PHP 5: Taller de Aplicaciones MVC Modulares

Eugenia Bahit

Índice de contenidos

Introducción a la Arquitectura de Software

¿Qué es la arquitectura de software?

Es necesario aclarar, que no existe una definición única, exacta, abarcadora e inequívoca de “arquitectura de software”. La bibliografía sobre el tema es tan extensa como la cantidad de definiciones que en ella se puede encontrar. Por lo tanto trataré, no de definir la arquitectura de software, sino más bien, de introducir a un concepto simple y sencillo que permita comprender el punto de vista desde el cual, este libro abarca a la arquitectura de software pero, sin ánimo de que ello represente “una definición más”.

A grandes rasgos, puede decirse que “la Arquitectura de Software es la forma en la que se organizan los componentes de un sistema, interactúan y se relacionan entre sí y con el contexto, aplicando normas y principios de diseño y calidad, que fortalezcan y fomenten la usabilidad a la vez que dejan preparado el sistema, para su propia evolución”.

Atributos de calidad

La Calidad del Software puede definirse como los atributos implícitamente requeridos en un sistema que deben ser satisfechos. Cuando estos atributos son satisfechos, puede decirse (aunque en forma objetable), que la calidad del software es satisfactoria. Estos atributos, se gestan desde la arquitectura de software que se emplea, ya sea cumpliendo con aquellos requeridos durante la ejecución del software, como con aquellos que forman parte del proceso de desarrollo de éste.

Atributos de calidad que pueden observarse durante la ejecución del software

1. Disponibilidad de uso
2. Confidencialidad, puesto que se debe evitar el acceso no autorizado al sistema
3. Cumplimiento de la Funcionalidad requerida
4. Desempeño del sistema con respecto a factores tales como la capacidad de respuesta
5. Confiabilidad dada por la constancia operativa y permanente del sistema
6. Seguridad externa evitando la pérdida de información debido a errores del sistema
7. Seguridad interna siendo capaz de impedir ataques, usos no autorizados, etc.

Atributos de calidad inherentes al proceso de desarrollo del software

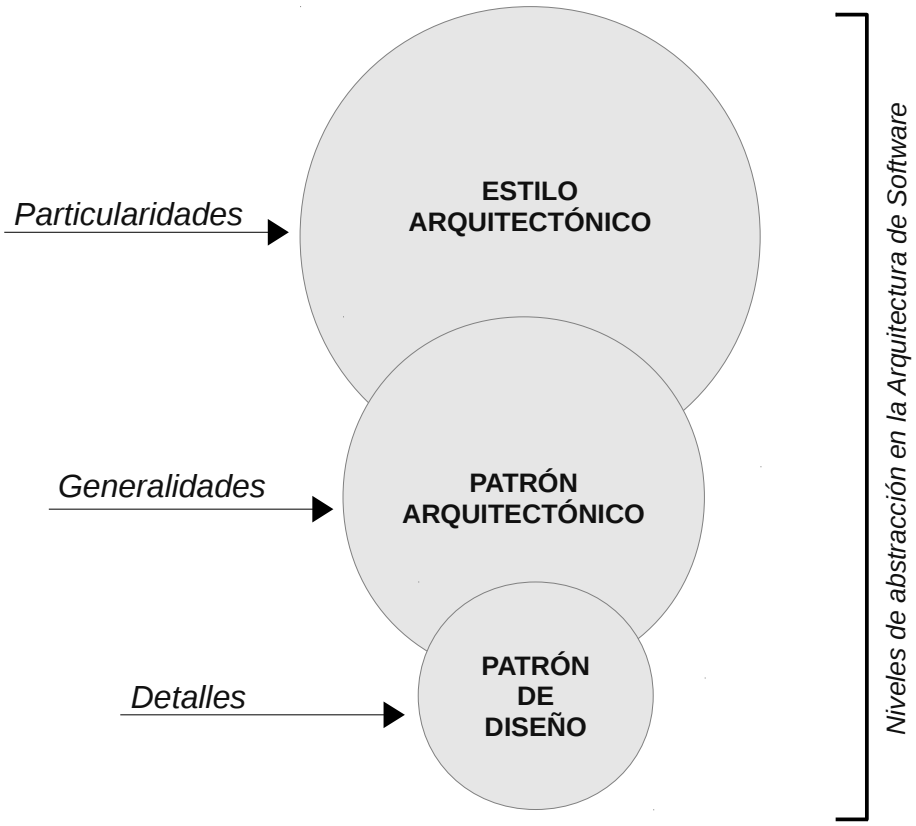
1. Capacidad de Configurabilidad que el sistema otorga al usuario a fin de realizar ciertos cambios
2. Integrabilidad de los módulos independientes del sistema
3. Integridad de la información asociada
4. Capacidad de Interoperar con otros sistemas (interoperabilidad)
5. Capacidad de permitir ser modificable a futuro (modificabilidad)
6. Ser fácilmente Mantenible (mantenibilidad)
7. Capacidad de Portabilidad, es decir que pueda ser ejecutado en diversos ambientes tanto de software como de hardware
8. Tener una estructura que facilite la Reusabilidad de la misma en futuros sistemas
9. Mantener un diseño arquitectónico Escalable que permita su ampliación

(escalabilidad)
10. Facilidad de ser Sometido a Pruebas que aseguren que el sistema falla cuando es lo que se espera (testeabilidad)

Niveles de abstracción

Podemos decir que la AS se compone de tres niveles de abstracción bien diferenciados: Estilo Arquitectónico, Patrón Arquitectónico y Patrón de Diseño. Existe una diferencia radical entre estos tres elementos, que debe marcarse a fin de evitar las grandes confusiones que inevitablemente, concluyen en el mal entendimiento y en los resultados poco satisfactorios. Éstos, son los que en definitiva, aportarán “calidad” al sistema resultante. En lo sucesivo, trataremos de establecer la diferencia entre estos tres conceptos, viendo como los mismos, se relacionan entre sí, formando parte de un todo: la arquitectura de software.

Estilo Arquitectónico, Patrón Arquitectónico y Patrón de Diseño, representan -de lo general a lo particular- los tres niveles de abstracción que componen la Arquitectura de Software.



Estilo Arquitectónico

El estilo arquitectónico define a niveles generales, la estructura de un sistema y cómo éste, va a comportarse. Mary Shaw y David Garlan, en su libro “Software Architecture” (Prentice Hall, 1996), definen los estilos arquitectónicos como la forma de determinar el

los componentes y conectores de un sistema, que pueden ser utilizados a instancias del estilo elegido, conjuntamente con un grupo de restricciones sobre como éstos pueden ser combinados:

> *"[...] an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined [...]"*

Mary Shaw y David Garlan -en el mismo libro-, hacen una distinción de estilos arquitectónicos comunes, citando como tales a:

1. Pipes and filters (filtros y tuberías)
2. Data Abstraction and Object-Oriented Organization (Abstracción de datos y organización orientada a objetos)
3. Event-based (estilo basado en eventos)
4. Layered Systems (Sistemas en capas)
5. Repositories (Repositorios)
6. Table Driven Interpreters

Viendo la clasificación anterior, es muy frecuente que se encuentren relaciones entre los estilos arquitectónicos y los paradigmas de programación. Sin embargo, debe evitarse relacionarlos en forma directa.

Patrón Arquitectónico

Un patrón arquitectónico, definirá entonces, una plantilla para construir el Software, siendo una particularidad del estilo arquitectónico elegido.

En esta definición, es donde se incluye a MVC, patrón que a la vez, puede ser enmarcado dentro del estilo arquitectónico orientado a objetos (estilo arquitectónico basado en el paradigma de programación orientada a objetos).

Patrón de Diseño

Dentro de niveles de abstracción de la arquitectura de Software, los patrones de diseño representan el nivel de abstracción más detallado. A nivel general, nos encontramos con el Estilo Arquitectónico. En lo particular, hallamos al Patrón Arquitectónico y, finalmente, el Patrón de Diseño es "el detalle".

Matt Zandstra en su libro "PHP Objects, Patterns and Practice" (Apress, 2010) define los patrones de diseño como:

> *"[…] is a problem analyzed with good practice for its solution explained [...]" (un problema analizado con buenas prácticas para su solución explicada)*

Un patrón de diseño, entonces, es un análisis mucho más detallado, preciso y minucioso de una parte más pequeña del sistema, que puede incluso, trabajar en interacción con otros patrones de diseño. Por ejemplo, un *Singleton* puede coexistir con un *Factory* y éstos, a la vez, con un *Abstract Factory*.

En este sentido, un Patrón Arquitectónico como MVC, podría utilizar diversos patrones de diseño en perfecta coexistencia, para la creación de sus componentes.

Introducción al Patrón Arquitectónico MVC

MVC -por sus siglas en inglés, model-view-controller (modelo-vista-controlador)- es un patrón arquitectónico que nos permite desarrollar aplicaciones, manteniendo separada la lógica de negocios de las vistas, utilizando un "controlador" como conector (o intermediario) entre ambas.

Entendiendo el funcionamiento de MVC

En MVC, todo comienza con una petición del usuario. En una aplicación Web, la petición del usuario podría ser, por ejemplo, "agregar un nuevo registro".

- **¿Cómo realiza esta petición el usuario?** A través del navegador.
- **¿Cómo se identifica la petición?** Por medio de la URL ingresada por el usuario.

Un ejemplo:

Tenemos una aplicación Web, cuyos módulos son:

```
Módulo de Usuarios
Módulo de Proyectos
```

Cada módulo, a la vez, se encontrará dividido en "modelos" (objetos):

```
Módulo de Usuarios
  → Modelo Usuario
  → Modelo Permiso
```

Y estos modelos, ofrecerán diversos recursos (funcionalidades de cada modelo):

```
Módulo de Usuarios
  → Modelo Usuario
       → Recursos:
            → Agregar Usuario
            → Modificar Usuario
            → Eliminar Usuario
            → Obtener Usuario
```

Las peticiones del usuario, entonces, se realizarán vía navegador, siendo descifradas por la URL ingresada, la cual, guardará un formato sugerido como el siguiente:

```
dominio/modulo/modelo/recurso[/atributos]
```

Esto significa, que si el usuario desea agregar un nuevo usuario, su petición, debería ser:

```
http://app.dominio.com/usuarios/usuario/agregar-usuario/
```

- **¿A quién efectúa la petición el usuario?** Al controlador.
- **¿Cómo maneja el controlador, la petición del usuario?** A través de *Handler*.

Un *handler* (o "manejador") de peticiones, es un objeto encargado de gestionar las peticiones del usuario a nivel de la aplicación. Este *handler*, descifrará dichas peticiones, realizando un trabajo de pseudo ingeniería inversa, sobre la URI.

