Arquitecturas Web modulares con MVC en Python y PHP

MVC avanzado desde la óptica
del Arquitecto de Software y de
la del Programador

Eugenia Bahit

Registrado en SafeCreative. Nº de registro: NNNNNNNN
Impreso en España por Bubok Publishing S.L.

Una copia digital de este libro (copia no impresa) puede obtenerse de forma gratuita en http://mvc.eugeniabahit.com/

“El que sabe que es profundo se esfuerza por ser claro; el que quiere parecer profundo se esfuerza por ser oscuro.”

Federico Nietzsche

Contenidos

Capítulo I: 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 Configuración 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[1] se divide en 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 en los que se divide la Arquitectura de Software.

Estilo Arquitectónico

El estilo arquitectónico define un nivel general de la estructura del 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

1 Arquitectura de Software

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 [...]"*

(Traducción: 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 Composite.

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

Capítulo II: 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 sistemas informáticos manteniendo separados el diseño de los objetos (modelos) de la lógica negocio y sus interfaces gráficas (vistas), utilizando un conector intermediario (controlador) entre ambas.

Entendiendo el funcionamiento de MVC

En MVC, todo comienza con una petición del usuario, la cual es capturada y manejada por el controlador. En una aplicación Web, la petición del usuario podría ser (en idioma usuario), “agregar un nuevo elemento al sistema”.

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

Si bien el patrón arquitectónico se denomina

modelo-vista-controlador, no es el modelo quien actúa en primera instancia, sino el controlador.

En líneas generales, puede decirse que el proceso en MVC consiste en:

1. El usuario realiza una petición al controlador;
2. El controlador se comunica con el modelo y éste, le retorna -al controlador- la información solicitada;
3. Finalmente, el controlador le entrega dicha información a la vista y ésta, es quien finalmente, mostrará la información al usuario.

Módulos, modelos y recursos: manejando las peticiones del usuario

Podemos decir que los grandes sistemas informáticos, se encuentran divididos en módulos.

Un **módulo** puede describirse como la mayor fracción en la que un conjunto de objetos del sistema, puede ser agrupada.

Un módulo es aquella porción de un sistema informático que de forma individual, podría considerarse "una aplicación independiente con posibilidad de ser implementada en diversos

programas sin requerir modificaciones".

Si comparamos un sistema informático con una organización empresarial, podríamos decir que: *"los módulos son a un sistema informático lo que los departamentos son a una empresa"*.

Los módulos, a la vez se encuentran subdivididos en **modelos** (grupos de clases de objetos relacionados), **vistas** (interfaces gráficas -GUI- y lógica de negocio) y **controladores** los cuáles a la vez, proveen de ciertos **recursos** (métodos que dominarán la lógica de negocios de toda la aplicación).

De esta forma, un «*módulo de contabilidad*», podría contener un modelo Comprobante (destinado a crear facturas, notas de pedido, notas de crédito, etc.) y según la lógica de negocio, proveer de recursos como *agregar*, *guardar*, *modificar*, *ver*, *listar* y *eliminar*.

En MVC, toda petición del usuario, debe ser precisa y bien definida, indicando con exactitud, el módulo al cual el usuario se refiere, el modelo y el recurso que necesita.

Por este motivo, todas las URIs, deben guardar una estructura fija, del tipo `dominio/modulo/modelo/recurso`.

Si tuviese un módulo llamado "contabilidad" y

quisiera “generar” un nuevo “balance contable”, mi URL se vería así:

`http://myapp.net/`**`contabilidad/balance-contable/generar`**

Preparando la estructura de directorios de la aplicación

La organización del sistema de archivos y directorios en una aplicación MVC, cumple un papel clave, pues esta estructura forma parte de la arquitectura del sistema.

Todos los módulos del sistema, deben contar con su propio directorio, el cual contendrá a la vez, tres carpetas: `models`, `views` y `controllers`. A la vez, toda aplicación deberá contar con un directorio `core` (destinado a almacenar archivos del núcleo del sistema).

Una muy buena práctica es a la vez, agrupar todos los módulos dentro de un mismo directorio. Esto permitirá una mejor portabilidad de la aplicación y la instalación y desinstalación rápida de cualquiera de sus módulos.

Un árbol de directorios, podría verse -en principio- como el siguiente:

```
myapp
├── core
└── modules
    ├── contabilidad
    │   ├── controllers
    │   ├── models
    │   └── views
    ├── deposito
    │   ├── controllers
    │   ├── models
    │   └── views
    └── produccion
        ├── controllers
        ├── models
        └── views
```

En MVC, todos los archivos estáticos (html, css, javascript, imágenes y cualquier otro binario), deberán almacenarse en un directorio independiente, el cual incluso, pueda ser alojado en un servidor distinto al de la aplicación:

```
myapp
└── static
    ├── css
    ├── html
    ├── img
    ├── js
    └── multimedia
```

En cuanto a archivos fijos o estándar (aquellos que siempre estarán presentes en cualquier sistema), habrá que tener en cuenta que en el caso particular de **Python**, todos los directorios (excepto `static`), deberán poder ser tratados como paquetes. Por este motivo, se deberá crear un archivo **`__init__.py`** en cada directorio de la aplicación, incluyendo en su

carpeta raíz.

Luego, tanto en Python como en PHP, tendremos los siguientes archivos estándar en la raíz de la aplicación:

```
myapp
├── application.py|php
└── settings.py|php
```

El archivo *application*

Será el encargado de inicializar el motor de la aplicación. En el caso de PHP, será quien importe los principales archivos del núcleo y en el de Python, quien contenga la función `application()` de WSGI.

El archivo settings

Estará destinado a configurar todas aquellas variables/constantes globales necesarias para el "arranque" de la aplicación. Aprovecharemos la ocasión para ya dejarlas configuradas. Básicamente, lo mínimo que necesitaremos definir, será:

- Datos de acceso a la base de datos;
- Ruta absoluta de la aplicación;
- Ruta al directorio de archivos estáticos.

```
# Archivo: myapp/settings.py
DB_NAME = 'myapp'
DB_HOST = 'localhost'
DB_USER = 'root'
DB_PASS = 'secret'

APP_DIR = '/srv/myapp/'
STATIC_DIR = '/srv/myapp/static/'

# Archivo: myapp/settings.php
const DB_NAME = 'myapp';
const DB_HOST = 'localhost';
const DB_USER = 'root';
const DB_PASS = 'secret';

const APP_DIR = '/srv/myapp/';
const STATIC_DIR = '/srv/myapp/static/';
```

Ingeniería del Sistema

Anteriormente, comentamos que si bien el patrón se denomina modelo-vista-controlador, la ejecución de la aplicación se inicia en el controlador, quien llama al modelo y finalmente a la vista. Es decir, que a nivel lógica de negocio, el proceso sería una sucesión *controlador-modelo-vista*. Pero la ingeniería del sistema, es quien cumple con el proceso sucesivo exacto que hace honor al nombre del patrón, *modelo-vista-controlador*.

El sistema comienza a desarrollarse a partir de los **modelos**, siguiendo los pasos tradicionales enumerados a continuación:

1. Diseño de objetos en lenguaje natural[2];

2 Para conocer más sobre el diseño de objetos en lenguaje natural,

2. Desarrollo de las clases con sus propiedades y métodos correspondientes;
3. Mapeo de objetos y creación de la base de datos;

*Vale aclarar, que **antes de desarrollar los modelos, es necesario contar con un core básico**. Más adelante veremos como diseñar un core básico, reutilizable para cualquier aplicación MVC modular.*

Una vez desarrollados los modelos mínimos necesarios con los que deberá contar el sistema, deberá encontrarse preparada, la **interfaz gráfica** para cada modelo (GUI[3]) a fin de desarrollar en forma paralela, la **lógica de negocios** de dichas **vistas** y sus **controladores**.

Breve resumen del Capítulo II

La lógica relacional de los procesos en MVC consiste en:

1. El usuario solicita un recurso mediante la URI, quien conserva el formato: `/modulo/modelo/recurso`;
2. Dicha solicitud, llega al controlador del

se recomienda leer el libro "Teoría sintáctico-gramatical de objetos": http://www.bubok.es/libros/219288/

3 Graphical User Interface

modelo correspondiente;

3. El controlador, se conecta con el modelo, del cuál obtendrá la información necesaria para procesar el recurso;
4. La información obtenida desde el modelo, es entregada a la vista por e controlador;
5. La vista, sustituye la información recibida en la GUI y finalmente la muestra al usuario en pantalla de forma humanamente legible.

