tabla usando estilos CSS enlazados a la clase tabla. En este ejemplo le he pasado como atributo HTML la clase tabla1. Esto me permite definir tablas con diferentes colores creando las reglas CSS apropiadas.

La hoja de estilos sería de la forma:

```
table.tabla1{
     background-color: #b6c7cd;
     border-collapse: collapse;
}

table.tabla1 th{
     background-color: blue;
     color:white;
}

table.tabla1 tr.par{
     background-color: #edd8d8;
     color:black;
}
table.tabla1 tr.impar{
     background-color: #f8f0d6;
     color:black;
}

table.tabla1 tr td{
     padding: 2px;
}
```

La tabla se vería

El widget CPager es un componente que dibuja un paginador. Se podría agregar un paginador al ejemplo anterior.

Primero definimos la cabecera en la accion del controlador:

```
//opciones del paginador
$opcPaginador= array("URL" => Sistema::app()->
    generaURL(array("articulos","index")),
    "TOTAL_REGISTROS" => 799,
    "PAGINA_ACTUAL" => 5,
    "REGISTROS_PAGINA" => 30,
    "TAMANIOS_PAGINA"=>array(5=>"5",
        10=>"10",
        20=>"20",
        30=>"30",
        40=>"40",
        50=>"50"),
    "MOSTRAR_TAMANIOS"=>true,
    "PAGINAS_MOSTRADAS"=>7,
);

//llamo a la vista con las definiciones para
//el CGrid y el CPager
$this->dibujaVista("index",
    array("filas"=>$filas,
        "cabe"=>$cabecera,
        "opcPag"=>$opcPaginador),
    "lista de articulos");
```

La vista index quedaría

```
//creo el paginador
$pagi=new CPager($opcPag,array());


//se crea el objeto CWidget con las definiciones
```

```
//apropiadas
$tabla=new CGrid($cabe,$filas,
    array("class"=>"tabla1"));


//dibujo el paginador
echo $pagi->dibujate();

//se dibuja la tabla
echo $tabla->dibujate();

//dibujo el paginador
echo $pagi->dibujate();
```

Para que se vea correctamente se deben incluir los siguientes estilos correspondientes al paginador:

```
.pager{
     background-color: white;
     color:black;
}

.pager .izq{
     float: left;
     //width: 40%;
     padding-left: 5px;
     color: green;
}

.pager .der{
     float: right;
     //width: 58%;
     text-align: right;
     padding-right: 5px;'
}

.pager .der a{
     text-decoration: none;
     color:green;
}
.pager .der a:hover{
     text-decoration: none;
     color:red;
}

.pager .final{
     clear: both;
}
```

El resultado sería:

El elemento más importante para crear el CPager son las opciones.

Las opciones es un array asociativo con las siguientes posiciones:

- URL: cadena con la url que se usa como base para los enlaces
- TOTAL_REGISTROS: número total de registros que se tendrán. Por defecto 0
- PAGINA_ACTUAL: pagina actual que se mostraría. Por defecto 1.
- REGISTROS_PAGINA: numero de registros por página. Por defecto 10.
- TAMANOS_PAGINA: array en el que aparecen los posibles tamaños de pagina. Cada tamaño posible será de la forma valor=>etiqueta. Por defecto array(10=>”10”,20=>”20”,30=>”30”,40=>”40”).
- MOSTRAR_TAMANOS: booleano que indica si se muestra el combo con los tamaños posibles o no. Por defecto true
- PAGINAS_MOSTRADAS: Numero de páginas que se muestran como enlaces. Por defecto 5.

Lo importante en este caso es rellenar correctamente los parámetros. Hay varios que se calculan a partir de la consulta que se quiere mostrar.

Por ejemplo si yo tengo una consulta que me devolvería en total 205 registros, si yo defino un tamaño de pagina de 10 registros por página, en total habría 21 páginas (205 / 10). La primera página incluiría registros del 1 al 10, la segunda 11 al 20, …. la página 21 del 201 al 205.