Por ejemplo, para la URI anterior, el handler podría realizar los siguientes pasos:

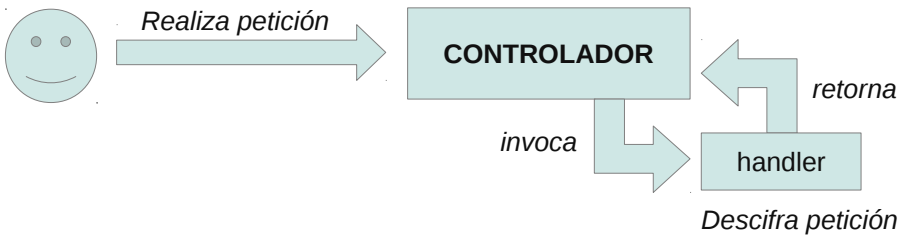
1) Identificar el dominio, eliminarlo del contexto y así solo obtener el último tramo de la URI:

```
$dominio = "http://{$_SERVER['SERVER_NAME']}";
$uri = "{$dominio}{$_SERVER['REQUEST_URI']}";
$ultimo_tramo = str_replace("{$dominio}/", NULL, $uri);
```

2) Hacer un explode del último tramo de la URI, para obtener un array de 3 elementos (módulo, modelo y recurso respectivamente):

```
$partes = explode("/", $ultimo_tramo);
```

3) Finalmente, el objeto handler, retornará al controlador, el módulo, el modelo y el recurso solicitados por el usuarios, y será el controlador, quien prosiga con el resto.

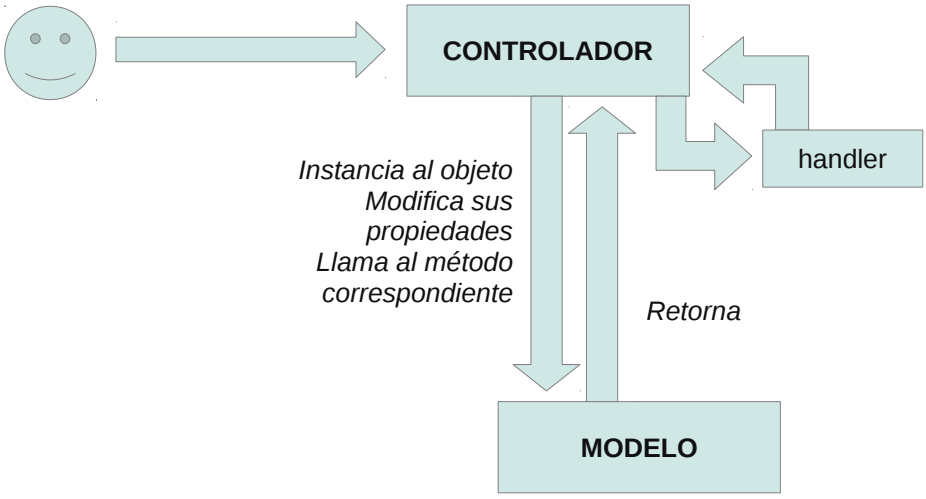


- **¿Qué hace el controlador una vez que recibe la información retornada por el handler?** La analiza para saber a que modelo deberá instanciar.

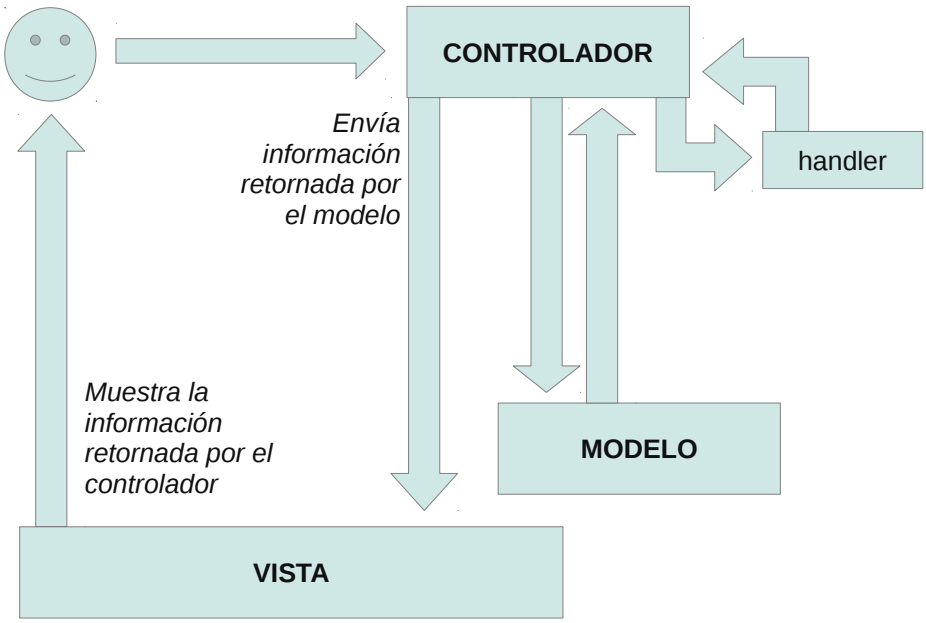
Una vez que el controlador recibe la información retornada por el *handler*, procede a analizarla en conjunto.

```
El handler me envió la siguiente información:

Módulo: usuarios
Modelo: usuario
```

```
Recurso: agregar-usuario

Entonces, debo agregar un nuevo usuario. Para ello, voy a instanciar
el objeto, modificar sus propiedades y llamar al método
correspondiente.
```



- **¿Qué hace el controlador con la información retornada por el modelo?** La entrega a la vista.
- **¿Para qué el controlador le entrega la información a la vista?** Para que ésta, se la muestre al usuario.



Modelos en MVC

Un modelo, en MVC, es una clase. Como patrón que guía nuestras arquitecturas, MVC nos obliga a escribir clases “puras” que respeten el verdadero espíritu de la orientación a objetos.

Objetos puros: características de un modelo

1. Una clase, debe representar solo y únicamente, un “modelo” para crear un objeto único.
2. Las propiedades de un objeto, siempre deberán ser -solo y únicamente- “sustantivos” cuyo valor, pueda ser definido por una cualidad o más de una.
3. Cuando una propiedad, sea definida por más de una cualidad, deberá considerarse la opción de componer el objeto: dicha propiedad, adquirirá las cualidades de otro objeto.
4. Las propiedades no compuestas del objeto -mayormente-, deben poder modificarse de forma directa, sin necesidad de requerir la llamada a un método.
5. Las propiedades compuestas -mayormente-, deberán contener un método -en la clase- del tipo set_propiedad(Objeto $objeto) para ser modificadas.
6. Los métodos de un objeto, deberán ser “acciones” intrínsecas del objeto y jamás, podrán representar acciones genéricas, que puedan ser consideradas ajenas al objeto. Por ejemplo, un objeto jamás podría contener entre sus métodos, una acción destinada a filtrar datos para prevenir inyecciones SQL.
7. Solo los modelos, podrán interactuar con la base de datos. No se podrá acceder a una base de datos, desde un ámbito externo a los modelos.
8. Preferentemente, cada modelo accederá a la base de datos, mediante otro objeto-conector (capa de abstracción que pertenecerá al núcleo de la aplicación).

Creando modelos bajo MVC en PHP

Herencia y Composición

Piensa en una camisa como un objeto. ¿Qué propiedades podemos encontrar? Una camisa tendrá un tipo de tela determinado con un color específico, botones, un talle, etc. ¿cierto? Podríamos decir que la clase para crear una camisa, podría tener las siguientes propiedades:

```
class Camisa {
    public $tela = "seda";
    public $botones = 8;
    public $color = "blanca";
    public $talle = 42;
}
```

Imagínate que trabajas en el salón de ventas de una camisería y un cliente te pregunta ¿De qué material son los botones de la camisa? ¿Son botones ciegos o con ojales? ¿qué color son? Podríamos deducir, que cada uno de los 8 botones, tiene características particulares:

```
boton.material = "nylon"
boton.color = "marfil"
boton.ojales = 2
```

Cuando hablamos de una propiedad con múltiples cualidades, como Arquitectos, debemos pensar -en principio- en dos opciones: **herencia** o **composición**.

Un botón ¿no es acaso un objeto con atributos propios? Es allí, donde probablemente, comiencen a surgir dudas. La primera pregunta que deberás hacerte, entonces, es: ¿Qué relación existe entre Camisa y Botón? Para responder a esta pregunta, deberás hacerlo formulando nuevas incógnitas:

- ¿Es Objeto A una extensión ampliada de Objeto B? Si la respuesta es sí, pensarás en herencia. Si la respuesta es no, te harás la siguiente pregunta:
- Objeto B ¿forma parte integrante de Objeto A? Si la respuesta en sí, pensarás entonces en composición.

Entonces, la pregunta es: ¿Es Camisa una extensión ampliada de Botón? La respuesta clara es “No”. Pues Camisa y Botón, nada tienen en común. Sin embargo, Botón, forma parte integrante de Camisa:

```
# archivo: models/camisa.php

class Camisa {
    public $tela = "seda";
    public $botones = array();
    public $color = "blanca";
    public $talle = 42;

    public function set_botones(Boton $boton, $cuantos_botones) {
        $iterar = 0;
        while($iterar < $cuantos_botones) {
            $this->botones[] = $boton;
            $iterar++;
        }
    }

}

# archivo: models/boton.php

class Boton {
    public $material = "nylon";
```

```
    public $color = "marfil";
    public $ojales = 2;
}

# archivo: crear_camisa.php

$camisa = new Camisa();
$camisa->set_botones(new Boton(), 8);

$botones = count($camisa->botones);

if($camisa->botones[0]->ojales < 1) {
    $tipo_boton = "ciego";
} else {
    $tipo_boton = "ojalado";
}

print "La camisa de {$camisa->tela} color {$camisa->color}
tiene {$botones} botones de {$camisa->botones[0]->material} {$tipo_boton}s
en color {$camisa->botones[0]->color}");

/*
Salida:
La camisa de seda color blanca tiene 8 botones de nylon ojalados en color marfil
*/
```

Acceso a bases de datos

Un “objeto-conector” a nivel del *core*, que actúe como intermediario entre los modelos y la base de datos, será muy recomendable para respetar el verdadero espíritu del patrón arquitectónico MVC.

Este objeto, será una clase preferentemente estática. Es decir, que necesitará ser instanciada para hacer uso de los métodos destinados a acceder a las bases de datos.