La estructura de directorios mínima inicial de toda aplicación MVC modular, debe ser como la siguiente:

```
myapp/
├── core/
├── modules/
│   └── modulo/
│       ├── controllers/
│       ├── models/
│       └── views/
├── static/
│   ├── css/
│   ├── html/
│   ├── img/
│   └── js/
├── application (file)
└── settings (file)
```

Y en el caso de Python, todos los directorios (excepto `static`) deberán además, contar con un archivo `__init__.py`

Capítulo III: Configurando un *Virtual Host* de Apache para aplicaciones MVC modulares

El *Virtual Host* de una aplicación MVC modular, no tiene demasiadas diferencias con el de un sitio Web estándar. Pero sí, las suficientes como para no poder funcionar de forma correcta, si las mismas no son tenidas en cuenta.

En principio se debe considerar que una aplicación Web modular, no es lo mismo que un simple sitio Web con ciertas funcionalidades. Una aplicación de esta magnitud, debe contar con su propio *Virtual Host* siendo altamente recomendable que el directorio raíz de la aplicación, sea a la vez, el directorio raíz para el *Virtual Host*.

El *Virtual Host* para una Aplicación Web en Python

En el caso de Python, la particularidad del *Virtual Host* es la declaración de la directiva `WSGIScriptAlias` apuntando hacia

`myapp/application.py`, pues este archivo será quien contenga la función `application` que encienderá "el motor" tras la llamada realizada por el módulo WSGI de Apache *(módulo recomendado por Python para la ejecución de éste sobre Apache)*.

Paquetes necesarios

Antes de crear el *Virtual Host*, es muy importante tener instalado Apache (lógicamente) y el módulo WSGI:

```
# apt-get install apache2 libapache2-mod-wsgi
```

Configuración del *Virtual Host*

En todo momento me referiré tanto al *host* de la aplicación como a la aplicación en sí misma, como `myapp`. Puedes modificar este nombre sin que esto, perjudique su arquitectura o el rendimiento de tu aplicación.

Primer paso: creación del archivo.-

```
# touch /etc/apache2/sites-available/myapp
```

Segundo paso: configuración del *Virtual Host*.-

```
<VirtualHost *:80>
     ServerName myapp.net

     DocumentRoot /srv/null
     WSGIScriptAlias / /srv/myapp/application.py
```

```
    Alias /static /srv/myapp/static
    Alias /favicon.ico /srv/icons/favicon.ico

    <Directory />
       AllowOverride None
    </Directory>

    ErrorLog /srv/logs/myapp/error.log
    CustomLog /srv/logs/myapp/access.log combined

</VirtualHost>
```

Explicación de lo anterior:

`<VirtualHost *:80>`

80 es el puerto por defecto para servir aplicaciones Web. Si tu aplicación correrá a través de otro puerto, modifica este valor.

`ServerName myapp.net`

Es el nombre del *host* propiamente dicho, a través del cual se accederá a tu aplicación. En ambientes de desarrollo locales, suele indicarse `.local` como extensión.

`DocumentRoot /srv/null`

El `DocumentRoot` de los *Virtual Host* de aplicaciones MVC modulares, es recomendable configurarlo en un directorio vacío con permisos de lectura, ya que la ejecución de toda la aplicación, pasará por un único archivo.

`WSGIScriptAlias / /srv/myapp/application.py`

Esta es la directiva principal. Con ella estamos informando que WSGI será el encargado de procesar las peticiones, redirigiendo todas y cada una de las solicitudes, al archivo `application.py` de nuestra aplicación.

`Alias /static /srv/myapp/static`

Como todas las solicitudes serán controladas por un único archivo, es necesario definir un alias para el acceso a archivos estáticos ya que de lo contrario, esto también debería manejarse desde el propio controlador.

`Alias /favicon.ico /srv/icons/favicon.ico`

El mismo caso anterior, se da con el famoso `favicon.ico` que de no estar presente acumulará errores 404 en los *logs* de Apache, consumiendo espacio y recursos sin sentido. Es preferible colocar un único `favicon.ico` para todos sus dominios en un directorio destinado a tal fin y simplemente definiendo un alias, nos ahorraremos sortear falsos errores en los *logs*.

`ErrorLog /srv/logs/myapp/error.log`

Configurar una ruta personalizada para los *logs* de errores en aplicaciones Python es fundamental e inevitable ya que cualquier error interno del servidor

(Error 500), en Python solo será visible leyendo el *log* de errores. Elige un directorio para guardar los *logs* (solo debes crear el directorio pues del archivo se encargará Apache) e indicca la ruta en esta directiva. Cuando tu aplicación genere errores, ejecuta el siguiente comando para verlos:

```
$ tail -12 /srv/logs/myapp/error.log
```

```
CustomLog /srv/logs/myapp/access.log combined
```

Podría parecer innecesario configurar una ruta personalizada para los *logs* de acceso. Pero por el contrario, cuando al revisar el *log* de errores la información resulte poco suficiente, controlar los pasos seguidos hasta producirse el error, será una forma de auditarlo, prevenirlo y corregirlo sin necesidad de intentar adivinarlo.

El *Virtual Host* para una Aplicación Web en PHP

En el caso de PHP, existen una única particularidad a contemplar en el *Virtual Host* y es la necesidad de habilitar la reescritura de las URL.

Paquetes necesarios

Antes de crear el *Virtual Host*, es muy importante tener instalado Apache (lógicamente), PHP y PHP CLI. Si no tienes ni Apache ni PHP instalado, con

solo instalar PHP (y PHP CLI) será suficiente ya que el paquete php5 tiene a apache2 como dependencia:

```
# apt-get install php5 php5-cli
```

Configuración del *Virtual Host*

En todo momento me referiré tanto al *host* de la aplicación como a la aplicación en sí misma, como `myapp`. Puedes modificar este nombre sin que esto, perjudique tu arquitectura o el rendimiento de tu aplicación.

Primer paso: creación del archivo.-

```
# touch /etc/apache2/sites-available/myapp
```

Segundo paso: configuración del *Virtual Host*.-

```
<VirtualHost *:80>
     ServerName myapp.net

     DocumentRoot /srv/myapp/
     Alias /favicon.ico /srv/icons/favicon.ico

     <Directory />
        Options -Indexes
        AllowOverride All
     </Directory>

     ErrorLog /srv/logs/myapp/error.log
     CustomLog /srv/logs/myapp/access.log combined

</VirtualHost>
```

Explicación de lo anterior:

`<VirtualHost *:80>`

80 es el puerto por defecto para servir aplicaciones Web. Si tu aplicación correrá a través de otro puerto, modifica este valor.

`ServerName myapp.net`

Es el nombre del *host* propiamente dicho, a través del cual se accederá a tu aplicación. En ambientes de desarrollo locales, suele indicarse `.local` como extensión.

`DocumentRoot /srv/myapp`

Es el directorio raíz de toda la aplicación.

`AllowOverride On`

Esta directiva es la que habilita la reescritura de las URL. Todo el proceso se complementará con un archivo `.htaccess` dentro del directorio raíz de la aplicación.

`Alias /favicon.ico /srv/icons/favicon.ico`

Cuando `favicon.ico` no está presente, acumulará errores 404 en los *logs* de Apache, consumiendo espacio y recursos sin sentido. Es preferible colocar un único `favicon.ico` para todos tus dominios en

un directorio destinado a tal fin y simplemente definiendo un alias, nos ahorraremos sortear falsos errores en los *logs*.

`ErrorLog` **`/srv/logs/myapp/error.log`**

Configurar una ruta personalizada para los *logs* de errores, en aplicaciones PHP no es tan importante como en Python. Sin embargo, nunca está demás tener este *log* a mano para obtener un mejor control de los errores de nuestra aplicación. Elige un directorio para guardar los *logs* (solo debes crear el directorio pues del archivo se encargará Apache) e indica la ruta en esta directiva. Cuando tu aplicación genere errores, podrás ejecutar además, el siguiente comando para verlos con mejor detalle:

```
$ tail -5 /srv/logs/myapp/error.log
```

`CustomLog` **`/srv/logs/myapp/access.log`** `combined`

Podría parecer innecesario configurar una ruta personalizada para los *logs* de acceso. Pero por el contrario, cuando al revisar el *log* de errores la información resulte poco suficiente, controlar los pasos seguidos hasta producirse el error, será una forma de auditarlo, prevenirlo y corregirlo sin necesidad de intentar adivinarlo.

En PHP, finalmente, será necesario habilitar el módulo `rewrite` de Apache:

```
# a2enmod rewrite
```

y definir un archivo `.htaccess` en la carpeta raíz de la aplicación, con las siguientes instrucciones:

```
# Archivo: myapp/.htaccess
RewriteEngine On
RewriteRule !(^static) application.php
```

Pasos finales tras la configuración del *Virtual Host* -Python y PHP-

Una vez configurado el Virtual Host, solo deberás:

1) Habilitar tu nuevo *host* virtual:

```
# a2ensite myapp
```

2) Reiniciar Apache:

```
# service apache2 restart
```

3) Si estás trabajando localmente, habilitar el host:

```
# echo '127.0.0.1    myapp.net' >> /etc/hosts
```

Por favor, notar que en el paso 3, se debe colocar el mismo nombre que se indicara en la directiva `ServerName` del *Virtual Host*.

