Manual de MVC: (3) Los objetos View

En el capítulo anterior, vimos como identificar los diferentes tipos de sustituciones que desde las vistas de los modelos, en MVC, pueden realizarse. En esta entrega veremos como crear los objetos View de cada modelo, los cuáles implementarán todo lo aprendido en el capítulo anterior.

Escrito por: **Eugenia Bahit** (Arquitecta GLAMP & Agile Coach)

Eugenia es **Arquitecta de Software**, **docente** instructora de tecnologías **GLAMP** (GNU/Linux, Apache, MySQL, Python y PHP) y **Agile coach** (UTN) especializada en Scrum y eXtreme Programming. Miembro de la **Free Software Foundation** e integrante del equipo de **Debian Hackers**.

Webs:
Cursos de programación a Distancia: www.cursosdeprogramacionadistancia.com
Web personal: www.eugeniabahit.com

Redes sociales:
Twitter / Identi.ca: **@eugeniabahit**

En el capítulo anterior, estuvimos viendo ejemplos de los diferentes algoritmos que pueden utilizarse tanto en Python como en PHP, para realizar sustituciones estáticas y dinámicas (*render*) en los archivos HTML.

Dichos algoritmos, formarán parte de los métodos de cada una de las vistas, pudiendo además, crearse objetos `View` a nivel del `core`, para ser reutilizados en las vistas de cada modelo.

> *En MVC, cada modelo debe contar con una vista (objeto ModelView)*

En principio, debemos saber que para cada modelo debe haber un objeto vista (objeto View). Sin embargo, no todos los recursos, deberán tener un método en la vista (pero sí, en el controlador del modelo, que veremos más adelante).

Cuando en MVC se poseen las funcionalidades propias de un ABM, como agregar, modificar y eliminar un objeto, las vistas, deberán tener al menos, dos métodos: uno para mostrar el formulario de alta y otro, para mostrar el formulario de edición. En los

ABM, también son frecuentes otros dos métodos: uno para mostrar el listado de una colección de objetos determinada y otro, para visualizar un objeto puntual. Por supuesto, la cantidad de métodos de una vista, depende solo y exclusivamente de los requerimientos gráficos de cada aplicación. No obstante, el objetivo de esta entrega, es ver como crear los objetos View. Luego, la cantidad de métodos a desarrollar, dependerá del lector y de los requerimientos de su GUI.

Los métodos de la vista, serán invocados por el controlador del modelo. Éste, será quien entregue los datos a la vista, mientras que la última, será la encargada de realizar las sustituciones pertinentes e imprimir el resultado de las mismas en pantalla. Básicamente, la secuencia podría describirse como la siguiente:

1. El usuario solicita un recurso a través del navegador
2. El `FrontController` (descrito en la edición #1), analiza la petición del usuario e instancia al controlador correspondiente (veremos controladores más adelante)
3. El constructor de cada controlador, es quien realiza una llamada de retorno a un método propio el cual se encargará de:
    - Instanciar al modelo correspondiente (aunque generalmente, esta instancia se realiza desde el constructor en la mayoría de los casos)
    - Realizar las modificaciones necesarias sobre el modelo (modificar propiedades, llamar a los métodos necesarios, etc.)
    - Entregar los datos retornados por el modelo a un método de la vista.
4. La vista, traerá las plantillas necesarias, para sustituir los datos que le han sido entregados por el controlador. Finalmente, imprimirá en pantalla, el resultado de dicha sustitución.