Otro punto a destacar es que el paginador define una serie de botones con los que poder ir a otra página. Cuando se pulsan se llama a la URL definida indicando la página deseada y el número de registros por página. Por ejemplo si pulso en el botón correspondiente a la página 6 se llamaría al navegador con la url

http://www.framework.es/index.php?co=articulos&ac=index&reg_pag=30&pag=6

Los parámetros son reg_pag (registros por página) y pag (página a mostrar). Es función del programador hacer que los registros para la tabla se correspondan con estos parámetros.

4.1.8.- OTROS ELEMENTOS DEL FRAMEWORK.

SISTEMA

La clase **Sistema** es la clase que permite ejecutar la aplicación y hacer que la misma funcione correctamente. Es una clase con las propiedades y métodos estáticos necesarios para el lanzamiento de la aplicación. Entre otras cosas se encarga de registrar un método para la autocarga de clases. De esta forma no es necesario indicar require e include para una serie de elementos. La autocarga de clases la realiza usando:

- Atributo de clase $_clasesBase. Contiene un enlace por cada clase del framework. Esto no debe cambiarse salvo que creemos nuestras propias clases para el framework, añadiendo enlaces a las mismas.
- Rutas: Se pueden indicar una o varias rutas en las que buscaría de forma automática los ficheros que contienen las clases que necesitemos en nuestra aplicación.

Para la carga de rutas se usa un fichero de configuración definido en la aplicación. Además es posible indicar rutas de búsqueda mediante el método de clase **nuevaRuta**

Por ejemplo podríamos tener:

```
//indicaría como ruta para la búsqueda de clases
// /aplicacion/misclases/ dentro de la raiz del sitio web
Sistema::nuevaRuta($_SERVER["DOCUMENT_ROOT"].
"/aplicacion/misclases/");
```

Sistema además nos da acceso a la aplicación mediante el método app().

```
//devuelve el objeto aplicación
Sistema::app();
```

CAplicacion

Esta clase define una serie de propiedades y métodos básicos con los que trabajar en la aplicación. Parte de los elementos ya se han visto anteriormente como los métodos generaURL, irAPagina o paginaError.

Esta clase contiene enlaces privados a otros elementos que se necesiten para trabajar con nuestra aplicación. Ya veremos como a lo largo de las prácticas, se irán creando elementos para acceso a la sesión, la ACL, la BD, etc.

El fichero index.php

Este fichero es el único accesible en nuestra aplicación y no debería modificarse.

Su contenido es

```
<?php
```

```php
//ruta al fichero de Sistema base
$pedrosa=dirname(__FILE__).'/framework/Sistema.php';

//ruta al fichero de configuración
$configuracion=dirname(__FILE__).'/aplicacion/config/config.php';

//incluye los ficheros de sistema y de configuracion
require_once($pedrosa);
require_once($configuracion);


//crea la aplicación y la ejecuta
Sistema::crearAplicacion($config)->ejecutar();
```

El fichero config.php (/aplicacion/config/config.php**).**

El fichero config.php se encarga de indicar configuraciones generales para la aplicación. Inicialmente su contenido es:

```php
<?php

    $config=array("CONTROLADOR"=> array("inicial"),
                  "RUTAS_INCLUDE"=>array("aplicacion/modelos"));
```

La configuración se indica mediante un array con varias posiciones asociativas:

- CONTROLADOR: indica cual es el controlador/acción que se carga inicialmente. En nuestro caso es el controlador inicial.
- RUTAS_INCLUDE: indica las rutas en las que colocaremos clases que se cargarán de forma automática en la aplicación (con la autocarga de clases que lleva implementado el framework). Por defecto viene definida con la carpeta aplicación/modelos donde se colocarán los modelos que necesitemos.

En este fichero se irán añadiendo las configuraciones necesarias para el correcto funcionamiento de los componentes que se añadan a nuestro framework.

CBaseDatos.php (/framework/clases/bd/CBaseDatos.php).