Capítulo IV: El *core* en una aplicación MVC modular

En cualquier aplicación modular con una arquitectura MVC, el núcleo del sistema es algo que se diseña y programa una única vez y, más allá de que pueda «*refactorizarse*» y mejorarse con el tiempo, un único *core*, será suficiente para disponer de él en todas tus siguientes aplicaciones. Aquí radica la importancia de crear un *core* 100% genérico, robusto y reutilizable.

Archivos del núcleo

Entre los archivos del núcleo de la aplicación, mínimamente nos tendremos que encargar de generar los siguientes:

`core/dblayer` → `DBLayer()`

La capa de abstracción a bases de datos, que permita a los modelos conectarse y ejecutar consultas, será un recurso indispensable para nuestros modelos. No puede faltar ni ser reemplazada por otra herramienta, ni siquiera en el caso de que desee sumar un "plus" al *core*, desarrollando un ORM propio, puesto que éste, también la necesitará.

*En este capítulo **no veremos** como crear una capa de abstracción a bases de datos, puesto que es un tema que se ha tratado previamente en el **Capítulo X** de mi anterior libro: «**Teoría sintáctico-gramatical de objetos**[4]».*

`core/front_controller` → `FrontController()`

Es el "alma" de MVC en cuanto al control de la aplicación. Es el archivo sobre el cuál se manejará todo su Software. En el mismo, se iniciarán verdaderamente todas las solicitudes del usuario, puesto que será a `FrontController` a quien tu `application` le entregue dicha información. La responsabilidad de `FrontController`, será:

1. Analizar las peticiones del usuario a través de un *Application Handler*;
2. Decidir sobre el análisis realizado;
3. Importar e instanciar de forma dinámica, al controlador responsable de responder a la petición del usuario.

4 Puede obtener el libro impreso en papel o su versión digital gratuita ingresando en: http://www.bubok.es/libros/219288

`core/apphandler` → `ApplicationHandler()`

Será el responsable de:

1. Obtener la URI;
2. Analizarla;
3. Proveer la información necesaria a `FrontController` para que éste, pueda invocar al controlador correspondiente.

`core/helpers/template` → `Template()`

Un objeto que provea de una lógica genérica a las vistas. Nos encargaremos de él más adelante cuando hablemos de las vistas.

`core/helpers/patterns` → `DesignPattern()`

Si bien es opcional contar con proveedor de Patrones de Diseño, tomar la decisión de desarrollarlo será muy acertado sobretodo si en él se definen al menos dos patrones: `Factory` y `Compose`. Pues contar con ambos patrones predefinidos, nos permitirá diseñar modelos y controladores mucho más limpios y fáciles de mantener.

Nos concentraremos en este objeto más adelante, tras hacer la primera prueba con nuestra aplicación.

Front Controller y Application Handler

Comentamos anteriormente que `FrontController` iba a ser el encargado de analizar la URI (a través del analizador de un *Application Handler*) a fin de localizar la solicitud exacta del usuario y servirla.

También se mencionó al principio, que toda solicitud del usuario llegaría a través de la URL cumpliendo con el formato `modulo/modelo/recurso`.

Antes de diseñar el FrontController, es preciso comprender ilustrando con código, como éste deberá cumplir con su responsabilidad.

Analizar la URI (Application Handler)

La URI siempre tendrá el mismo formato:

```
http://example.org/modulo/modelo/recurso
```

El `FrontController` tendrá que utilizar la información contenida en la URI para:

- importar el controlador;
- instanciarlo y que éste ejecute el método que corresponde al recurso.

Es importante saber que todo modelo tendrá su

controlador. Esto significa que dado un modelo "a" (`models/a.py|php`) existirá un controlador "a" (`controllers/a.py|php`) y seguramente, también una vista "a" (`views/a.py|php`), pero ya llegaremos a eso.

También sabemos que tanto los directorios *models*, como *views* y *controllers* se alojan en la carpeta del módulo. Es decir que, dado un módulo foo, con un modelo bar, tendremos la siguiente estructura de archivos y directorios:

```
modules
└── foo
    ├── controllers/
    │   └── bar.py|php
    ├── models/
    │   └── bar.py|php
    └── views/
        └── bar.py|php
```

Lo anterior, significará que los siguientes objetos habrán sido definidos:

```
Bar()              en el modelo
BarController()    en el controlador y
BarView()          en la vista
```

Por consiguiente, nos conducirá a contar con una URI como la siguiente:

[domonio]/foo/bar/*recurso*

Si observamos la URI, ésta nos provee de:

a) el nombre de la carpeta (módulo) donde localizar el controlador;

b) el nombre del archivo donde se define el controlador;

c) lo necesario para armar el nombre del controlador e instanciarlo (si el archivo se llama `bar` el controlador será `BarController`);

d) finalmente, el recurso *«siempre»* será el nombre de un método del controlador.

Es decir, que si fuésemos capaces de dividir la URI utilizando como factor de división la barra diagonal (/), estaríamos obteniendo todo lo necesario. Veamos esto paso a paso, en ambos lenguajes.

En Python:

```
>>> uri = '/foo/bar/recurso'
>>> datos = uri.split('/')
>>> datos
['', 'foo', 'bar', 'recurso']
>>> modulo = datos[1]
>>> modelo = datos[2]
>>> recurso = datos[3]
>>> # archivo a importar
...
>>> archivo = 'modules.%(modulo)s.controllers.%(modelo)s' % dict(
...     modulo=modulo, modelo=modelo)
>>> archivo
'modules.foo.controllers.bar'
>>>
```

```
>>> # nombre del controlador
...
>>> controller_name = '%(modelo)sController' % dict(
...     modelo=modelo.title())
>>> controller_name
'BarController'
```

Ahora, en PHP:

```
php > $uri = 'foo/bar/recurso';
php > $datos = explode('/', $uri);
php > print_r($datos);
Array
(
    [0] => foo
    [1] => bar
    [2] => recurso
)
php > $modulo = $datos[0];
php > $modelo = $datos[1];
php > $recurso = $datos[2];
php >
php > # archivo del controlador
php >
php > $archivo = "modules/$modulo/controllers/$modelo.php";
php > echo $archivo;
modules/foo/controllers/bar.php
php >
php > # nombre del controlador
php >
php > $controller_name = ucwords($modelo) . "Controller";
php > echo $controller_name;
BarController
```

Es posible que luego del recurso, en la URL se pase un argumento (generalmente, la ID de un objeto a eliminar, visualizar o editar).

En ese caso, ya no siempre tendremos solo 3 elementos, sino que a veces, podremos tener 4.

También es importante contemplar la posibilidad de que en la URI, no se pasen los elementos suficientes y solo tengamos el dominio, el módulo y el modelo, el módulo solo, etc.

Considerando todas estas posibilidades, estamos en condiciones de diseñar el *Application Handler* que utilizará nuestro *Front Controller*.

En Python:

Primero, crearemos un *helper* para que nos resulte más simple (y menos redundante) la validación.

```
# Archivo: core/helprs/functions.py
def isset(var, i):
    result = False
    if len(var) >= (i + 1):
        result = False if var[i] == '' else True
    return result
```

```
# Archivo: core/apphandler.py
from settings import DEFAULT_MODULE, DEFAULT_MODEL,\
    DEFAULT_RESOURCE
from core.helpers.functions import isset


class ApplicationHandler(object):

    def analizer(cls, environ):
      uri = environ['REQUEST_URI']
      datos = uri.split('/')
      datos.pop(0)
      modulo = datos[0] if isset(datos, 0) else DEFAULT_MODULE
      modelo = datos[1] if isset(datos, 1) else DEFAULT_MODEL
      recurso = datos[2] if isset(datos, 2) else DEFAULT_RESOURCE
```

```
    arg = datos[3] if isset(datos, 3) else 0
    arg = 0 if arg == '' else arg
    return (modulo, modelo, recurso, arg)
```

En PHP:

```
# Archivo: core/apphandler.php
require_once 'settings.php';


class ApplicationHandler {

    public static function analizer() {
      $uri = $_SERVER['REQUEST_URI'];
      $datos = explode('/', $uri);
      array_shift($datos);
      $modulo = isset($datos[0]) ? $datos[0] : DEFAULT_MODULE;
      $modelo = isset($datos[1]) ? $datos[1] : DEFAULT_MODEL;
      $recurso = isset($datos[2]) ? $datos[2] : DEFAULT_RESOURCE;
      $arg = isset($datos[3]) ? $datos[3] : 0;
      $arg = ($arg == '') ? 0 : $arg;
      return array($modulo, $modelo, $recurso, $arg);
    }
}
```

Por favor, notar que en ambos casos, en las líneas donde se define el valor del módulo, del modelo y del recurso, se está haciendo referencia a 3 constantes que deberán definirse en el `settings`.