> *Antes de continuar, si aún no lo has hecho,* ***te recomiendo leer la segunda entrega del manual de MVC***[14] *y practicar con los ejemplos allí descritos, a fin de poder comprender mejor, todo lo expuesto en este artículo*

Cómo bien se comentó, debe existir una vista (objeto View) por cada modelo. El nombre de cada vista, generalmente, será el nombre del modelo, seguido de la palabra View. Por ejemplo, dados los modelos: `Vidrio`, `Marco` y `Ventana`, tendremos 3 vistas: `VidrioView`, `MarcoView` y `VentanaView`:

```
# en PHP
class VidrioView {
}
```

14 http://www.hdmagazine.org/?magazine=HackersAndDevelopers&num=2

```
# en Python
class VidrioView(object):
    pass
```

Suponiendo que para nuestro modelo `Vidrio` hayamos definido un recurso `agregar` (que deberá ser un método del controlador, como veremos más adelante), **en principio, debemos contar con la GUI** para este recurso (aquí no incluiremos aún la plantilla general, sino solo el contenido relativo a este recurso. La plantilla general se incluirá al final de este artículo):

```
<h3>Agregar un nuevo vidrio</h3>
<form method="POST" action="/mimodulo/vidrio/guardar">
<!-- notar que el nombre de los campos, debe coincidir con el nombre de las
propiedades del objeto, siempre que esto sea posible-->
    <label for="grosor">Grosor:</label><br/>
    <input type="text" name="grosor" id="grosor" size="3"/> mm<br/><br/>
    <label for="sup">Superficie:</label><br/>
    <input type="text" name="superficie" id="sup"/><br/><br/>
    <label for="color">Color:</label><br/>
    <input type="text" name="color" id="color"/><br/><br/>
    <input type="submit" value="Guardar"/>
</form>
```

Como podemos observar, la GUI siempre debe ser el primer paso en el proceso de desarrollo de las vistas.

En este ejemplo en particular, nos encontramos con que la GUI, no requiere de ninguna sustitución (ni estática ni dinámica). Entonces, lo único que necesitaremos tener en la lógica de nuestra vista, es un método que traiga dicha GUI y la muestre en pantalla. Este método, no necesariamente debe llamarse `agregar()`. Podría tener un nombre más descriptivo como por ejemplo, `mostrar_form_alta()`:

```
# PHP: Archivo /myapp/modulo/views/vidrio.php
class VidrioView {

    function __construct() {
    }

    function mostrar_form_alta() {
        $plantilla = file_get_contents("/ruta/a/agregar_vidrio.html");
        print $plantilla;
    }

}
```

```
# Python: Archivo /myapp/modulo/views/vidrio.py
class VidrioView(object):

    def __init__(self):
        pass
```

```
    def mostrar_form_alta(self):
        with open("/ruta/a/agregar_vidrio.html", "r") as archivo:
            plantilla = archivo.read()
        print plantilla
```

Cuando el formulario sea enviado, será solicitado el recurso `guardar` (ver atributo "`action`" del formulario HTML). Este recurso, será procesado por el controlador pero ¿qué haremos finalmente con el usuario? El controlador, podrá decidir que sería una buena idea, mostrarle el objeto creado, al usuario. En ese caso, deberá existir un recurso "`ver`" que también será manejado por el controlador (al igual que todos los recursos). El recurso `ver` será un método del controlador. Éste, se encargará de recuperar el objeto `Vidrio` recién creado y entregárselo a la vista para que haga lo suyo. Entonces ¿qué debemos tener primero? La GUI, igual que siempre:

```
<!-- GUI para PHP -->
<h3>Vidrio ID {vidrio_id}</h3>
<b>Grosor:</b> {grosor} mm<br/>
<b>Superficie:</b> {superficie}<br/>
<b>Color:</b> {color}
```

```
<!-- GUI para Python -->
<h3>Vidrio ID $vidrio_id</h3>
<b>Grosor:</b> $grosor mm<br/>
<b>Superficie:</b> $superficie<br/>
<b>Color:</b> $color
```