Representará una conexión a una base de datos MySql usando la API mysqli. Tiene los siguientes elementos.

- __construct($Servidor, $usuario, $contrasenia, $nombreBD). Se encarga de crear un objeto mysqli.
- error() y mensajeError() que indican respectivamente el código de error y el mensaje de error al conectar a MYSQL. El código de error se numera como 0 (no hay error), 1 (error xxx), etc.
- getEnlace(). Devuelve la conexión mysql creada.
- crearConsulta($sentencia). Devuelve un objeto CCommand correspondiente a la sentencia sql indicada.
- cerrarConexion(). Cierra la conexión.

- beginTran(). Inicia una nueva transacción.
- commit(). Confirma la transacción.
- rollback(). Deshace la transacción.

CCommnad.php (/framework/clases/bd/CCommand.php).

Representa una consulta a la base de datos. Tiene los siguientes elementos.

- __construct($conexion, $sentencia). Constructor al que se pasa un objeto CBaseDatos y una sentencia a ejecutar. Se encarga de ejecutar la sentencia.
- Métodos públicos error() y mensajeError() devuelven respectivamente el código y el mensaje de error producidos.
- Método numFilas(). Devuelve el numero de filas que se ha obtenido al ejecutar la consulta. Si se ha producido un error o la consulta no devuelve filas, este método devuelve false.
- Método público filas(). Devuelve un array con todas las filas obtenidas de la consulta (como array asociativo). Si se ha producido algún error devuelve false. Si la sentencia ejecutada no devuelve un conjunto resultado se devuelve false.
- Método público fila(). Devuelve una fila asociativa correspondiente a una fila del conjunto resultado de la consulta.
- Método público idGenerado(). Devuelve el id generado automáticamente para consultas de inserción o false si no se ha generado o hay algún error.
- Método público free(). Libera el conjunto resultado.

Los objetos de base de datos se pueden acceder directamente. Un código posible podría ser:

```
$conex=new CBaseDatos("127.0.0.1","root","root","relacion6");

//para devolver las filas para una sentencia sql
$filas=$conex->crearConsulta("select * from articulos")->filas();

//para una insercion
$consulta=$conex->crearConsulta("insert into articulos .......");
$id=$consulta->idGenerado();
```

CGeneral (/framework/clases/general/CGeneral.php).

Esta clase incluye algunos métodos estáticos útiles. Los métodos son:

- fechaMysqlANormal($fecha). Devuelve una fecha en formato ‘dd/mm/yyyy’ a partir de la fecha $fecha en formato Mysql (‘yyyy-mm-dd’).
- fechaNormalAMysql($fecha). Devuelve una fecha en formato Mysql (‘yyyy-mm-dd’) a partir de la fecha $fecha en formato normal (‘dd/mm/yyyy’).
- addSlashes($cadena). Devuelve una cadena en la que se ha escapado el carácter ‘.
- stripSlashes($cadena). Devuelve una cadena en la que se ha quitado el escape del carácter ‘.

CValidaciones (/framework/clases/general/CValidaciones).

Esta clase contiene una serie de métodos estáticos relacionados con operaciones de validación.

- validaDNI($dni,&$partes). Comprueba el $dni devolviendo true en caso de que sea correcto o false en caso contrario. Si es válido devuelve los elementos en el parámetro $partes.
- validaEntero**(**&$numero, $valorDefecto, $min, $max). Este método comprueba si $número se encuentra en el rango indicado por $min y $max. Devuelve true si es válido y false en caso contrario. Si no es válido asigna a $numero el valor $valorDefecto.
- validaReal**(**&$numero, $valorDefecto, $min, $max). Este método comprueba si $número se encuentra en el rango indicado por $min y $max. Devuelve true si es válido y false en caso contrario. Si no es válido asigna a $numero el valor $valorDefecto.
- validaCodPostal**(**&$codpostal**)**. Este método devuelve true si es válido el código postal y false en caso contrario. Si no es válido asigna al $codPostal el valor '00000'
- validaEMail(&$email,$valorDefecto="aa@aaa.es"). Este método devuelve true si es válido el email. Si no es válido devuelve false y asigna a $email el valor por defecto.
- validaLista(&$elemento,$lista). Esta función devuelve true si encuentra $elemento como valor en la $lista.
- validaFecha(&$fecha). Devuelve true si la fecha es válida en el formato "dd/mm/yyyy" y false en caso contrario. Si es válida la ajusta para que el día y el mes aparezcan con dos dígitos (01, 02, ….).
- validaHora(&$hora). Devuelve true si la fecha es válida en el formato "hh:mm:ss". Ajusta la fecha para que las horas, minutos y segundos queden con dos dígitos (00, 01, 02,….).