Otro punto fundamental a observar, es que la importación del *settings*, tanto en Python como en PHP, se ha realizado como si éste, estuviese siendo llamado desde el mismo directorio. ¿A qué se debe esto? A que en aplicaciones de este tipo, donde la interrelación entre sus componentes es tan

compleja, en ambos lenguajes se debe definir (en el `application`, como veremos más adelante), la ruta de importación por defecto, haciendo referencia al directorio de la aplicación. De esta manera, todas las importaciones se harán con el *namespace* (en el caso de Python) o la ruta de directorios (en el de PHP), de forma absoluta partiendo de la carpeta raíz de la aplicación.

Enviar la solicitud al controlador correspondiente (Front Controller)

Tenemos definido un *analizer* en nuestro *Application Handler*, quien nos proveerá de todo lo necesario para instanciar al controlador correspondiente y pedirle que sirva el recurso solicitado por el usuario.

Cómo vimos párrafos atrás, lo primero que tendrá que hacer el `FrontController` es solicitarle la información al `ApplicationHandler` para luego:

- Generar la ruta para la importación del archivo del controlador;
- Generar el nombre del controlador, para poder instanciarlo y que éste, sirva el recurso.

En Python:

```
# Archivo: core/front_controller.py
from core.apphandler import ApplicationHandler
```

```
class FrontController(object):

    def start(cls, environ):
      modulo, modelo, \
        recurso, arg =
ApplicationHandler().analizer(environ)

      ruta = 'modules.%(modulo)s.controllers.%(modelo)s' \
        % dict(modulo=modulo, modelo=modelo)
      cname = '%sController' % modelo.title()
      exec "from %(ruta)s import %(cname)s" % dict(
          ruta=ruta, cname=cname)
      controller = locals()[cname](recurso, arg,
         environ)
    return controller.output
```

En PHP:

```
# Archivo: core/front_controller.php
require_once 'core/apphandler.php';


class FrontController {

    public static function start() {
      list($modulo, $model, $recurso,
          $arg) = ApplicationHandler::analizer();

      $ruta = "modules/$modulo/controllers/$model.php";
      $cname = ucwords($model) . 'Controller';
      require_once $ruta;
      $controller = new $cname($recurso, $arg);
    }

}
```

Por favor, notar que al controlador, se le debe pasar el recurso como parámetro, para que éste, pueda efectuar una llamada de retorno desde su propio constructor. Por lo tanto, el constructor del controlador, debe estar preparado para recibirlo. Así

mismo, sucede con el argumento.

No obstante, existe la alternativa de que el *Front Controller* sea quien llame directamente al recurso del controlador una vez lo ha instanciado. Pero entonces ¿cómo elegir la opción correcta? Solo se deben analizar cuáles son las responsabilidades de cada uno de los objetos que intervienen en el mismo proceso.

Front Controller es un patrón de diseño (al igual que, por ejemplo: *Factory*, *Singleton*, *Composite*, *Proxy*, entre otros). Cómo tal, su función es la de *controlar toda la aplicación desde sí mismo*. Sin embargo, la relación y conducta interactiva entre éste y el controlador se ve limitada por la responsabilidad de este último: El controlador, a diferencia del *Front Controller*, es más que una clase. Es un objeto con sus métodos correspondientes. Y sólo él, puede decidir si el recurso que le está siendo solicitado, puede ser -o no- servido al usuario.

Es entonces, que la responsabilidad del *Front Controller*, termina en la instanciación del controlador y la entrega del recurso a éste, quien en definitiva, será el único autorizado a decidir si el recurso, puede ser servido o denegado.

Por otra parte, una gran diferencia a notar entre el *Front Controller* de Python y el de PHP, es que en el de Python, el *Front Controller* debe

indefectiblemente, capturar la salida del controlador para poder compatibilizar su funcionamiento con WSGI. Esto significa que: ***en Python, no son las vistas quiénes imprimen la GUI en pantalla, sino que éstas, las retornan al controlador y éste, a la vez, debe capturar dicha salida y almacenarla en una propiedad pública de clase para que WSGI acceda a ella y la imprima***. Por favor, notar que esto último, responde a la forma en la cual trabaja el módulo WSGI de Apache para procesar los archivos Python y permitir su ejecución en interfaces Web.

URIs compuestas: un caso particular a ser contemplado cuando se programa orientado a objetos

Es muy frecuente encontrarse con modelos cuyos nombres de clases se compongan de varias palabras. Es así que podremos encontrar en nuestro modelos, clases como `NotaDeCredito`, `NotaDeDebito`, etc.

En estos casos, tendremos que tener en cuenta dos factores:

1) al transmitir dicho modelo por la URI, habrá que considerar una forma simple de poder transformarlo en el nombre exacto (con estilo *CamelCase*) sin utilizar dicho estilo en la URI;

2) el nombre del archivo del modelo, sin embargo, podrá formarse por el nombre de la clase en minúsculas sin separación de

palabras o en todo caso, decidir por separar las palabras mediante guiones bajos;

La pregunta, entonces, es: ¿quién se encargará de considerar dichos factores?

Para responder a esta pregunta, se deberá tener en cuenta la responsabilidad de cada uno de los componentes:

- *Application Handler*: sabemos que es quién se debe encargar de analizar la URI y obtener los datos necesarios para que el *Front Controller* pueda llamar al controlador correspondiente;
- *Front Controller*: sabemos que es el encargado de llamar al controlador solicitado por el usuario.

Si *Front Controller* solo tiene la responsabilidad de comunicarse con el controlador y *Application Handler*, la de analizar la URI, a simple vista, quien debe hacerse cargo de considerar los factores antes mencionados, es el *Application Handler*.

Pero aquí, surge una nueva pregunta: ¿cómo transmitirá al controlador el nombre del modelo? ¿Con formato de clase o con formato de archivo? O por el contrario ¿transmitirá un nuevo elemento?

Lo cierto, es que la solución más simple, implica

contemplar cuestiones relativas a los estilos aplicados en el diseño de escritura de la aplicación.

Si convenimos en que los nombres de archivos se definan mediante la separación de palabras por guiones bajos, el *Application Handler*, podría transferir el modelo al *Front Controller*, solo en formato de archivo y éste, reemplazar los guiones bajos por espacios, convertir a *CamelCase* y así, quitar los espacios para obtener el nombre del controlador.

En este caso, estaríamos en presencia de una responsabilidad compartida. La alternativa, sería que el proceso de conversión anterior, fuese realizado íntegramente por el *Application Handler* y que el *Front Controller* se encargara de obtener ambos datos, con la salvedad de que el nombre de la clase y del archivo, *Application Handler* los debería enviar agrupados, pues pertenecen a un mismo componente. Y esta alternativa implicaría que *Front Controller* tras recibir los datos, se encargara de dividir dicho agrupamiento. Personalmente, prefiero la primera alternativa, aunque considero que estaría sujeta a discrepancia.

Recomiendo la utilización de la primera alternativa. No obstante, aquí **expondré sendas opciones para que el lector, decida** cuál de las dos implementar.

Caso #1: Application Handler retorna el modelo con formato de archivo (opción recomendada)

En Python:

```
# Cambios en apphandler.py (resaltados en negritas)
def analizer(cls, environ):
    uri = environ['REQUEST_URI']
    datos = uri.split('/')
    datos.pop(0)
    modulo = datos[0] if isset(datos, 0) else DEFAULT_MODULE
    modelo = datos[1] if isset(datos, 1) else DEFAULT_MODEL
    modelo = modelo.replace('-', '_')
    recurso = datos[2] if isset(datos, 2) else DEFAULT_RESOURCE
    arg = datos[3] if isset(datos, 3) else 0
    arg = 0 if arg == '' else arg
    return (modulo, modelo, recurso, arg)


# Cambios en front_controller.py
# (resaltados en negritas)
def start(cls, environ) {
    modulo, modelo, \
      recurso, arg = ApplicationHandler().analizer(environ)
    ruta = 'modules.%(modulo)s.controllers.%(modelo)s'\
      % dict(modulo=modulo, modelo=modelo)
    cname = modelo.replace('_', ' ').title().replace(
      ' ', '')
    cname = '%sController' % cname
    exec "from %(ruta)s import %(cname)s" % dict(
          ruta=ruta, cname=cname)
    controller = locals()[cname](recurso, arg, environ)
    return controller.output
```

En PHP:

```
# Cambios en apphandler.php (resaltados en negritas)
public static function analizer() {
    $uri = $_SERVER['REQUEST_URI'];
    $datos = explode('/', $uri);
    array_shift($datos);
    $modulo = isset($datos[0]) ? $datos[0] : DEFAULT_MODULE;
```

```
    $modelo = isset($datos[1]) ? $datos[1] : DEFAULT_MODEL;
    $modelo = str_replace('-', '_', $modelo);
    $recurso = isset($datos[2]) ? $datos[2] : DEFAULT_RESOURCE;
    $arg = isset($datos[3]) ? $datos[3] : 0;
    $arg = (arg == '') ? 0 : $arg;
    return array($modulo, $modelo, $recurso, $arg);
}


# Cambios a front_controller.php
# (resaltados en negritas)
public static function start() {
    list($modulo, $model, $recurso,
      $arg) = ApplicationHandler::analizer();

    $ruta = "modules/$modulo/controllers/$model.php";
    $cname = str_replace('_', '', $model);
    $cname = ucwords($cname) . 'Controller';
    $cname = str_replace(' ', '', $cname);
    require_once $ruta;
    $controller = new $cname($recurso, $arg);
}
```