En este caso, la vista deberá contar con un método, que se encargue de realizar una **sustitución estática**. Para esto, el controlador, le deberá pasar un objeto `Vidrio` como parámetro. Ampliemos el ejemplo anterior:

```
# PHP: Archivo /myapp/modulo/views/vidrio.php
class VidrioView {

    function __construct() {
    }

    function mostrar_form_alta() {
        $plantilla = file_get_contents("/ruta/a/agregar_vidrio.html");
        print $plantilla;
    }

    function mostrar_objeto($objeto_vidrio) {
        # Traigo la plantilla
        $plantilla = file_get_contents("/ruta/a/ver_vidrio.html");

        # Creo el diccionario
        settype($objeto_vidrio, 'array');
        foreach($objeto_vidrio as $clave=>$valor) {
            $objeto_vidrio["{{$clave}}"] = $valor;
```

```
            unset($objeto_vidrio[$clave]);
        }

        # Realizo la sustitución
        $render = str_replace(array_keys($objeto_vidrio),
            array_values($objeto_vidrio), $plantilla);

        # Imprimo el resultado en pantalla
        print $render;
    }

}
```

```
# Python: Archivo /myapp/modulo/views/vidrio.py
from string import Template


class VidrioView(object):

    def __init__(self):
        pass

    def mostrar_form_alta(self):
        with open("/ruta/a/agregar_vidrio.html", "r") as archivo:
            plantilla = archivo.read()
        print plantilla

    def mostrar_objeto(self, objeto_vidrio):
        # Traigo la plantilla
        with open("/ruta/a/ver_vidrio.html", "r") as archivo:
            plantilla = archivo.read()

        # Creo el diccionario
        diccionario = vars(objeto_vidrio)

        # Realizo la sustitución
        render = Template(plantilla).safe_substitute(diccionario)

        # Retorno el resultado
        # para que application lo imprima en pantalla
        return render
```

Luego, si el resultado obtenido, se desea imprimir dentro de una plantilla general (plantilla de base), solo será cuestión de crear un método `show()` en una vista a nivel del `core`. La plantilla general, deberá contar con al menos un parámetro de sustitución para el contenido y será la vista de cada modelo, quien le pase a `show()` el valor de sustitución del mismo para finalmente, imprimir en pantalla el resultado retornado por `show()`. Aquí, una vez más, debemos contar previamente con la GUI:

```
<!-- PLANTILLA HTML PARA PHP -->
<!doctype html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <link rel="stylesheet" type="text/css" href="/static/css/style.css"/>
```

```
    <link rel="icon" href="/static/img/favicon.png" type="image/png"/>
    <title>{APP_TITLE}</title>
</head>
<body>
    <header>
        <h1>{APP_TITLE}: {MODULE_TITLE}</h1>
        <nav>
            <a href="/">{APP_TITLE}</a> &gt;
            <a href="/{MODULO}">{MODULE_TITLE}</a> &gt;
            <a href="/{MODULO}/{MODELO}">{MODEL_TITLE}</a> &gt;
            <b>{RESOURCE_TITLE}</b>
        </nav>
    </header>
    <section>
        <!-- aquí irá el contenido sustituido por la vista -->
        {CONTENIDO}
    </section>
    <footer>
        &copy; 2013 HDMagazine.org - GPL v3.0
    </footer>
</body>
</html>
```

```
<!-- PLANTILLA HTML PARA PYTHON -->
<!doctype html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <link rel="stylesheet" type="text/css" href="/static/css/style.css"/>
    <link rel="icon" href="/static/img/favicon.png" type="image/png"/>
    <title>$APP_TITLE</title>
</head>
<body>
    <header>
        <h1>$APP_TITLE: $MODULE_TITLE</h1>
        <nav>
            <a href="/">$APP_TITLE</a> &gt;
            <a href="/$MODULO">$MODULE_TITLE</a> &gt;
            <a href="/$MODULO/$MODELO">$MODEL_TITLE</a> &gt;
            <b>$RESOURCE_TITLE</b>
        </nav>
    </header>
    <section>
        <!-- aquí irá el contenido sustituido por la vista -->
        $CONTENIDO
    </section>
    <footer>
        &copy; 2013 HDMagazine.org - GPL v3.0
    </footer>
</body>
</html>
```

A nivel del core, se podrá tener una vista para la sustitución de la plantilla general. La misma, podrá realizarse mediante la llamada estática a un método de clase o a una función (fuera del contexto de una clase). Aquí, lo haremos en el contexto de una clase. Crear una clase con un método estático, es una buena alternativa para centralizar cualquier otro método relacionado con las vistas, directamente disponible desde el núcleo de la aplicación.