Código fuente de una capa de abstracción a nivel del core

```
class DBObject {

    protected static $conn;
    protected static $stmt;
    protected static $reflection;
    protected static $sql;
    protected static $data;
    public static $results;

    protected static function conectar() {
        self::$conn = new mysqli(DB_HOST, DB_USER, DB_PASS, DB_NAME);
        self::$conn->autocommit(False);
    }

    protected static function preparar() {
        self::$stmt = self::$conn->prepare(self::$sql);
        self::$reflection = new ReflectionClass('mysqli_stmt');
    }

    protected static function set_params() {
```

```
        $method = self::$reflection->getMethod('bind_param');
        $method->invokeArgs(self::$stmt, self::$data);
    }

    protected static function get_data($fields) {
        $method = self::$reflection->getMethod('bind_result');
        $method->invokeArgs(self::$stmt, $fields);
        while(self::$stmt->fetch()) {
            self::$results[] = unserialize(serialize($fields));
        }
    }

    protected static function finalizar() {
        self::$stmt->close();
        self::$conn->close();
    }

    public static function ejecutar($sql, $data, $fields=False) {
        self::$sql = $sql;
        self::$data = $data;
        self::conectar();
        self::preparar();
        self::set_params();
        self::$stmt->execute();
        if($fields) {
            self::get_data($fields);
        } else {
            self::$conn->commit();
        }
        self::finalizar();
    }

}
```

Implementación desde el modelo:

```
require_once('/app/core/db_object.php');

class Boton {

    public $material = NULL;
    public $color = NULL;
    public $ojales = NULL;
    public $boton_id = 0;

    public function save() {
        $sql = "INSERT INTO boton (material, color, ojales) VALUES (?, ?, ?)";

        $data = array("ssi",
                      "{$this->material}",
                      "{$this->color}",
                      "{$this->ojales}");

        return DBObject::ejecutar($sql, $data);
    }

    public function get() {
        $sql = "SELECT material, color, ojales FROM boton WHERE boton_id > ?";

        $data = array("i", "{$this->boton_id}");
```

```
        $fields = array("Material"=>"",
                        "Color"=>"",
                        "Ojales"=>"");

        DBObject::ejecutar($sql, $data, $fields);
        return DBObject::$results;
    }
}
```

Object Relational Mapping (ORM)

Si observamos la clase `Boton`, notaremos que el método `save()`, declara él mismo la sentencia SQL que será utilizada por `DBObject` para agregar registros en la base de datos. Pero una capa de abstracción a bases de datos, podría -y debería- otorgar una abstracción mucho más completa.

El **Mapeo Relacional de Objetos** (Object Relational Mapping) es una técnica de programación que nos permite vincular los objetos (modelos) de nuestra aplicación, con una base de datos relacional, otorgándonos de manera colateral, una abstracción completa entre los modelos y el motor de la base de datos.

Ventajas y desventajas del utilizar ORMs

Sin dudas, las principales **ventajas del ORM** -en orden de prioridades-, son:

1. **Independizar los modelos de las bases de datos**, lo cual nos obliga a razonar nuestra aplicación, 100% orientada a objetos (ya no pensaremos en como crear las tablas, solo crearemos nuestros modelos -modelar la app-).
2. **Independencia del motor de base de datos:** este punto es fundamental, ya que un buen ORM debe ser capaz de mapear los modelos de forma tal que las llamadas a sus métodos, sean capaces de conectarse a cualquier tipo de base de datos y general consultas con el lenguaje SQL de la misma. Esta característica, es la que permite una mayor portabilidad de la aplicación.
3. **Acelera el proceso de desarrollo**, puesto que se ahorra tiempo tanto en la creación y diseño de las bases de datos como en la escritura de las consultas.

Sin embargo, la utilización de ORM, trae aparejadas ciertas **desventajas** que no pueden ser obviadas al momento de tomar una decisión:

- **Lenguaje de consulta propio:** la mayoría de las librerías ORM disponibles en el mercado, poseen su propio lenguaje de consulta a bases de datos. Esta característica, que erróneamente es considerada una ventaja por muchos autores (debido al escaso conocimiento del lenguaje SQL) genera una pseudo-abstracción

(o abstracción falaz) que induce a errores en el modelado de objetos, puesto que estos pseudo-lenguajes que ofrecen los ORM, no son más que una especie de "lenguaje SQL resumido".

- **Reduce el rendimiento de la aplicación**, puesto que el proceso de conversión desde el pseudo-lenguaje al lenguaje SQL de la base de datos y el de ésta a objetos, demanda un mayor consumo de recursos.
- **Baja escalabilidad:** también derivado del uso de pseudo-lenguajes de consulta propio, cuanto más robusta y compleja se va haciendo una aplicación, mayor necesidad tendrá el programador, de escribir consultas en lenguaje SQL "crudo".

ORM Frameworks para PHP

En PHP, existen prestigiosos frameworks ORM, que podríamos utilizar en nuestra aplicaciones. Entre los más destacados, podremos encontrar:

- **Doctrine:** http://www.doctrine-project.org/
- **Propel:** http://www.propelorm.org/
- Entre otros.

Bibliografía recomendada

Para **complementar la lectura sobre modelos en PHP**, se recomienda leer bibliografía sobre:

- Programación orientada a objetos
- Patrones de diseño

Libros recomendados:

- PHP Objects, Patterns and Practice (Third Edition) – Matt Zandstra (Apress, 2010)
- Pro PHP – Patterns, Frameworks, Testing and More (Part I: OOP and Patterns) – Kevin McArthur (Apress, 2008)
- Pro PHP Programming (Chapter I:Object Orientation) – Peter MacIntyre, Brian Danchilla & Mladen Gogala (Apress, 2011)

Las vistas

Embeber código HTML, CSS y JavaScript en el código PHP, con MVC queda en el olvido. Sin embargo, muchos Frameworks MVC para PHP, aún conservan esa mala costumbre que los programadores, solemos arrastrar de la programación estructurada.

Basaremos este curso, en el espíritu original y más desarrollado de MVC, el cual nos propone una completa abstracción de las vistas, subdividiendo éstas, en la parte lógica (código PHP que se encargará de hacer un *render* de la parte gráfica) y en la GUI (parte gráfica inherente al diseño gráfico), incorporando algunos *Katas* para no volver a embeber código y lograr mantener un verdadero diseño orientado a objetos.

¿Por dónde empezar a desarrollar las vistas?

Lo primero que debemos tener en cuenta, es la completa abstracción de la lógica de su correspondiente GUI (Grafical User Interface).

El diseño gráfico es un verdadero arte, que abarca múltiples disciplinas y que, como programadores, el peor error que podemos cometer, es querer convertir al diseño gráfico en un "producto científico", ya que **la lógica utilizada en el arte, debe indefectiblemente estar sometida al ingenio y creatividad**, puesto que **de lo contrario, se desaprovecharían todas las virtudes que el arte**, tiene para ofrecernos a fin de hacer nuestro Software "más humano".

> *La GUI de nuestras aplicaciones, debe -sí y solo sí-* ***basarse en cuestiones inherentes al arte y la creatividad, al servicio de la usabilidad y experiencia del usuario.***

El primer paso, entonces, consistirá en crear las interfaces gráficas de la aplicación. Es sumamente importante, considerar la posibilidad de contar con artistas expertos en el Diseño Gráfico y experiencia del usuario, puesto que son los únicos que cuentan con la capacidad necesaria y la autoridad profesional suficiente, para crear verdaderas GUI.

En este curso, no haremos demasiado énfasis -ni ahondaremos- en temas relativos al diseño gráfico, puesto que el curso, está basado en la AS. Los ejemplos que utilizaremos, lejos están de ser tomados como parámetro de arte aplicado al diseño gráfico. Nos enfocaremos entonces, solo y únicamente, en aquellos aspectos que como programadores y/o arquitectos, debemos considerar transmitir a los profesionales encargados de diseñar la GUI de nuestras aplicaciones.

Desarrollando la GUI

Componentes de la GUI

La GUI deberá estar compuesta, por todos aquellos archivos -y datos- estáticos, que son ejecutados del lado del cliente. Entre ellos, nos encontramos con archivos **HTML, CSS,**

JS, imágenes, archivos de audio y video y cualquier otro tipo de documento estático (como PDFs, ODTs, etc...).

Arquitectura

Refiriéndonos a Arquitectura como “Arquitectura de Software” (AS) y no, como Arquitectura de la Información (AI), la GUI de nuestra aplicación, deberá estar concentrada en un solo lugar. Esto puede ser:

– Un directorio exclusivo dentro de la App

```
myapp/
└── site_media /
```

– Un servidor independiente del servidor de la aplicación

La **estructura más recomendada** que puede tener la GUI, es aquella que siga una organización como la que sigue:

```
myapp/
└── site_media /
    ├── css/
    │   ├── core/         Archivos CSS aplicables a toda la app
    │   ├── module-a/     Archivos CSS pertenecientes solo al módulo A
    │   └── module-b/     Archivos CSS pertenecientes solo al módulo B
    ├── html/
    │   ├── core /
    │   ├── module-a /
    │   └── module-b /
    ├── img/
    │   ├── core /
    │   ├── module-a /
    │   └── module-b /
    └── js/
        ├── core /
        ├── module-a /
        └── module-b/
```

Esta estructura de directorios, si bien **es la más escalable** de todas, lógicamente podrá variar de acuerdo a las necesidades de cada app. Por ejemplo, también podrá incluir otros directorios destinados a almacenar archivos de audio, de video, PDFs, etc.

Preparando la GUI para interactuar con la lógica

Los archivos de la GUI (más específicamente, aquellos destinados como maqueta de nuestra app -generalmente, archivos HTML), solo podrán contener lenguaje de marcado| diseño exclusivamente. Pero ¿Qué sucede con aquellos datos que deberán sustituirse dinámicamente?

```
<!doctype html>
<head>
    <meta charset="utf-8">
    <title>Listado de Camisas</title>
</head>

<body>
    <header>
        <h1>MyApp: Camisas</h1>
        <nav>
            <ul>
                <li><a href="menu-option-1"
                     title="Menu Option 1">Ver camisas</a></li>
                <li><a href="menu-option-2"
                     title="Menu Option 2">Agregar camisa nueva</a></li>
            </ul>
        </nav>
    </header>
    <section>
        <header>
            <h2>Camisas creadas</h2>
        </header>
        <table>
            <tr>
                <th>ID</th>
                <th>Camisa</th>
                <th>Tela</th>
            </tr>
            <tr>
                <td>1</td>
                <td>Tombolini Home</td>
                <td>Seda italiana blanca</td>
            </tr>
        </table>
    </section>
</body>

</html>
```

Si miramos el código anterior, observando el texto en **negritas**, podremos notar que de tratarse de un *template*, esos datos, deberán ser plasmados dinámicamente. Es en estos casos, donde el diseñadores, deberá utilizar “comodines” que luego, la lógica de esa vista, se encargará de renderizar.

Comodines ¿Qué son y cómo implementarlos?

Un comodín, puede ser cualquier texto plano que nos ayude a identificar -como programadores, en nuestra lógica de negocios- aquellos datos que necesitan ser sustituidos dinámicamente.

Estos comodines, deberán seguir un patrón que, por un lado, permita al diseñador tener una vista previa real de su diseño y por otro, permita a nuestra lógica de negocios, identificarlos con facilidad.