Caso #2: Application Handler retorna el modelo con formato de archivo y de clase

En Python:

```
# Cambios en apphandler.py (resaltados en negritas)
def analizer(cls, environ):
    uri = environ['REQUEST_URI']
    datos = uri.split('/')
    datos.pop(0)
    modulo = datos[0] if isset(datos, 0) else DEFAULT_MODULE
    modelo = datos[1] if isset(datos, 1) else DEFAULT_MODEL
    modelo = modelo.replace('-', '_)
    cnanme = modelo.replace('_', ' ').title().replace(
      ' ', '')
    recurso = datos[2] if isset(datos, 2) else DEFAULT_RESOURCE
    arg = datos[3] if isset(datos, 3) else 0
    arg = 0 if arg == '' else arg
    return (modulo, (modelo, cname), recurso, arg)
```

```
# Cambios en front_controller.py
# (resaltados en negritas)
def start(cls, environ) {
    modulo, modelo, \
      recurso, arg = ApplicationHandler().analizer(environ)

    modelo = modelo[0]
    cname = "%sController" % modelo[1]

    ruta = 'modules.%(modulo)s.controllers.%(modelo)s'\
      % dict(modulo=modulo, modelo=modelo)
    exec "from %(ruta)s import %(cname)s" % dict(
          ruta=ruta, cname=cname)
    controller = locals()[cname](recurso, arg, environ)
    return controller.output
```

En PHP:

```
# Cambios en apphandler.php (resaltados en negritas)
public static function analizer() {
    $uri = $_SERVER['REQUEST_URI'];
    $datos = explode('/', $uri);
    $modulo = isset($datos[0]) ? $datos[0] : DEFAULT_MODULE;
    $modelo = isset($datos[1]) ? $datos[1] : DEFAULT_MODEL;
    $modelo = str_replace('-', '_', $modelo);
    $cname = str_replace('_', '', $modelo);
    $cname = str_replace(' ', '', ucwords($cname));
    $recurso = isset($datos[2]) ? $datos[2] : DEFAULT_RESOURCE;
    $arg = isset($datos[3]) ? $datos[3] : 0;
    $arg = (arg == '') ? 0 : $arg;
    return array($modulo, array($modelo, $cname),
      $recurso, $arg);
}


# Cambios a front_controller.php
# (resaltados en negritas)
public static function start() {
    list($modulo, $modelo, $recurso,
      $arg) = ApplicationHandler::analizer();
    $modelo = $modelo[0];
    $cname = "{$modelo[1]}Controller";
```

```
    $ruta = "modules/$modulo/controllers/$model.php";

    require_once $ruta;
    $controller = new $cname($recurso, $arg);
}
```

Primeros pasos con *Application*

Pues bien. A esta altura tenemos los componentes principales para ya, poder comenzar a probar nuestra aplicación y ver los resultados. El *core*, cuenta con lo mínimo necesario:

- El *Front Controller*;
- El *Application Handler*;
- Y si ya la has creado, la capa de abstracción a base de datos, que para hacer nuestra primera prueba, no nos será necesaria.

Solo nos restaría completar nuestro `application`.

Recordemos que este archivo, estará fuera del *core* y dentro de la carpeta raíz de la aplicación. Nos enfocamos en él dentro de este capítulo, solo a fin de poder realizar nuestra primera prueba y verificar a través del navegador, que la aplicación efectivamente está yendo por un buen camino.

Lo primero que debemos mencionar sobre *Application*, es una de sus principales

responsabilidades, que consiste en configurar la ruta de importación por defecto. En el caso de Python, lo hará con la ayuda del diccionario `environ` que le es entregado por WSGI. En el de PHP, bastará con importar el `settings` y utilizar la constante `APP_DIR`.

La otra responsabilidad que tendrá a su cargo, será la de inicializar al método `start()` de `FrontController`.

Antes de ver el código, es de hacer notar que si bien en la mayoría de los archivos los algoritmos son muy similares cuando se compara los de Python con los de PHP, en `application` se implementa una lógica diferente, ya que en Python, para las aplicaciones Web bajo Apache, todos los procesos lógicos se ven sujetos a los condicionamientos impuestos por el módulo WSGI. El más significativo, es la condición de la presencia de una función denominada `application()` -no puede tener otro nombre- y el siguiente, que cualquier modificación a los encabezados HTTP, debe hacerse mediante la función `start_response` recibida por `application()` a través de WSGI. Prepararemos el *Application* de Python, previendo estas circunstancias.

En Python:

```
# Archivo: application.py
from sys import path
```

```
def application(environ, start_response):
    ruta = environ['SCRIPT_FILENAME'].replace(
      'application.py', '')
    path.append(ruta)
    from core.front_controller import FrontController

    # Manejar la salida
    # para manipular encabezados HTTP
    headers = []
    salida = FrontController().start(environ)
    if isinstance(salida, tuple):
        headers.append(salida)
        salida = ''
    else:
        headers.append(('Content-Type',
             'text/html; charset=utf-8'))
    start_response('200 OK', headers)

    return salida
```

En PHP:

```
# Archivo: application.php
require_once 'settings.php';
ini_set('include_path', APP_DIR);

require_once 'core/front_controller.php';
FrontController::start();
```

Como se puede observar, la diferencia entre ambos *scripts* (el de Python y el de PHP) es radical. Es de hacer notar, que Python no es un lenguaje originalmente pensado para el desarrollo de aplicaciones Web y que a diferencia de PHP, el módulo de Apache que se encarga de su interpretación, requiere de modificaciones

ineludibles en el código fuente de los programas.

No obstante, WSGI es el mejor módulo de Apache para correr aplicaciones Web desarrolladas con Python. Personalmente, agradezco su existencia pero no estoy conforme con algunos de los condicionamientos que impone. Sin embargo, jamás me he sentado a intentar mejorar el módulo WSGI ni mucho menos, a intentar crear uno nuevo. Por ese motivo, lo sigo utilizando "sin quejas".

Pero, ambos lenguajes tienen "sus pro y sus contra". Ninguno de los dos es perfecto y solo tenemos dos alternativas:

1) Usarlos sin quejas;
2) Intentar mejorarlos o crear uno nuevo;

Yo elijo usarlos y si puedo mejorarlos -por más mínima que resulte esa mejora- desde ya que lo hago. Pero lo que no deja de molestarme un poco, es que se los compare y "se los haga competir". En mi opinión, no es uno mejor que el otro y ambos, tienen excelentes características que sabiéndolas aprovechar, se pueden desarrollar implementaciones efectivas y sumamente poderosas.

Ahora que ya hemos creado nuestro *Application*, crearemos un módulo de prueba (al que llamaré *"modulo"*) con solo un controlador (al que llamaré

"FooController") que cuente con un mínimo recurso (al que llamaré *"bar"*) que retorne un simple 'Hola Mundo'.

Necesitaremos (¡A no olvidarse!) las siguientes carpetas y archivos:

```
modules/
└── modulo
    ├── controllers
    │   └── foo.py|php
    ├── models
    └── views
```

Y en el archivo `foo` haremos el siguiente controlador de prueba:

En Python:

```
class FooController(object):

    def __init__(self, recurso, arg, environ):
        self.output = getattr(self, recurso)(arg)

    def bar(self, *arg):
        return 'Hola mundo!'
```

En PHP:

```
class FooController {

    public function __construct($recurso, $arg) {
        call_user_func(array($this, $recurso), $arg);
    }

    public function bar() {
```

```
        print 'Hola mundo!';
    }

}
```

Para probar la aplicación, ingresaremos en

`http://myapp.net/modulo/foo/bar`

(o el dominio indicado en el `ServerName` del `VirtualHost`).

Ayudantes: Los *Helpers Design Pattern* y *Template*

El *helper* `DesignPattern` si bien es opcional, es altamente recomendable cuando se trabaja con objetos.

En Python, no podrá evitarse utilizar un *Helper* para componer, puesto que el lenguaje no permite de forma nativa, la composición de objetos especificando previamente el tipo esperado. En cambio, PHP a pesar de también ser un lenguaje de *tipado* dinámico, toma "prestada" de C, la idea de componer objetos especificando su tipo en el parámetro de un método o función. Por este motivo, en PHP podría considerarse innecesario sumar un *helper* para esto.