4.1.9.- AMPLIACION DEL FRAMEWORK.

El framework puede ser ampliado con los componentes que vamos creando, permitiendo así una mayor funcionalidad. Por ejemplo podemos agregar componentes para el control de la sesión, para el control de acceso (clases ACL, …), componentes para diseño de interfaz como tablas, menús, paginadores, etc.

Para poder incorporar un nuevo componente debe indicarse en el framework como localizarlo. Este se realiza a través del registro del componente en la clase Sistema.

El componente debe situarse en un fichero php de nombre igual a la clase. Después se debe guardar en una carpeta dentro de framework/clases. Una vez incorporado se agrega la línea apropiada en la clase Sistema en la propiedad estática $_clasesBase.

Como ejemplo podemos ver el componente CBaseDatos. Lo primero es tener el fichero CBaseDatos.php (dentro encontramos la clase CBaseDatos). Este fichero se puede encontrar en la carpeta /framework/clases/bd. Por último se tiene registrada la clase CBaseDatos en Sistema:

```
class Sistema
{
    static private $_clasesBase=array(
        "CAplicacion"=>"/base/CAplicacion.php",
        "CGeneral"=>"/general/CGeneral.php",
        "CValidaciones"=>"/general/CValidaciones.php",
        "CControlador"=>"/mvc/CControlador.php",
        "CActiveRecord"=>"/mvc/CActiveRecord.php",
        "CHTML"=>"/forms/CHTML.php",
        "CBaseDatos"=>"/bd/CBaseDatos.php",
        "CCommand"=>"/bd/CCommand.php");
```

Como se observa la ruta indicada para el fichero es relativo a /framework/clases.

A partir de ese momento es posible usar la clase CBaseDatos directamente:

```
//creo el enlace a la base de datos
$enlace=new CBaseDatos("localhost","root","root","relacion6");
```

En muchas ocasiones nos interesa tener una instancia, de uno de los componentes del framework, que se cree automáticamente para estar disponible en cualquier punto de nuestra aplicación. Esto se consigue registrando esa instancia en la clase CAplicación.

Volviendo con CBaseDatos, se puede acceder a una instancia creada de forma automática mediante ***Sistema::App()->BD()***

```
$sentencia="select * from articulos";

$consulta=Sistema::App()->BD()->crearConsulta($sentencia);
$filas=$consulta->filas();
```

Esto se consigue añadiendo a la clase CAplicacion:

- Una propiedad privada que contendrá la instancia.

- Un método público que devuelva la instancia.
- Creando la instancia en el constructor.

En el caso de CBaseDatos el código correspondiente para crear la instancia en CAplicacion es:

```
class CAplicacion
{
    ……..
    //se define la propiedad privada para la instancia
    private $_BD;

    public function __construct($config)
    {
        //se carga la configuracion

        ……..

        //se inicializa la instancia en el constructor
        if (isset($config["BD"]) && $config["BD"]["hay"]===true)
        {
            $this->_BD=new
            CBaseDatos($config["BD"]["servidor"],
                $config["BD"]["usuario"],
                $config["BD"]["contra"],
                $config["BD"]["basedatos"]);
        }
    }

    //método público para devolver la instancia.
    public function BD()
    {
        return $this->_BD;
    }
```