Algunos ejemplos:

```
[TEXTO IDENTIFICADOR]
{TEXTO IDENTIFICADOR}
```

Lo importante, es que en toda la GUI, **siempre** se utilice el mismo patrón:

```
<!doctype html>
<head>
    <meta charset="utf-8">
    <title>{TITULO DE PAGINA}</title>
</head>

<body>
    <header>
        <h1>MyApp: {MODULO}</h1>
        <nav>
            {MENU}
        </nav>
    </header>
    <section>
        <header>
            <h2>{SUBTITULOS}</h2>
        </header>
        {TABLA}
    </section>
</body>

</html>
```

{TABLA} y **{MENU}** serán a la vez, dos nuevos archivos HTML. En el segundo caso (menú), será sencillo:

```
<ul>
    <li><a href="ver-camisas">Ver camisas</a></li>
    <li><a href="crear-camisa">Crear camisa</a></li>
</ul>
```

Pero en el caso de la tabla **¿cómo haremos para obtener identificadores iterativos?** Si miramos el código de la tabla que listará las camisas, podremos notar que la fila destinada a la descripción de cada registro, deberá repetirse tantas veces como registros se encuentren:

```
<table>
    <tr>
        <th>ID</th>
        <th>Camisa</th>
        <th>Tela</th>
    </tr>
    <!-- a partir de aquí los datos deben iterar -->
    <tr>
        <td>{CAMISA ID}</td>
        <td>{CAMISA DESCRIPCION}</td>
        <td>{CAMISA DATOS TELA}</td>
```

```
    </tr>
    <!-- desde aquí, ya no iteran -->
</table>
```

De la misma forma que en el código anterior, un comentario explica que esa fila debe ser iterada, será como los diseñadores, deberán especificárnoslo a nosotros, solo que de forma más simple y siguiendo algún patrón, como por ejemplo:

Para identificar dónde inicia una iteración:

```
<!-- iniciar-loop: NOMBRE DE IDENTIFICADOR DEL LOOP -->
```

Para identificar dónde finaliza:

```
<!-- finalizar-loop: NOMBRE DE IDENTIFICADOR DEL LOOP -->
```

Por ejemplo, el diseñador podría hacer lo siguiente:

```
    <!-- iniciar-loop: CAMISA -->
    <tr>
        <td>{CAMISA ID}</td>
        <td>{CAMISA DESCRIPCION}</td>
        <td>{CAMISA DATOS TELA}</td>
    </tr>
    <!-- finalizar-loop: CAMISA -->
```

Diseñando la lógica de negocios

La lógica de negocios de las vistas, es aquella que se encargará de traer el contenido estático y sustituir los “comodines” dinámicamente, con los datos que le sean entregados por el controlador.

La lógica de negocios, tendrá un **diseño distribuido**:

- Una parte -reutilizable y común a toda la aplicación- a nivel del *core*
- Otra parte, a nivel del módulo, mediante la cual, cada modelo tendrá su propia lógica

```
myapp/
├── core
│   └── view.php
├── module
│   ├── models
│   │   ├── boton.php
│   │   └── camisa.php
│   └── views
│       ├── boton.php
│       └── camisa.php
└── site_media
```

La lógica principal, será la del *core*. Ésta, consistirá en una librería que nos provea de los métodos necesarios para:

- Traer los *templates*
- Identificar comodines e iteraciones
- Sustituir los comodines dinámicamente
- Retornar el *render* del *template*

La lógica de cada modelo, será una extensión de la lógica principal, con métodos propios que requieran de los métodos heredados de la clase principal, para que cada *render*, sea específico y particular.

Lógica principal

Como comentamos anteriormente, la lógica principal será una librería -una clase- con métodos que se encarguen, a niveles generales, de traer *templates* y renderizarlos. Un ejemplo de ello, sería la siguiente clase (a nivel del *core*), llamada **`RenderTemplate`**:

```
# archivo /myapp/core/render_template.php

class RenderTemplate {

    public $file = "";
    public $data = array();
    protected $comodines = array();
    protected $values = array();
    public $html = "";


    # Traer contenido HTML de una plantilla
    protected function get_html($str=NULL) {
        return isset($str) ? $str : file_get_contents($this->file);
    }

    # Setear comodines y valores
    public function set_data($data=array()) {
        $this->comodines = array_keys($data);
        $this->values = array_values($data);
        $this->set_comodines();
```

```
    }

    # Modificar comodines (envolver entre llaves)
    private function set_comodines() {
        foreach($this->comodines as &$comodin) {
            $comodin = "{" . $comodin . "}";
        }
    }

    # Renderizar plantilla
    public function render_template($str=NULL) {
        $this->html .= str_replace($this->comodines,
                                   $this->values,
                                   $this->get_html($str));
    }

}
```

Esta clase, podrá ser heredada, por cualquier vista, pudiendo acceder y modificar cualquier propiedad o método, público o protegido.

Veamos un ejemplo de implementación en detalle:

Tenemos la siguiente plantilla HTML:

```
<!-- archivo: /myapp/site_media/html/template.html -->
<!doctype html>
<head>
    <meta charset="utf-8">
    <title>{TITULO}</title>
</head>

<body>
    <header>
        <h1>MyApp: {MODULO}</h1>
    </header>
    <section>
        <header>
            <h2>{SUBTITULO}</h2>
        </header>
        {CONTENIDO}
    </section>
</body>

</html>
```

De la plantilla anterior, necesitamos reemplazar 4 comodines: `TITULO`, `MODULO`, `SUBTITULO` y `CONTENIDO`. Hasta ahora, solo contamos con un modelo (`Boton`) y la lógica de la vista a nievel del core, `RenderTemplate`. No tenemos ni un controlador, ni la lógica de la vista para el modelo, así que probaremos implementar la clase `RenderTemplate` desde un archivo de pruebas ubicado en **`/myapp/module/prueba.php`**.

Lo primero que necesitaremos, será importar la clase `RenderTemplate`:

```
require_once("../core/render_template.php");
```

El paso siguiente, será definir un *array* asociativo, donde las claves sean los comodines y los valores, los datos por los cuales esos comodines, deberán ser reemplazados:

```
$data = array("TITULO" => "Administración de Camisas",
              "MODULO" => "Camisas",
              "SUBTITULO" => "Listado de Botones",
              "CONTENIDO" => "Aquí debo mostrar el listado de botones");
```

Finalmente, tendremos que:

```
# Crear una instancia de RenderTemplate
$view = new RenderTemplate();

# Preparar los datos a ser renderizados
$view->set_data($data);
```

El método `set_data` de la clase `RenderTemplate`, se encargará de:

- Dividir el *array* asociativo en dos. Un *array* lo destinará a los comodines (en la propiedad `$comodines` y el segundo, será almacenado en la propiedad `$values`
- A los comodines, les agregará las llaves de apertura y cierre, para generar el patrón que los identifique en la plantilla como tales.

```
# Modificar la propiedad $file indicando el path del template HTML a ser renderizado
$view->file = "../site_media/html/template.html";

# Llamar al método render_template
$view->render_template();

# Imprimir el render resultante
print $view->html;
```

`RenderTemplate,` almacena el resultado del *render* en la propiedad pública `$html.`

Pero una cuestión muy importante es **¿qué sucede si necesitamos hacer un render iterativamente?** Seguramente, podríamos utilizar esta misma clase, almacenando "mini-plantillas" con el código HTML de las iteraciones, y luego, hacer un bucle en la llamada a `set_data` y `render_template.` Pero esto, sería poco escalable, redundante y hasta tedioso.

Supongamos que `CONTENIDO,` a veces pueda ser un texto, otras un leyer con imágenes o una tabla (como vimos al principio) que deba completarse dinámicamente, dependiendo

de la cantidad de registros devueltos por una consulta a la DB.

Colocar la tabla, texto o layer en el template.html, sería contraproducente. Pues ese contenido es dinámico. Pero a la vez, ese mismo contenido que podrá variar, también variará de forma dinámica. Por ejemplo:

```
        <table>
            <tr>
                <th>Material</th>
                <th>Color</th>
                <th>Ojales</th>
            </tr>
            <!-- aquí comienza un loop con reemplazos dinámicos -->
            <tr>
                <td>{Material}</td>
                <td>{Color}</td>
                <td>{Ojales}</td>
            </tr>
            <!-- aquí finaliza el loop -->
        </table>
```

Lo más conveniente entonces, será que cada uno de esos contenidos (tablas, layers, textos, etc), se almacenen en una nueva plantilla HTML, identificando cada fragmento a ser reemplazado iterativamente, como vimos al comienzo:

```
<!-- archivo: /myapp/site_media/html/lista_botones.html -->

        <table>
            <tr>
                <th>Material</th>
                <th>Color</th>
                <th>Ojales</th>
            </tr>
            <!-- ini-loop: BOTONES -->
            <tr>
                <td>{Material}</td>
                <td>{Color}</td>
                <td>{Ojales}</td>
            </tr>
            <!-- end-loop: BOTONES -->
        </table>
```

Hasta aquí, logramos resolver el tema de las plantillas HTML (GUI de la vista). Para obtener el reemplazo del comodín `CONTENIDO`, solo necesitaríamos hacer un `file_get_contents` de esta nueva plantilla:

```
$data = array("TITULO" => "Administración de Camisas",
              "MODULO" => "Camisas",
              "SUBTITULO" => "Listado de Botones",
              "CONTENIDO" => file_get_contents('../site_media/html/lista_botones.html');
```

Sin embargo **¿Cómo le decimos a `RenderTemplate`, que además de reemplazar todos los comodines anteriores, deberá reemplazar los del loop "`BOTONES`" de**

nuestra tabla?

La solución, será crear una nueva clase reutilizable (a nivel del *core*), para renderizar datos dinámicamente, que a la vez, herede de `RenderTemplate`. Veamos como sería:

```
# Archivo: /myapp/core/render_data.php

class RenderData extends RenderTemplate {

    protected $pattern_tags = array();
    protected $pattern = "";

    # Setear pattern tags
    protected function set_tag($tag) {
        $this->pattern_tags = array("<!-- ini-loop: {$tag} -->",
                                    "<!-- end-loop: {$tag} -->");
    }

    # Obtener posición de los pattern tags
    private function get_position($tag) {
        return strpos($this->get_html(), $this->pattern_tags[$tag]);
    }

    # Obtener longitud total del pattern
    private function get_longitud() {
        $longitud = $this->get_position(1) - $this->get_position(0);
        return $longitud + strlen($this->pattern_tags[1]);
    }

    # Setear el contenido del pattern
    protected function set_pattern_content() {
        $this->pattern = substr($this->get_html(),
                                $this->get_position(0),
                                $this->get_longitud());
    }

    # Eliminar el patrón HTML y sustituirlo por el render
    private function delete_pattern() {
        $str_final = str_replace($this->pattern_tags, "", $this->html);
        return str_replace($this->pattern, $str_final, $this->get_html());
    }

    # Renderizar datos
    public function render_data($tag, $data) {
        $this->set_tag($tag);
        $this->set_pattern_content();

        foreach($data as $array) {
            $this->set_data($array);
            $this->render_template($this->pattern);
        }

        $this->html = $this->delete_pattern();
    }
}
```

Lo primero que hará esta nueva clase, es setear el patrón que identifica los tags de inicio y finalización del fragmento HTML a renderizar dinámicamente:

```
    protected function set_tag($tag) {
        $this->pattern_tags = array("<!-- ini-loop: {$tag} -->",
                                    "<!-- end-loop: {$tag} -->");
    }
```

Este método, necesitará que se le indique el nombre del identificador (tag) que en nuestro ejemplo, será "BOTONES".