Sin embargo, un *factory helper*, será sumamente necesario (en ambos lenguajes) al momento de:

- Recuperar un objeto desde el método `get()`

de un modelo, donde alguna de sus propiedades sea compuesta;

- Recuperar uno o más objetos desde cualquiera de los métodos de un controlador.

En el caso de Python, crearemos ambos *helpers* y en el de PHP, solo *Factory*. Veamos entonces el nuevo código.

En Python:

```
# Archivo: core/helpers/patterns.py

# Composite
def compose(obj, cls):
    if isinstance(obj, cls) or obj is None:
      return obj
    else:
      raise TypeError('%s no es de tipo de %s') % (
        type(obj), cls)


# Factory
def make(cls, value, name):
    obj = cls()
    property = name if name else '%s_id' % \
      obj.__class__.__name__.lower()
    setattr(obj, property, value)
    obj.get()
    return obj
```

En PHP (solo Factory):

```
# Archivo: core/helpers/patterns.php
function make($cls, $value, $name='') {
    $property = ($name) ? $name : strtolower($cls) . '_id';
```

```
    $obj = new $cls();
    $obj->$property = $value;
    $obj->get();
    return $obj;
}
```

Para conocer más sobre *Factory*, su implementación y forma en la cual ayudará a nuestros objetos, recomiendo leer el Capítulo XVI de mi anterior libro «*Teoría sintáctico-gramatical de objetos*[5]».

Con respecto al *Template*, lo dejaremos para más adelante cuando hablemos sobre las vistas.

5 http://www.bubok.es/libros/219288

Capítulo V: Los modelos en MVC

Como comentábamos en el Capítulo II, los modelos en MVC son archivos que agrupan clases que definen a un único tipo de objeto y en caso de existir, a sus subtipos y/o relacionales.

De esta forma, en un módulo de contabilidad, podremos tener un objeto `Comprobante`, con los subtipos `NotaDeDebito`, `NotaDeCredito`, `Factura`, `Remito`, `NotaDePedido`, etc. Todas las definiciones de estos objetos (clases), para MVC, pertenecerían a un mismo modelo llamado como el objeto principal: `comprobante.py|php`.

Sin embargo, la agrupación de clases en un único modelo, suele traer aparejados más traspiés que beneficios, cuando la cantidad de definiciones es superior a dos (incluso, a veces trae conflictos cuando son solo dos).

Los conflictos aparejados al agrupamiento de clases, está intrínsecamente relacionado a la complejidad de manejo de los recursos. Pues desde un mismo controlador (`ComprobanteController`) se deben manejar los recursos para todos los subtipos.

Por ello, mi consejo es **un modelo por clase** independientemente de su tipo.

Las **únicas excepciones**, serán:

1) Los *Singleton* colectores;

2) Los casos en los cuáles, se necesite el diseño de **Conectores Lógicos Relacionales** y/o de **Relacionales Simples**, donde sin excepción (y sobre todo, para evitar una incómoda recursividad en las importaciones), **se agruparán en el modelo del objeto compuesto**.

De esta forma, dados los objetos `NotaDePedido`, `Producto` y `ProductoNotaDePedido`, este último se agrupará con su compuesto `NotaDePedido` en el modelo `notadepedido`.

Nuevamente, recomiendo la lectura de mi anterior libro *«Teoría sintáctico-gramatical de objetos*[6]» para saber con exactitud, como diseñar objetos de forma correcta y crear sus estructuras de datos correspondientes, mediante un mapeo relacional de los mismos. Sobretodo, puedes enfocarte en los Capítulos VI, XVIII y IX los cuáles te serán de utilidad para ampliar la información sobre Objetos Relacionales Simples, Conectores Lógicos y Modelado de objetos, respectivamente.

---

6 http://www.bubok.es/libros/219288

Para finalizar este capítulo y solo a modo de ejemplo (para luego reutilizar los mismos en la medida que vayamos avanzando), crearemos un módulo y un modelo "más real" que "módulo Foo" y lo mapearemos relacionalmente para crear nuestra primera base de datos.

Crearemos entonces, un módulo llamado `produccion` donde incorporaremos los modelos `MateriaPrima`, `MateriaPrimaCollection`, `Producto` y `ProductoCollection` (si lo deseas, puedes eliminar el anterior módulo de pruebas y su controlador, puesto que no volveremos a utilizarlo).

Tendremos entonces, la siguiente estructura de archivos y directorios:

```
modules/
└── construccion
    ├── controllers
    ├── models
    │   ├── materia_prima.py|php
    │   └── producto.py|php
    └── views
```

Es hora, de comenzar a "codear" nuestro modelos.

En Python:

```
# Archivo: materia_prima.py
from core.dblayer import run_query


class MateriaPrima(object):

    def __init__(self):
      self.materiaprima_id = 0
      self.denominacion = ''

    def save(self):
      if self.materiaprima_id == 0:
          sql = """INSERT INTO materiaprima
                   (denominacion)
                   VALUES ('%s')
          """ % self.denominacion
          self.materiaprima_id = run_query(sql)
      else:
          sql = """UPDATE materiaprima
                   SET denominacion = '%s'
                   WHERE materiaprima_id = %i
          """ % (self.denominacion, self.materiaprima_id)
          run_query(sql)

    def get(self):
      sql = """SELECT materiaprima_id, denominacion
               FROM materiaprima
               WHERE materiaprima_id = %i""" % \
                   self.materiaprima_id
      fields = run_query(sql)[0]
      self.denominacion = fields[1]

    def destroy(self):
      sql = """DELETE FROM materiaprima
               WHERE materiaprima_id = %i""" % \
                   self.materiaprima_id
      run_query(sql)
      self.materiaprima_id = 0



# Archivo: producto.py
from core.dblayer import run_query
```

```
from core.helpers.patterns import make, compose
from modules.produccion.models.materia_prima import MateriaPrima


class Producto(object):

    def __init__(self):
      self.producto_id = 0
      self.denominacion = ''
      self.materiaprima_collection = []

    def add_materiaprima(self, obj):
      mp = componse(obj, MateriaPrima)
      self.materiaprima_collection.append(mp)

    def save(self):
      if self.producto_id == 0:
          sql = """INSERT INTO producto
                   (denominacion)
                   VALUES ('%s')
          """ % self.denominacion
          self.producto_id = run_query(sql)
      else:
          sql = """UPDATE producto
                   SET denominacion = '%s'
                   WHERE producto_id = %i
          """ % (self.denominacion, self.producto_id)
          run_query(sql)

      rel = MateriaPrimaProducto(self)
      rel.save()

    def get(self):
      sql = """SELECT producto_id, denominacion
               FROM producto
               WHERE producto_id = %i""" % \
                   self.producto_id
      fields = run_query(sql)[0]
      self.denominacion = fields[1]
      rel = MateriaPrimaProducto(self)
      rel.get()

    def destroy(self):
      sql = """DELETE FROM producto
               WHERE producto_id = %i""" % \
                   self.producto_id
```

```
        run_query(sql)
        self.producto_id = 0


class MateriaPrimaProducto(object):

    def __init__(self, obj):
        self.rel_id = 0
        self.producto = compose(obj, Producto)
        self.collection = obj.materiaprima_collection

    def save(self):
        self.destroy()
        sql = """INSERT INTO materiaprimaproducto
                 (producto, materiaprima)"""
        data = []
        tmpvar = 0

        for mp in self.collection:
            sql += ', ' if tmpvar > 0 else ' VALUES '
            sql += '(%i, %i)'
            data.append(self.producto.producto_id)
            data.append(mp.materiaprima_id)
            tmpvar += 1

        run_query(sql % tuple(data))

    def get(self):
        sql = """SELECT materiaprima
                 FROM materiaprimaproducto
                 WHERE producto = %i""" % \
                     self.producto.producto_id
        fields = run_query(sql)

        for tupla in fields:
            self.producto.add_materiaprima(
                make(MateriaPrima, tupla[0])

    def destroy(self):
        sql = """DELETE FROM materiaprimaproducto
                 WHERE producto = %i""" % \
                     self.producto.producto_id
        run_query(sql)
```