Por favor, notar que en los siguientes ejemplos, la constante `APP_TITLE` *se supone definida en un archivo* `settings`

```php
# Archivo: /myapp/core/view.php
class CoreView {

    public static function show($modulo, $modelo, $recurso, $render) {
        $plantilla = file_get_contents("/ruta/a/template.html");
        $diccionario = array(
            "{APP_TITLE}" => APP_TITLE,
            "{MODULE_TITLE}" => ucwords($modulo),
            "{MODULO}" => $modulo,
            "{MODELO}" => $modelo,
            "{MODEL_TITLE}" => ucwords($modelo),
            "{RESOURCE_TITLE}" => ucwords("$recurso $modelo"),
            "{CONTENIDO}" => $render
        );

        print str_replace(array_keys($diccionario), array_values($diccionario),
            $plantilla);
    }

}
```

```python
# Archivo: /myapp/core/view.py
class CoreView(object):

    def __init__(cls):
        pass

    def show(cls, modulo, modelo, recurso, render) {
        with open("/ruta/a/template.html", "r") as archivo:
            plantilla = archivo.read()

        diccionario = dict(
            APP_TITLE=APP_TITLE,
            MODULE_TITLE=modulo.title(),
            MODULO=modulo,
            MODELO=modelo,
            MODEL_TITLE=modelo.title(),
            RESOURCE_TITLE="%s %s" % (recurso.title(), modelo.title()),
            CONTENIDO=render
        )

        return Template(plantilla).safe_substitute(diccionario)
```

Luego, solo será necesario que cada una de las vistas, en vez de imprimir el resultado de la sustitución en pantalla, imprima el resultado de la llamada estática al método `show()` de `CoreView`:

```php
# PHP: Modificación del archivo /myapp/modulo/views/vidrio.php
    function mostrar($objeto_vidrio) {
        $plantilla = file_get_contents("/ruta/a/ver_vidrio.html");
        settype($objeto_vidrio, 'array');
```

```
        foreach($objeto_vidrio as $clave=>$valor) {
            $objeto_vidrio["{{$clave}}"] = $valor;
            unset($objeto_vidrio[$clave]);
        }
        $render = str_replace(array_keys($objeto_vidrio),
            array_values($objeto_vidrio), $plantilla);

        # Se imprime el resultado de la llamada estática al método show()
        print CoreView::show('modulo', 'vidrio', 'ver detalles de', $render);
    }
```

```
# Python: Modificación del archivo /myapp/modulo/views/vidrio.py
    def mostrar(self, objeto_vidrio):
        with open("/ruta/a/ver_vidrio.html", "r") as archivo:
            plantilla = archivo.read()
        diccionario = vars(objeto_vidrio)
        render = Template(plantilla).safe_substitute(diccionario)

        # Retorno el resultado de la llamada estática al método show()
        return CoreView().show('modulo', 'vidrio', 'ver detalles de', render)
```

Siempre digo a mis alumnos, que "el arte de las vistas en MVC, consiste en ***lograr convertir lo que se tiene, en lo que se necesita"***

Generalmente, siempre se tendrá un objeto y lo que se necesitará, será un diccionario formado por pares de clave-valor, donde los valores, no sean del tipo colección (es decir, sean de un tipo de datos simple). **Convertir un objeto en un diccionario, es la base de la lógica de las vistas en MVC.**

En la ***próxima entrega****, nos enfocaremos en los* ***controladores*** *de los modelos, para ir finalizando nuestro Manual de MVC en Python y PHP.*