El siguiente paso que realiza, es obtener el contenido HTML de todo el fragmento. De eso se encarga el método `set_pattern_content.` Veamos como actúa este método:

```
    protected function set_pattern_content() {
        $this->pattern = substr($this->get_html(), # método de la clase principal
                                $this->get_position(0),
                                $this->get_longitud());
    }
```

Este método, se vale de la función `substr` de PHP, para obtener un fragmento de código de una cadena de texto. La cadena de texto que almacena el código fuente HTML, será retornada por el método `get_html` de la clase principal:

```
    protected function get_html($str=NULL) {
        return isset($str) ? $str : file_get_contents($this->file);
    }
```

Si no se le pasa una `string,` retornará el contenido HTML del template seteado en la propiedad `$file`.

La posición de inicio para extraer una "sub-cadena", la obtiene a través del método get_`position()`:

```
    private function get_position($tag) {
        return strpos($this->get_html(), $this->pattern_tags[$tag]);
    }
```

Este método, recibe el índice del tag correspondiente (recordemos que se setean solo 2) y simplemente se vale de `strpos` para obtener la posición en la cual dicho tag se encuentra.

Y finalmente, para conocer la longitud total de la cadena, llama al método get_`longitud()`:

```
    private function get_longitud() {
        $longitud = $this->get_position(1) - $this->get_position(0);
```

```
        return $longitud + strlen($this->pattern_tags[1]);
    }
```

Este método, se basa simplemente en una lógica matemática. La longitud total del patrón, será igual a la diferencia entre la posición de inicio del tag de apertura del bucle, y de la del tag de cierre del bucle, más la longitud de ese tag. Veamos un pequeño ejemplo, que nos permita entender esto, haciendo una comprobación manual:

```
Soy <b>una frase</b> corta
```

Necesitamos obtener únicamente, el texto envuelto en negritas (pero también necesitamos que las etiquetas “b” nos sean retornadas.

Contemos los caracteres:

S	o	y		<	b	>	u	n	a		f	r	a	s	e	<	/	b	>		c	o	r	t	a
0	1	2	3	**4**	5	6	7	8	9	10	11	12	13	14	15	**16**	17	18	19	20	21	22	23	24	25

¿Cómo obtendremos una frase? Haremos lo siguiente:

1. Le diremos a `strpos` que nos retorne la posición de nuestro primer patrón . `strpos` nos retornará 4.
2. Luego le pediremos la posición de nuestro segundo patrón y nos devolverá 16.
3. Para conocer la longitud entre 4 y 16, necesitamos una simple resta: 16 – 4 = 12.
4. Ahora, cuenta (en la tabla) desde el 4, 12 casilleros. ¡Llegamos al casillero 15! Aún nos resta obtener nuestro patrón de cierre . Para ello, simplemente le pedimos a `strlen` que nos devuelva la longitud de nuestro patrón de cierre. Nos dirá 4. Entonces, solo sumamos esos 4 a 12: 12 + 4 = 16.
5. Si contamos 16 casilleros desde el 4, obtendremos nuestro patrón completo (desde el 4 al 19 inclusive).

> ***No te dejes confundir por la igualdad de los números:***
>
> *Da la casualidad que el inicio del primer patrón se produce en la posición 4 y que la longitud del segundo patrón también es 4. Esto es coincidencia. El primer patrón podría iniciar en la posición 7 y la longitud del segundo patrón podría ser 19! Sí o sí, el cálculo debe ser el que hemos hecho.*

Tenemos nuestra lógica a nivel del *core*, totalmente terminada. Contamos con dos clases de las cuales podremos heredar indistintamente. Pero si heredamos de `RenderData`, será como “matar dos pájaros de un tiro”.

Desde el archivo de prueba anterior, veremos como implementar entonces, ambas lógicas, a través de `RenderData`, pero esta vez, utilizaremos también nuestro modelo:

```
require_once('../core/settings.php');
require_once('../core/db_object.php');
require_once("../core/render_template.php");
require_once("../core/render_data.php");
require_once('models/boton.php');

$boton = new Boton();
$boton->boton_id = 10;
$data = $boton->get();

$view = new RenderData();
$view->file = "../site_media/html/lista_botones.html";
$view->render_data('BOTONES', $data);
$contenido = $view->html;


$data = array("TITULO" => "Administracion de Camisas",
              "MODULO" => "Camisas",
              "SUBTITULO" => "Listado de Botones",
              "CONTENIDO" => $contenido);

$view = new RenderTemplate();
$view->set_data($data);
$view->file = "../site_media/html/template.html";
$view->render_template();
print $view->html;
```

Lógica a nivel de módulo

Llegó la hora de desarrollar la lógica de las vistas para nuestros modelos. Recordemos tres cosas:

1. La lógica de la vista para cada modelo, será la encargada de mostrar (hacer un echo|print) el render del HTML al usuario;
2. Los datos que debe renderizar (datos dinámicos), se los deberá entregar el controlador (la lógica de la vista no puede conectarse con el modelo. Solo el controlador puede hacerlo).
3. Contamos con los métodos necesarios para renderizar una plantilla HTML y datos iterativamente. Entonces, “por descarte”, **nuestras vistas solo deberán encargarse de:**
    1. Definir los comodines a ser reemplazados
    2. Solo cuando sea necesario, asociar los comodines con los datos recibidos desde el controlador
    3. Modificar las propiedades de la lógica principal, definiendo los archivos a ser renderizados y haciendo la llamada a los métodos de esa lógica, que apliquen a cada caso
    4. Finalmente, deberá mostrarle esos renders al usuario

Características mínimas que debe tener la lógica

Debe ser una clase que herede de la lígica principal:

```
class NombreDeLaVista extends RenderData { }
```

Cada vista, debe estar asociada al modelo:

```
class BotonView extends RenderData { }
```

Mínimamente, cada vista deberá tener un método público (que pueda ser llamado desde el controlador), para cada uno de los métodos públicos del modelo al que pertenezca:

```
class BotonView extends RenderData {

    public function save_view() { }

    public function get_view() { }

}
```

Deberá estar preparada para recibir datos desde el controlador. Ya sea, contener una propiedad pública que el controlador pueda modificar:

```
class BotonView extends RenderData {

    public $data = NULL;

    public function save_view() { }

    public function get_view() { }

}
```

O sino, definir en cada método, el|los parámetro|s necesario|s para recibir estos datos:

```
class BotonView extends RenderData {

    public function save_view($mensaje='') { }

    public function get_view($registros=array()) { }

}
```

Los datos que la vista reciba desde el controlador, generalmente serán de alguno de los siguientes tipos:

- Mensajes retornados por el modelo al controlador
- Datos retornados al controlador por el modelo

Cuando no haya sido necesario que el controlador contactase con el modelo (por ejemplo, cuando la petición del usuario sea ver un formulario para agregar un nuevo registro), luego de hacer el switch, el controlador contactará directamente con la vista. Para ello, la

vista debe estar preparada, para recibir un llamado de petición. En estos casos, el tipo de datos recibido, será una petición que generalmente, estará representada por la llamada a un método:

```
class BotonView extends RenderData {

    public function save_view($mensaje='') { }

    public function get_view($registros=array()) { }

    public function mostrar_formulario_alta_boton() { }

}
```

En otros casos, la vista podrá tener un método público, que simplemente switchee la petición y ella misma, haga la llamada a un método de sí misma:

```
class BotonView extends RenderData {

    public function save_view($mensaje='') { }

    public function get_view($registros=array()) { }

    public function recibir_peticion($peticion) {
        switch ($peticion) {
            case 'form-nuevo-boton':
                $this->mostrar_formulario_alta_boton();
                break;
        }
    }

    private function mostrar_formulario_alta_boton() { }

}
```

Creando la lógica de la vista para un modelo

A continuación, basándonos en las características que definimos en el punto anterior y, en nuestro archivo de pruebas, crearemos la lógica de la vista encargada de mostrar el listado de botones. Esta clase, tendrá por el momento, solo tres métodos:

- Un método constructor, encargado de setear el diccionario de datos inicial (array asociativo donde las claves serán los comodines)
- Otro método público, para ser llamado por el controlador (será quien se encargue de hacer el render interativo)
- Y un tercer método privado para renderizar y mostrar el *template* general

Nuestro código, se verá así:

```
# Archivo: /myapp/module/views/boton.php

class BotonView extends RenderData {

    public function __construct() {
        $this->dict = array("TITULO" => "Administracion de Camisas",
                            "MODULO" => "Camisas",
```

```
                            "SUBTITULO" => "",
                            "MENU" => "",
                            "CONTENIDO" => "");
    }

    public function get_boton($registros=array()) {
        $this->file = "../site_media/html/lista_botones.html";
        $this->render_data('BOTONES', $registros);
        $this->dict['SUBTITULO'] = "Listado de Botones";
        $this->dict['CONTENIDO'] = $this->html;
        $this->html = NULL;
        $this->mostrar();
    }

    private function mostrar() {
        $this->set_data($this->dict);
        $this->file = "../site_media/html/template.html";
        $this->render_template();
        print $this->html;
    }

}
```

Finalmente, modificaremos nuestro archivo de pruebas, para simular la implementación de lo anterior:

```
require_once('../core/settings.php');
require_once('../core/db_object.php');
require_once("../core/render_template.php");
require_once("../core/render_data.php");
require_once('models/boton.php');
require_once('views/boton.php');

$boton = new Boton();
$data = $boton->get();

$view = new BotonView();
$view->get_boton($data);
```

El resultado final, será el *template* y la tabla del listado de botones, renderizada:

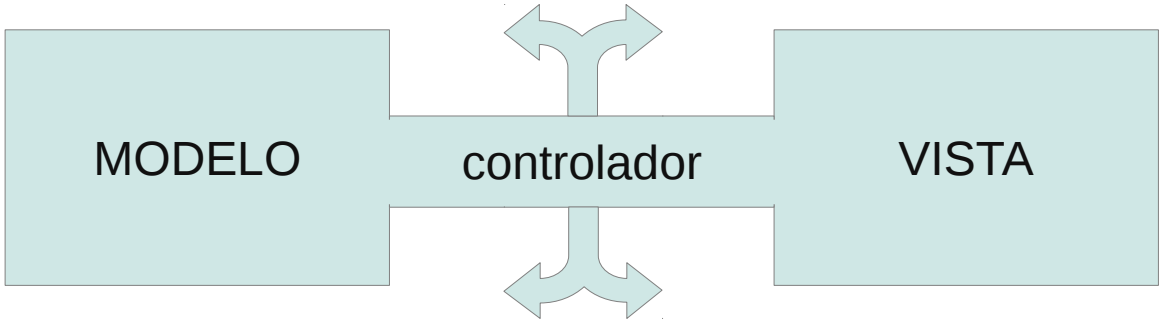
```
<!doctype html>
<head>
    <meta charset="utf-8">
    <title>Administracion de Camisas</title>
</head>

<body>
    <header>
        <h1>MyApp: Camisas</h1>
        <nav>

        </nav>
    </header>
    <section>
        <header>
            <h2>Listado de Botones</h2>
        </header>
```

```
        <table>
            <tr>
                <th>Material</th>
                <th>Color</th>
                <th>Ojales</th>
            </tr>

            <tr>
                <td>nylon</td>
                <td>verde</td>
                <td>4</td>
            </tr>

            <tr>
                <td>madera</td>
                <td>caoba</td>
                <td>0</td>
            </tr>

            <tr>
                <td>madera</td>
                <td>verde</td>
                <td>0</td>
            </tr>

            <tr>
                <td>metal</td>
                <td>oro</td>
                <td>0</td>
            </tr>

        </table>

    </section>
</body>

</html>
```

El controlador: el alma de MVC

En MVC, el controlador es parte esencial de la arquitectura. El controlador es una especie de “intermediario” entre el modelo y la vista. Es quien los conecta.

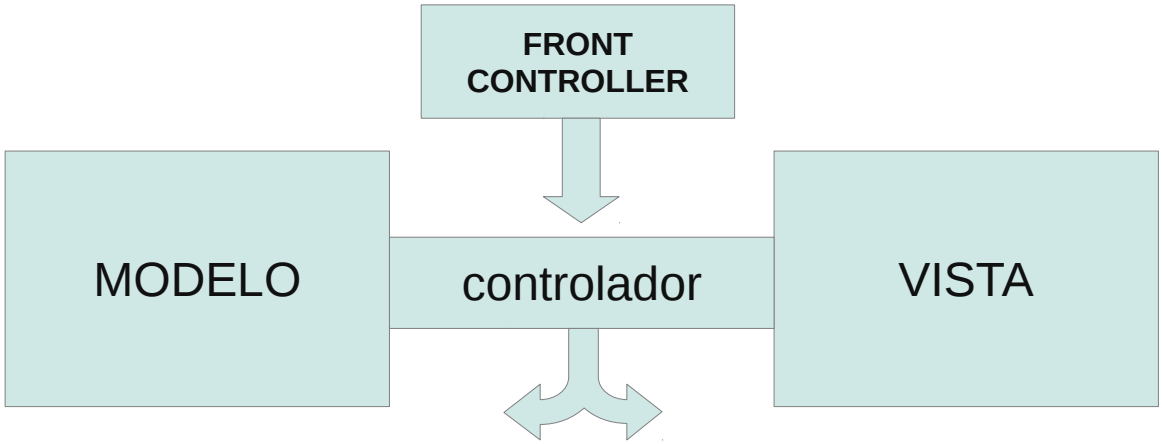
Podemos imaginar al controlador, como un túnel conector entre el modelo y la vista:



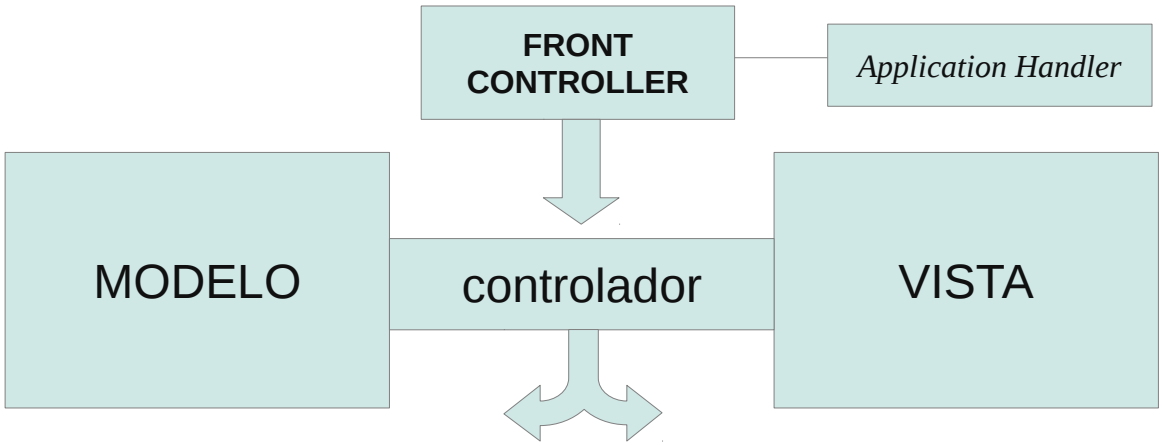
Cada módulo, contendrá entonces, un controlador para conectar cada uno de los modelos del módulo, con la vista correspondiente.

```
myapp/
├── core
├── module
│   ├── models
│   │   ├── boton.php
│   │   └── camisa.php
│   └── views
│   │   ├── boton.php
│   │   └── camisa.php
│   └── controllers
│       ├── boton.php
│       └── camisa.php
└── site_media
```

A la vez, el usuario llegará a cada controlador, por media de un controlador general, a nivel del núcleo de la aplicación:



Y este controlador general, manejará las peticiones del usuario, a través de un *handler* (*Application Handler*):



Front Controller: controlando la aplicación a nivel del core

El ***Front Controller*** será el encargado de manejar las peticiones del usuario a nivel de la aplicación por medio de un ***Application Handler*** y hacer la llamada al controlador correspondiente, el que dependerá, de cuál haya sido la petición del usuario.

Toda la aplicación, entonces, se manejará desde el *Front Controller*, significando este hecho, que cada petición del usuario, será recibida, en primer término, por este controlador.

Para ello, el ***Front Controller*****, necesitará conocer:**

- A qué módulo de la aplicación desea acceder el usuario
- A qué modelo hace referencia
- Qué recurso se está solicitando
- y, si se le están enviando parámetros (argumentos) a dicho recurso

Todos estos datos, *Front Controller* los recibirá invocando al ***Application Handler*** y éste,

a la vez, los obtendrá analizando la URI ingresada por el usuario.

> *El **Application Handler** obtiene el módulo, el modelo, el recurso y sus argumentos, analizando la URI a través de la cual, ha ingresado el usuario.*

Para que el *Application Handler*, pueda realizar un análisis simple de la URI, ésta, en consecuencia, también debería simplificarse. Para ello, utilizaremos las llamadas ***Friendly URL*** (o URL amigables).

Configurando URLs amigables para nuestra aplicación

Hasta ahora, accedíamos a nuestra aplicación, realizando las peticiones mediante HTTP GET. Con las URL amigables, reemplazamos dicho método, tomando el valor de cada argumento -anteriormente enviado por HTTP GET- y convirtiéndolo en parte de la URL. Veámoslo con un ejemplo.

Por HTTP GET, pasaríamos los argumentos así:

```
archivo.php?modulo=camisas&modelo=boton&recurso=get&boton_id=32
```

Pero ahora, convertiremos el valor de cada argumento, en parte de la URI:

```
/camisas/boton/get/32
```

Es decir, que todas nuestras URIs, ahora estarán conformadas siguiendo el patrón:

```
/modulo/modelo/recurso/argumento1/argumento2/argumento9/
```

Para poder trabajar de esta forma, vamos a necesitar:

1. Configurar Apache, habilitando el módulo `rewrite`
2. Modificar nuestro *virtual host*, permitiendo sustituir las URL
3. Crear un archivo .htaccess

Configuración de Apache

La configuración de Apache es sumamente sencilla. Lo único que tendremos que hacer es habilitar el módulo `rewrite`:

```
sudo a2enmod rewrite
```

Y luego, reiniciar Apache para que el cambio surta efecto:

```
sudo service apache2 restart
```

Modificar el VirtualHost

Habrá que editar el archivo VirtualHost de nuestro Web Site. Por defecto, en distribuciones basadas en Debian, este archivo se encuentra en **`/etc/apache2/sites-available/default`**
En otras distribuciones, este archivo puede localizarse dentro de `/etc/httpd/conf.d/`

Para modificarlo, dentro de `<VirtualHost>` tendremos que localizar (o crear si no existe), las etiquetas `<Directory />` y `</Directory>` y allí dentro, colocar el valor de la variable `AllowOverride` en `All`, para permitir que las URL sean sustituidas:

```
<VirtualHost>
    # directivas....
    <Directory />
        AllowOverride All
    </Directory>
</VirtualHost>
```

A continuación, recargaremos Apache:

```
sudo service apache2 reload
```

Creando el archivo .htaccess

El archivo .htaccess lo crearemos en el directorio principal de la aplicación. En él, nos encargaremos de habilitar la re-escritura con el módulo rewrite:

```
RewriteEngine On
```

E indicarle que todo tipo de peticiones, deberán ser devueltas a un archivo llamado init_core.php que crearemos a continuación:

```
RewriteRule ^ init_core.php
```

Creando un Application Handler

Nuestro Application Handler, será una clase que tendrá a su cargo:

- Analizar la URI
- Setear un array con la petición del usuario (módulo, modelo y recurso)

- Setear un array con todos los argumentos que le hayan sido pasados al recurso

Para ello, lo que tendremos que hacer es, declarar métodos que se encarguen de:

1. Obtener la URI
2. Dividirla en fragmentos
3. Asignar cada fragmento a la petición módulo/modelo/recurso correspondiente
4. Almacenar los argumentos

La clase AppHandler

```
# Archivo: /app/core/app_handler.php
class AppHandler {

    private $petition_names;
    private $uri;
    private $uri_data;
    public $peticiones = array();
    public $args = array();

    function __construct() {
        $this->petition_names = array("modulo", "modelo", "recurso");
        $this->uri = $this->get_uri();
        $this->uri_data = $this->set_peticiones();
        $this->peticiones = $this->get_peticiones();
        $this->args = $this->get_parametros();
    }

    private function get_uri() {
        return str_replace(APP_PATH, "", $_SERVER['REQUEST_URI']);
    }

    private function set_peticiones() {
        return explode("/", $this->uri);
    }

    private function get_long() {
        $partes = count($this->uri_data);
        return ($partes > 3) ? 3 : $partes;
    }

    private function get_peticiones() {
        $long = $this->get_long();
        $arr = array_combine(array_slice($this->petition_names, 0, $long),
                             array_slice($this->uri_data, 0, $long));

        foreach($arr as $clave=>$valor) {
            if(empty($valor)) unset($arr[$clave]);
        }

        return $arr;
    }

    private function get_parametros() {
        return array_slice($this->uri_data,
                           $this->get_long(),
                           count($this->uri_data));
    }
```

```
}
```

Analicemos la clase AppHandler:

El método `get_uri()` retorna el valor de la clave `REQUEST_URI` del array superglobal `$_SERVER`. Lo que hace es quitar el contenido de la constante `APP_DIR` (declarada en el settings) de la URI. Esto es necesario, si nuestra app, no está corriendo en el directorio raíz de nuestro servidor. Entonces, declaramos el path hasta el directorio de nuestra aplicación, en la constante `APP_DIR` y obtendremos la parte de la URI que nos interesa. El retorno de este método, será almacenado desde el constructor en la propiedad `$uri`.

```
    private function get_uri() {
        return str_replace(APP_PATH, "", $_SERVER['REQUEST_URI']);
    }
```

El método `set_peticiones()`, se encarga de dividir la URI anterior, tomando como referencia, la barra diagonal. Los valores retornados por este método, serán almacenados desde el constructor, en la propiedad `$uri_data`.

```
    private function set_peticiones() {
        return explode("/", $this->uri);
    }
```

El método `get_peticiones()` realiza un análisis más complejo de los resultados almacenados en la propiedad `$uri_data`. Primero, llama get_long() para calcular la longitud de las peticiones. Esto es:

Si hay 3 valores, estaremos en condiciones de asignar un valor a módulo, otro a modelo y el tercero, a recurso. Si hay más, será señal de que estamos recibiendo argumentos. Pero si hay menos, la longitud será la cantidad de elementos reales de `uri_data`:

```
    private function get_long() {
        $partes = count($this->uri_data);
        return ($partes > 3) ? 3 : $partes;
    }
```

`get_peticiones`, necesita de `get_long`, para poder extraer de uri_`data`, la cantidad correspondiente a los tres primeros elementos (o si hay menos, los que tenga). Mediante array_`slice`, extrae el módulo, modelo y recurso de uri_`data` (son los tres primeros elementos) y hace lo propio con el array que define los nombres (claves) para estos tres pedidos.

```
        $arr = array_combine(array_slice($this->petition_names, 0, $long),
                             array_slice($this->uri_data, 0, $long));
    }
```

Finalmente, recorre el array resultante en busca de elementos vacíos. Elementos vacíos

pueden suceder cuando la URI real, finalizaba en una "/". Si encuentra elementos vacíos, elimina la clave del array resultante y finalmente, retorna ese array. El resultado, será almacenado desde el constructor, en la propiedad pública `$peticiones.`

```
        foreach($arr as $clave=>$valor) {
            if(empty($valor)) unset($arr[$clave]);
        }

        return $arr;
```

Finalmente, el método `get_parametros` se encargará de extraer de `uri_data`, aquellos elementos "sobrantes" (es decir, cualquier elemento que se encuentre del tercero). Esos elementos, serán los argumentos que almacene la propiedad pública `args` desde el constructor:

```
    private function get_parametros() {
        return array_slice($this->uri_data,
                           $this->get_long(),
                           count($this->uri_data));
    }
```

Conclusión

Podemos deducir entonces, que la clase `AppHandler` será la encargada de analizar toda la URI, colocando en disposición pública un array asociativo con las peticiones módulo-modelo-recurso y otro, con todos los argumentos.

Esta clase, entonces, será utilizada por el `FronController,` quien solo la instanciará y, el método constructor de ésta, le proveerá a `FrontController` de un objeto AppHandler, con dos propiedades públicas `AppHandler->peticiones` y `AppHandler->args`, para que `FrontController` se encargue del resto, es decir, instancie al controlador correspondiente, enviándole la petición (recurso) con todos los argumentos que este recurso necesite.

La clase FrontController

Ahora sí, ya estamos en condiciones de crear nuestro **controlador general de la aplicación**.

La clase FrontController se encargará entonces, de:

- Llamar a AppHandler para obtener las peticiones solicitadas por el usuario y los argumentos enviados
- Buscar el controlador necesario e importarlo
- Llamar al controlador y enviarle la petición (recurso) y los argumentos

Veamos como será la clase FrontController:

```
# Archivo: /app/core/app_handler.php

class FrontController {

    private static $app_data;
    public static $error = NULL;

    public static function run() {
        self::$app_data = new AppHandler();
        if(!self::call_controller()) {
            self::$error = "El recurso solicitado no se encuentra disponible";
        }
    }

    private static function call_controller() {
        extract(self::$app_data->peticiones);
        if(isset($modulo) && isset($modelo) && isset($recurso)) {
            require_once("$modulo/controllers/$modelo.php");
            $controller = Helper::set_controller_name($modelo);
            $method = Helper::set_method_name($recurso);
            $c = new $controller($method, self::$app_data->args);
            return True;
        }
    }

}
```

Cuando `FrontControll` deba ser “activado”, será llamado de forma estática, su método `run.` Por ejemplo:

```
FronController::run();
```

Será el encargado de manejar todas las peticiones a nivel de la aplicación.

Para obtener las peticiones y argumentos, `FrontController` simplemente crea un objeto AppHandler que será almacenado en una propiedad estática (privada) `$app_data` (línea 9 de `app_handler.php`)

```
self::$app_data = new AppHandler();
```

Para buscar e importar el controlador necesario, primero verificará que las peticiones módulo, modelo y recurso se encuentren seteadas en el objeto `AppHandler` y de ser así, incluirá el archivo del controlador:

```
        # convierte en variables temporales todas las peticiones
        extract(self::$app_data->peticiones);
        # si modulo, modelo y recurso han sido seteadas...
        if(isset($modulo) && isset($modelo) && isset($recurso)) {
            # Importa el archivo del controlador
            require_once("$modulo/controllers/$modelo.php");
```

Si nos ponemos detallistas, podremos observar que FrontController, de manera indirecta

nos está forzando a mantener una estructura de directorios y una arquitectura, conforme lo que hemos visto hasta ahora sobre MVC.

Finalmente, para llamar al controlador y pasarle el recurso (petición) y sus argumentos, hace lo siguiente:

Se vale de un Helper para obtener el nombre adecuado del controlador. El mismo debe ser el nombre de la petición, sin guiones medios, con formato CamelCase seguido de la palabra Controller. De esto se encarga el método Helper::set_controller_name().

```
$controller = Helper::set_controller_name($modelo);
```

Luego, hace lo propio con el recurso. Para obtener el nombre del método-controlador del recurso, se vale de otro helper, el cual considerará las reglas para establecer dicho nombre: será el nombre del recurso, reemplazando guiones medios por bajos, todo en minúsculas y seguido del sufijo _controller.

```
$method = Helper::set_method_name($recurso);
```

Finalmente, instancia al controlador, pasándole dos parámetros: el recurso y los argumentos:

```
$c = new $controller($method, self::$app_data->args);
```

Conclusión

Si observamos el código anterior, concluiremos en que `FrontController`, nos ha guiado el camino para crear nuestros controladores. Los mismos deberá, contar sí o sí, con:

- Un método constructor preparado para recibir dos parámetros: el recurso (nombre del método que deberá invocar) y los argumentos (parámetros que deberá pasarle a ese método)
- Un método para cada recurso, donde el nombre del mismo, deberá ser `nombre_del_recurso_controller`
- cada uno de esos métodos, deberá estar preparado para recibir una cantidad incierta de parámetros

Y sí. Si estás pensando en `call_user_func_array()` y `func_get_args()`, estás en el camino correcto :)

> *Vale aclarar además, que **todos los archivos que integran el core** de la aplicación que fuimos creando a lo largo de este taller, **son reutilizables en cualquier tipo de aplicación modular MVC**. Esto significa, que para crear aplicaciones MVC modulares, podrás reutilizar este core y solo concentrarte en crear los módulos, con sus correspondientes modelos, vistas y controladores.*
>
> *Es decir, que **todos estos archivos del core**, no son meros ejemplos que deban ser interpretados y trasladados a los objetivos de tu aplicación, sino que por el contrario, **son el núcleo de tu aplicación, sea cual fuere su objetivo.***

A continuación, veremos como crear el controlador para nuestros modelos.

Creando controladores para nuestros modelos

Crear el controlador para nuestros modelos, será una tarea sumamente simple.

Por un lado, ya conocemos **qué es lo que debe hacer un método-controlador**:

- Instanciar al modelo
- Modificar sus propiedades (cuando sea necesario)
- Llamar a uno de sus métodos (el cual, nos retornará algún dato)
- Enviar los datos retornados por el modelo, a la vista

Por otro lado, nuestro `FrontController`, nos ha definido las características que tendrán que tener nuestros controladores:

- El nombre de la clase será el nombre del modelo, seguido del sufijo `Controller:`

```
class ModeloController { }
```

- Deberá tener un método constructor, preparado para recibir dos parámetros. El primer parámetro, será el nombre del recurso (método-controlador) y el segundo, un array de argumentos para enviar a ese recurso:

```
class ModeloController {
    public function __construct($recurso='', $argumentos=array() { }
}
```

- La clase `ModeloController`, deberá contar entonces, con un método por cada recurso, donde el nombre de éste se conforme por el nombre del recurso en minúsculas, palabras separadas por guiones medios, sucedido del sufijo `_controller`:

```
class ModeloController {

```

```
    public function __construct($recurso='', $argumentos=array() { }

    protected function nombre_del_recurso_controller() { }
}
```

- El método constructor, deberá ser quien invoque al método-controlador pasándole los argumentos necesarios. Para ello, utilizará la función nativa `call_user_func_array`:

```
class ModeloController {

    public function __construct($recurso='', $argumentos=array() {
        call_user_func_array(array($this, $recurso), $argumentos);
    }

    protected function nombre_del_recurso_controller() { }
}
```

- Y a la vez, los métodos-controladores que requieran parámetros, deberán estar preparados, para recibir una cantidad de argumentos incierta:

```
class ModeloController {

    public function __construct($recurso='', $argumentos=array() {
        call_user_func_array(array($this, $recurso), $argumentos);
    }

    protected function nombre_del_recurso_controller() {
        $argumentos = func_get_args();
    }
}
```

- Y finalmente, nuestros controladores, deberán almacenarse en un directorio `controllers` dentro del módulo al cual pertenezcan. El nombre, deberá ser el mismo que el delo modelo.

Características extras que debemos tener en cuenta para crear nuestros controladores son:

- Sabemos que todos los métodos-controlares (o casi todos), necesitarán instanciar al modelo. Pues entonces, lo haremos en el método constructor, asignando el objeto a una propiedad:

```
class ModeloController {

    public function __construct($recurso='', $argumentos=array() {
        call_user_func_array(array($this, $recurso), $argumentos);
        $this->modelo = new Modelo();
    }

    protected function nombre_del_recurso_controller() {
        $argumentos = func_get_args();
    }
```

```
}
```

- También sabemos que todos los métodos-controlares, deberán instanciar a la vista. Así que haremos lo propio:

```
class ModeloController {

    public function __construct($recurso='', $argumentos=array() {
        call_user_func_array(array($this, $recurso), $argumentos);
        $this->modelo = new Modelo();
        $this->view = new ModeloView();
    }

    protected function nombre_del_recurso_controller() {
        $argumentos = func_get_args();
    }
}
```

- Ya que el controlador será el único autorizado a conectar el modelo con la vista, será a la vez, el único encargado de importar los archivos correspondientes a su modelo y su vista:

```
require_once('modulo/models/modelo.php');
require_once('modulo/views/modelo.php');

class ModeloController {

    public function __construct($recurso='', $argumentos=array() {
        call_user_func_array(array($this, $recurso), $argumentos);
        $this->modelo = new Modelo();
        $this->view = new ModeloView();
    }

    protected function nombre_del_recurso_controller() {
        $argumentos = func_get_args();
    }
}
```

Lo anterior, entonces, será nuestro ***template* para crear controladores**. Entonces, con este *template*, crearemos -ahora sí-, nuestro controlador `BotonController`:

```
# Archivo: /app/module/controllers/boton.php

require_once('module/models/boton.php');
require_once('module/views/boton.php');

class BotonController {

    public function __construct($recurso='', $args=array()) {
        $this->boton = new Boton();
        $this->view = new BotonView();
        call_user_func_array(array($this, $recurso), $args);
    }

    private function get_controller() {
```

```
        $args = func_get_args();
        $this->boton->boton_id = (count($args) > 0) ? (int)$args[0] : 0;
        $data = $this->boton->get();
        $this->view->get_boton($data);
    }

}
```

Notas adicionales sobre el ejemplo

A fin de hacer más entretenido|productivo|real nuestra aplicación, haremos una pequeña modificación en el query del método get del modelo Boton, de forma tal que pueda traer un solo resultado que coincida con la ID seteada o todos, en caso de que la ID sea igual a cero. El modelo Boton, entonces, se verá así:

```
class Boton {

    public $material = NULL;
    public $color = NULL;
    public $ojales = NULL;
    public $boton_id = 0;


    public function save() {
        $sql = "INSERT INTO boton (material, color, ojales) VALUES (?, ?, ?)";

        $data = array("ssi",
                      "{$this->material}",
                      "{$this->color}",
                      "{$this->ojales}");

        return DBObject::ejecutar($sql, $data);
    }


    public function get() {
        $sql = "SELECT material, color, ojales FROM boton WHERE boton_id ";
        $sql .= ($this->boton_id > 0) ? "= ?" : "> ?";

        $data = array("i", "{$this->boton_id}");

        $fields = array("Material"=>"",
                        "Color"=>"",
                        "Ojales"=>"");

        DBObject::ejecutar($sql, $data, $fields);
        return DBObject::$results;
    }
}
```

Agregamos una línea que completa el *query*, diciendo que:

Si `boton_id` es mayor que cero, la cláusula `WHERE` se completará con `= ?`, sino, se completará con `> ?`

Completando la aplicación

Ahora sí, es hora de ir “redondeando” nuestra aplicación. En principio, haremos lo siguiente:

1. Mover el archivo `settings.php` al directorio raíz de la aplicación
2. Crear en el mismo directorio, un archivo llamado `init_core.php` que -como su nombre lo indica- se encargará de inicializar el núcleo de la aplicación.

De esta forma, la estructura de nuestra aplicación, deberá verse así:

```
app/
├── core/
│   ├── app_handler.php
│   ├── db_object.php
│   ├── front_controller.php
│   ├── helper.php
│   ├── render_data.php
│   └── render_template.php
├── init_core.php
├── module/
│   ├── controllers/
│   │   └── boton.php
│   ├── models/
│   │   └── boton.php
│   └── views/
│       └── boton.php
├── settings.php
└── site_media/
    └── html/
        ├── lista_botones.html
        └── template.html
```

A continuación, haremos algunas modificaciones a nuestro archivo settings.php

Por un lado, agregaremos 4 constantes, encargadas de definir:

- El path donde corre nuestra aplicación. Esto es, desde el directorio raíz del servidor, hasta el directorio raíz de la aplicación inclusive.
- El directorio que almacenará los archivos estáticos. Si es otro servidor, deberá definirse antecediendo el protocolo correspondiente. Por ejemplo: `http://xxx.xxx.xx.xx/app/`
- El directorio que almacena los archivos HTML dentro del directorio estático.
- El directorio del núcleo (en nuestro caso, `core/`)

```
const APP_PATH = "/mvcapp/";
const STATIC_PATH = "site_media/";
const STATIC_DIR = "html/";
const CORE_DIR = "core/";
```

Por otro lado, modificaremos -en tiempo de ejecución- el `include_path` del `php.ini`, a fin de poder incluir los archivos de forma más cómoda. Asignaremos el valor de la constante superglobal `__DIR__` a esta directiva:

```
ini_set("include_path", __DIR__);
```

Finalmente, agregaremos un pequeño script, para que nos muestre todos los errores, avisos y advertencias de PHP, siempre y cuando, no estemos en producción (si haz realizado el curso PHP para Principiantes o haz leído el libro del curso, el script que utilizaremos es el mismo que usamos en el ejemplo del capítulo “Tratamiento y Control de Errores”):

```
const PRODUCCION = False;

if(!PRODUCCION) {
    ini_set('error_reporting', E_ALL | E_NOTICE | E_STRICT);
    ini_set('display_errors', '1');
    ini_set('track_errors', 'On');
} else {
    ini_set('display_errors', '0');
}
```

Inicializador del núcleo

Finalmente, vamos a crear nuestro archivo inicializador del núcleo, `init_core.php`

Este archivo cumplirá dos funciones:

1) Incluir (importar) todos los archivos del núcleo en el orden preciso
2) Iniciar el `FronController`

```
require_once('settings.php');
require_once(CORE_DIR . 'app_handler.php');
require_once(CORE_DIR . 'front_controller.php');
require_once(CORE_DIR . 'db_object.php');
require_once(CORE_DIR . 'helper.php');
require_once(CORE_DIR . 'render_template.php');
require_once(CORE_DIR . 'render_data.php');

FrontController::run();
```

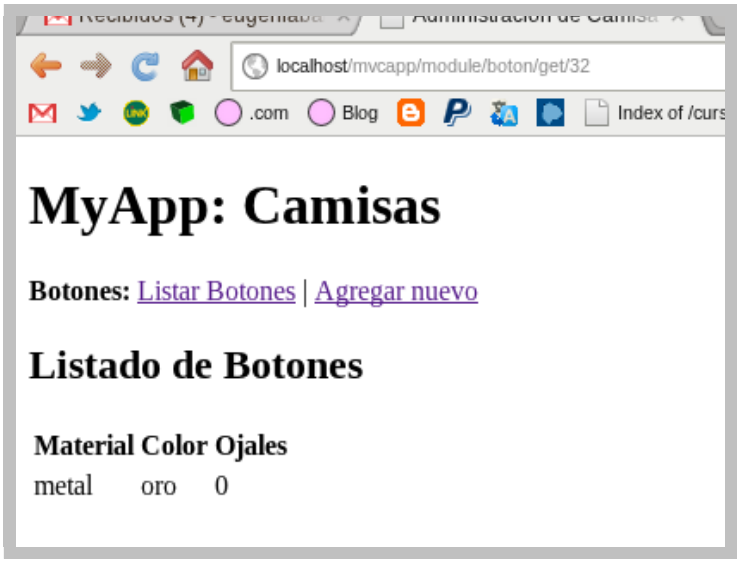
Para este ejemplo, agregaremos una tercer función que en realidad, es opcional. Ésta, será la de verificar si `FrontController` ha fallado, para imprimir un mensaje que lo indique.

```
if(!is_null(FrontController::$error)) {
    print FrontController::$error;
}
```

¡Ya tenemos nuestra aplicación funcionando!

Para probarla, podremos ingresar en nuestro navegador la URI correspondiente:
`http://dominio/aplicacion/module/boton/get`

O podremos también, pasar una ID de botón como argumento:
`http://dominio/aplicacion/module/boton/get/32`



Anexo I: creando una vista por defecto para la aplicación

Es posible crear una vista general, que sea mostrada al usuario cada vez que un recurso no sea especificado. Existen varios métodos mediante los cuales podremos lograrlo.

El más simple de todos, es crear una clase FrontView a nivel del core. Para ello necesitaremos:

1. Una plantilla HTML
2. Una clase FrontView que herede de RenderTemplate
3. Agregar un método de comprobación a FrontController que se encargue de llamar a nuestra nueva clase

Veamos como sería en la práctica.

Plantilla HTML (`site_media/html/default_template.html`):

```
<!doctype html>
<head>
    <meta charset="utf-8">
    <title>MVCApp</title>
</head>

<body>
    <header>
        <h1>MVC App</h1>
    </header>
    <section>
        <header>
            <h2>Módulos Disponibles en la aplicación</h2>
        </header>
        <section>
            <p><b>{FRONT-CONTROLLER-MESSAGE}</b><br/>
            Por favor, seleccione la opción de su interés</p>
            <h3>Módulo de Camisas</h3>
            <nav>
                <b>Botones:</b>
                    <a href="{APP_PATH}module/boton/get">Listar Botones<a>
            </nav>
        </section>
    </section>
</body>

</html>
```

La clase FrontView (`core/front_view.php`):

```
class FrontView extends RenderData {

    public function __construct($msg='') {
        $this->dict = array("APP_PATH" => APP_PATH,
                            "FRONT-CONTROLLER-MESSAGE" => $msg);
```

```
    }

    public function show_default_view() {
        $this->set_data($this->dict);
        $this->file = STATIC_PATH . STATIC_DIR . "default_template.html";
        $this->render_template();
        print $this->html;
    }
}
```

Método de validación en FrontController:

Método para setear el mensaje que será transmitido a la vista:

```
    private static function set_error() {
        if(isset(self::$app_data->peticiones['modulo'])) {
            self::$error = "El recurso solicitado no se encuentra disponible";
        }
    }
```

Este método, seteará siempre el mismo error, excepto cuando el usuario acceda a la carpeta raíz (index) de la aplicación.

Método que ejecuta un comando “mostrar vista por defecto”:

```
    private static function execute_command() {
        $command = new FrontView(self::$error);
        $command->show_default_view();
    }
```

Ambos métodos, implicarán modificar el método `run` de `FrontController`:

```
        if(!self::call_controller()) {
            self::set_error();
            self::execute_command();
        }
```

Eliminar la siguiente validación del `init_core.php`

```
if(!is_null(FrontController::$error)) {
    print FrontController::$error;
}
```

Y lógicamente, agregar la importación de `front_view.php` en el `init_core.php`

```
require_once(CORE_DIR . 'front_view.php');
```