En PHP:

```
# Archivo: materia_prima.php
require_once 'core/dblayer.php';


class MateriaPrima {

    function __construct() {
      $this->materiaprima_id = 0;
      $this->denominacion = '';
    }

    function save() {
      if($this->materiaprima_id == 0) {
          $sql = "INSERT INTO materiaprima
                  (denominacion)
                   VALUES (?)";
          $data = array('s', "{$this->denominacion}");
          $this->materiaprima_id = DBObject::ejecutar(
             $sql, $data);
      } else {
          $sql = "UPDATE materiaprima
                  SET denominacion = ?
                  WHERE materiaprima_id = ?";
          $data = array('si', "{$this->denominacion}",
             "{$this->materiaprima_id}");
          DBObject::ejecutar($sql, $data);
      }
    }

    function get() {
      $sql = "SELECT materiaprima_id, denominacion
              FROM materiaprima
              WHERE materiaprima_id = ?";
      $data = array('i', "{$this->materiaprima_id}");
      $fields = array('materiaprima_id'=>'',
             'denominacion'=>'');
      DBObject::ejecutar($sql, $data, $fields);
      $this->denominacion = $fields['denominacion'];
    }

    function destroy() {
      $sql = "DELETE FROM materiaprima
              WHERE materiaprima_id = ?";
```

```
      $data = array('i', "{$this->materiaprima_id}");
      DBObject::ejecutar($sql, $data);
      $this->materiaprima_id = 0;
    }

}



# Archivo: producto.php
require_once 'core/dblayer.php';
require_once 'modules/construccion/models/materia_prima.php';


class Producto {

    function __construct() {
      $this->producto_id = 0;
      $this->denominacion = '';
      $this->materiaprima_collection = array();
    }

    function add_materiaprima(MateriaPrima $obj) {
      $this->materiaprima_collection[] = $obj;
    }

    function save() {
      if($this->producto_id == 0) {
          $sql = "INSERT INTO producto
                  (denominacion)
                  VALUES (?)";
      $data = array('S', "{$this->denominacion}");
      $this->producto_id = DBObject::ejecutar($sql,
            $data);
      } else {
          $sql = "UPDATE producto
                  SET denominacion = ?
                  WHERE producto_id = ?";
          $data = array('si', "{$this->denominacion}",
             "{$this->producto_id}");
          DBObject::ejecutar($sql, $data);
      }
    }

    function get() {
      $sql = "SELECT producto_id, denominacion
```

```
                FROM producto
                WHERE producto_id = ?";
        $data = array('i', "{$this->producto_id}");
        $fields = array('producto_id'=>'',
               'denominacion'=>'');
        DBObject::ejecutar($sql, $data, $fields);
        $this->denominacion = $fields['denominacion'];
        $rel = new MateriaPrimaProducto($this);
        $rel->get();
    }

    function destroy() {
        $sql = "DELETE FROM producto
                WHERE producto_id = ?";
        $data = array('i', "{$this->producto_id}");
        DBObject::ejecutar($sql, $data);
        $this->producto_id = 0;
    }

}



class MateriaPrimaProducto {

    function __construct(Producto $obj) {
        $this->rel_id = 0;
        $this->producto = $obj;
        $this->collection = $obj->materiaprima_collection;
    }

    function save() {
        $this->destroy();
        $sql = "INSERT INTO materiaprimaproducto
                (producto, materiaprima)";
        $data = array('');
        $tmpvar = 0;

        foreach($this->collection as $obj) {
            $sql .= ($tmpvar > 0) ? ', ' : ' VALUES ';
            $sql .= '(?, ?)';
            $data[0] .= 'ii';
            $data[] = "{$this->producto->producto_id}";
            $data[] = "{$obj->materiaprima_id}";
            $tmpvar++;
        }
```

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

    function destroy() {
      $sql = "DELETE FROM materiaprimaproducto
              WHERE producto_id = ?";
      $data = array('i',
             "{$this->producto->producto_id}");
      DBObject::ejecutar($sql, $data);
    }

}
```

Estructura de datos (SQL):

```
# Archivo: ~/db.sql
CREATE DATABASE myapp;
USE myapp;

CREATE TABLE materiaprima (
  materiaprima_id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY
  , denominacion VARCHAR(50)
) ENGINE=InnoDB;

CREATE TABLE producto (
  producto_id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY
  , denominacion VARCHAR(50)
) ENGINE=InnoDB;

CREATE TABLE materiaprimaproducto (
  rel_id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY
  , producto INT(11)
  , INDEX(producto)
  , FOREIGN KEY (producto)
      REFERENCES producto (producto_id)
      ON DELETE CASCADE
  , materiaprima INT(11)
  , INDEX(materiaprima)
  , FOREIGN KEY (materiaprima)
      REFERENCES materiaprima (materiaprima_id)
      ON DELETE CASCADE
) ENGINE=InnoDB;
```

Capítulo VI: Las vistas

En la Ingeniería de sistemas modulares bajo el patrón arquitectónico MVC, como bien se comentó al comienzo, el orden en el que los componentes del sistemas se desarrollan, es justamente el que hace honor al nombre del patrón: primero los modelos, luego las vistas y finalmente, los controladores.

En el capítulo anterior hablamos de los modelos pero sin entrar en demasiados detalles, pues es un tema más intrínsecamente relacionado a la orientación a objetos y ya he hablado mucho sobre eso, en una publicación anterior.

Es entonces que en este capítulo, nos toca hablar pura y exclusivamente de las vistas. Para ello, primero -y con muy pocas palabras- dejaré bien en claro que **en MVC la independencia de la parte gráfica es literal. No existen *"peros"* ni excepciones** de ningún tipo para mezclar archivos estáticos (sobretodo me refiero a HTML) con *scripts* de programación.

Si te propones comprender y aceptar lo anterior, no habrá imposibles y todo te resultará tan simple como sencillo de entender. *De lo contrario, si no logras convencerte a ti mismo de que no existen excepciones, leerás el libro completo y seguirás*

teniendo cientos de dudas al momento de diseñar y desarrollar una aplicación con el patrón arquitectónico MVC.

¿De qué se trata entonces? Como las vistas en MVC se encuentran dividas en dos partes, la GUI (archivos estáticos -diseño gráfico de la aplicación y comportamiento visual del lado del cliente y técnicas como AJAX inclusive-) y la lógica de las mismas, hablando de "lógica" es lógico que no podrás idear la "lógica" de una gráfica que aún no tienes. Entonces, «***lo primero que se debe hacer, es diseñar la parte gráfica***» de la aplicación (con esto me refiero, a tener el diseño gráfico ya maquetado de al menos el modelo sobre el cual se trabajará).

Si aún te sigues preguntando por qué la parte gráfica primero, cambia tu pregunta de "por qué" a "para qué", puesto que la Ingeniería Informática es lógica y no filosófica, el "por qué" no te conducirá a nada "lógicamente relevante". Entonces ***¿para qué diseñar primero la parte gráfica?*** **Para saber exactamente cómo diseñar la parte lógica**. Pues viendo "cuál es la necesidad gráfica" necesitarás encontrar una solución a ella. Y solo así, lograrás "desarrollar la aplicación que se necesita" en vez de "generar necesidades en la aplicación que se desarrolla".

Debes tener muy en cuenta que «*la Ingeniería de*

Sistemas es aquella que encuentra soluciones a una necesidad informática o problema informático determinado». Sin embargo, muchos Ingenieros y sobretodo programadores y diseñadores, se toman la atribución de "sugerir" cambios negligentes a las ideas del dueño de producto, generando así un nuevo problema que debe ser resuelto. Si sugieres cambios a la interfaz gráfica "deseada" en la aplicación, solo porque consideras imposible alcanzar dicho deseo, simplemente es que no estás pensando adecuadamente de qué forma puedes resolver la necesidad de tu cliente. Pues tu responsabilidad, siempre, debe ser "encontrar la solución y no, generar un nuevo problema".

Dicho todo lo anterior, es hora de comenzar a definir algunos conceptos técnicos.

Clasificación de los requerimientos gráficos

Pasé muchísimos años estudiando la forma de lograr que este maravilloso patrón arquitectónico, encontrase una forma de cumplir, de manera literal, con aquello de mantener aislada e independiente a las interfaces gráficas del resto de la aplicación.

Así fue que hallé dos grandes requerimientos visuales a los cuales clasifiqué en:

1) Requerimientos directos;

2) Requerimientos sustitutivos.

Requerimientos directos

Los requerimientos directos son aquellos en los cuáles se requiere una interfaz gráfica estática sin factores de variabilidad. Es decir, que solo se requiere mostrar un diseño gráfico, tal cual ha sido maquetado y en su interior, no se deberán presentar datos o información variable.

Este tipo de requerimiento, jamás actuará solo. Vale decir, que es imposible que una aplicación Web solo tenga requerimientos directos. Este tipo de requerimiento, apenas podrá representar el 1% -cuando mucho- de las necesidades visuales de una aplicación. Es el caso típico de una aplicación que en un momento determinado, requiere, por ejemplo, mostrar un formulario de "login" en HTML plano y puro que no incluya más información que el mismo formulario. Un ejemplo de ello, lo puedes ver en el siguiente enlace:

http://demo.europio.org/users/user/login

En estos casos, solo será necesario leer el contenido del archivo para ser impreso en pantalla de forma directa y se logra de manera sumamente sencilla como veremos a continuación.

En Python:

```
with open('archivo.html', 'r') as archivo:
    return archivo.read()
```

En PHP:

```
return file_get_contents('archivo.html');
```

Vale aclarar que en estos casos, sería incorrecto -a nivel arquitectura del sistema- enviar al usuario directamente a:

http://example.org/static/html/archivo.html. Aunque no existe un verdadero impedimento técnico para hacerlo, la falta se estaría cometiendo desde el punto de vista arquitectónico, pues en MVC, la arquitectura se basa en el manejo de "recursos" y en el caso anterior, se estaría omitiendo el manejo de dicho recurso, lo cual haría más compleja la evolución del mismo (si se desea evolucionar un recurso inexistente, se deberá crear el recurso desde cero, lo cual ineludiblemente implicaría resolverlo de la forma recomendada. Es decir, que *todos los "recursos" conducen a "Roma"*).

Requerimientos sustitutivos

Aquí es donde comienza el verdadero *"arte de las vistas"*.

Los requerimientos sustitutivos son aquellos en los

cuales, se requiere una interfaz gráfica con ciertos datos o información variable. Es el requerimiento del 99% de la aplicación -cuando menos-.

Es válido aclarar que todo requerimiento sustitutivo se iniciará de la misma forma que uno directo, pero antes de retornar el resultado deberá ser sustituido por información variable. Es decir, deberá efectuar las "sustituciones" pertinentes antes de presentarlo al usuario.

No existen casos puntuales ni ejemplos generalizadores. No obstante, un caso tradicional es el de una plantilla general para la aplicación, donde el contenido interior de la misma, vaya variando de acuerdo al recurso solicitado. A la vez, dicho factor de variación, se puede producir de forma recursiva, cuando un mismo recurso, deba mostrar la misma gráfica con información variable.

Subdivisión de los requerimientos sustitutivos

Como comentaba en el párrafo anterior, este tipo de requerimientos sustitutivos, abarca un amplio abanico de posibilidades que, para su mejor estudio, he subdividido en tres grupos:

1) Sustituciones directas[7];
2) Sustituciones dinámicas o iterativas; y
3) Sustituciones combinadas.

Sustituciones directas

Dentro de los requerimientos sustitutivos, las sustituciones directas son las que mayor simplicidad reflejan y a las que más nos hemos referido los estudiosos del tema a lo largo de toda la bibliografía especializada en MVC.

Dichas sustituciones son aquellas en las cuáles una interfaz gráfica posee ciertos sectores únicos de información variable. Esto significa que en una interfaz gráfica se podrá encontrar el SECTOR A, SECTOR B, etc y que SECTOR A será reemplazado por VALOR A, SECTOR B por VALOR B y así sucesivamente. Es decir, que cada sector variable tendrá asociado un único dato (no confundir "un solo dato" con "un dato único").

Ejemplos podremos encontrar cientos y hasta decenas de miles, pero para mayor comprensión

7 en publicaciones anteriores me he referido a este tipo de sustituciones como "estáticas", pero he decidido modificar su nomenclatura puesto que el término "estático" no era realmente apropiado para la definición que quería otorgar.

podemos citar el caso de una plantilla en la cual se requiera modificar el título y el subtítulo, según el recurso solicitado. Solo existen dos sectores de información variable: TÍTULO y SUBTÍTULO. Los cuáles, en cada recurso, deberán ser sustituidos por un único valor para el sector TÍTULO y un único valor para el sector SUBTÍTULO.

Plantilla:

```
<!doctype html>
<html>
    <head>
        ...
    </head>
    <body>
        <h1>TITULO</h1>
        <h2>SUBTITULO</h2>
        ...
    </body>
</html>
```

Se dice que aquí la sustitución será directa, ya que el sector identificado con la palabra TITULO deberá ser reemplazo por un valor y el sector identificado por la palabra SUBTITULO deberá ser reemplazado por otro valor. Es decir, cada identificador tendrá asociado un único valor.

Este tipo de sustituciones se logra de la misma forma que en una cadena de texto se reemplaza un valor por otro. La forma más “prolija” y que más respeta el

principio de “mantenibilidad” de un Software, es la **sustitución directa por diccionarios**.

Para realizar una sustitución directa por diccionarios, en Python, los sectores de información variable se deberán indicar anteponiendo el signo dólar al nombre del sector - `$SECTOR` - y en PHP, si bien no se encuentra especificado, una técnica bastante convencional, es encerrar el nombre del sector entre dos llaves de apertura respectivamente – `{SECTOR}` -.

Identificación de sectores para Python:

```
<!doctype html>
<html>
    <head>
        ...
    </head>
    <body>
        <h1>$TITULO</h1>
        <h2>$SUBTITULO</h2>
        ...
    </body>
</html>
```

Identificación de sectores para PHP:

```
<!doctype html>
<html>
    <head>
        ...
    </head>
    <body>
        <h1>{TITULO}</h1>
        <h2>{SUBTITULO}</h2>
        ...
```

```
    </body>
</html>
```

Sustitución directa por diccionarios en Python:

```
from string import Template

with open('plantilla.html', 'r) as archivo:
    plantilla = archivo.read()

diccionario = dict(
  TITULO='Requerimientos Sustitutivos',
  SUBTITULO='Sustituciones directas')

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

Sustitución directa por diccionarios en PHP:

```
$plantilla = file_get_contents('plantilla.html');

$diccionario = array(
  '{TITULO}'=>'Requerimientos Sustitutivos',
  '{SUBTITULO}'=>'Sustituciones directas',
);

return str_replace(array_keys($diccionario),
  array_values($diccionario), $plantilla);
```

Sustituciones dinámicas o iterativas

Las sustituciones iterativas son aquellas en la cuales existe uno o más sectores de información variable, que deben ser sustituidos por más de un valor. Es decir, que cada sector, tendrá asociados múltiples valores. Básicamente son sustituciones directas cuyo reemplazo debe realizarse de forma cíclica. Es el

caso típico de los “listado de objetos”.

Las sustituciones iterativas suelen ser las que más dudas generan en el programador. Sin embargo, estas dudas, no dejan de ser un síntoma de SMD: “síndrome del miedo a lo desconocido” :)

Solo debes pensar en estas sustituciones, como sustituciones directas que debes realizar dentro de un bucle.

Imagina que de esta simple tabla, debes sustituir los sectores NOMBRE y APELLIDO, por tantos objetos Persona como tengas creados en el sistema:

```
<table>
  <tr>
    <td>NOMBRE</td>
    <td>APELLIDO</td>
  </tr>
</table>
```

No debes pensar nunca, que no se puede hacer sin embeber código. Solo tienes que analizar el problema para luego encontrarle una solución.

Entonces, analiza:

¿Qué parte de ese código es el que se debe replicar y sustituir, tantas veces como objetos Persona haya creados en el sistema?

La respuesta, te debería resultar sencilla, pero de no ser este el caso, lo que puedes hacer es lo siguiente:

1. En un archivo vacío, coloca el HTML **final**, es decir, tal cuál debería quedar después de la sustitución.
2. Verifícalo visualmente para cerciorarte de que realmente haz hecho lo correcto.
3. Luego, mira el código fuente y fíjate que líneas de código haz copiado y pegado en forma repetitiva. Eso mismo, es lo que deberás lograr en la lógica de tu vista.

Muchas veces, comenzar por crear "a mano" el resultado que esperas, ayuda a visualizar mejor el problema e ir desmembrándolo hasta hallar la solución.

Ahora bien. Si hallaste la simple respuesta habrás notado que la parte del código que debes replicar y sustituir de forma iterativa es esta:

```
<tr>
  <td>NOMBRE</td>
  <td>APELLIDO</td>
</tr>
```

Ahora, antes de continuar analizando, olvida el resto del código de la tabla y piensa. Toma nota mental de las herramientas con las que efectivamente sabes que puedes contar:

1. Sabes que una única sustitución, puedes realizarla con el método de sustitución directa.
2. Sabes que los valores de sustitución, los dispones en una propiedad colectora (un *array* si de PHP se trata o una lista en el caso de Python. Dentro de ésta, tienes varios objetos). Si no puedes imaginar la propiedad colectora tal cual es, simplemente escríbela a mano.
3. Sabes que en Python, puedes iterar sobre una lista con `for objeto in lista` y que en PHP, puedes hacer lo propio con `foreach($lista as $objeto)`.
4. Si vuelves a recordar el código HTML que te habías propuesto olvidar, sabes que tienes la parte del código sobre el cuál podrías iterar dentro de otro código.
5. Sabes que ese "código completo" lo puedes obtener como una *string*.
6. Sabes que de una *string* puedes obtener solo una parte de ella.

La siguiente pregunta es:

¿Qué (no "cómo", sino "qué") te hace falta entonces?

La respuesta debería saltar a la vista:

"Ver cómo lograr obtener la fracción de código sobre la cuál realizar la sustitución iterativa".

Es probable que ya te des una idea de cómo lograrlo. De hecho, se puede lograr de varias formas diferentes. Pero hay una forma que es la más simple y directa posible que no requiere embeber código de ningún tipo: **a través de expresiones regulares**.

Una expresión regular es aquella que define un cierto patrón (o cadena) sin expresar necesariamente el literal de la misma.

Por ejemplo: imagina los antiguos ejercicios de "completar las palabras que faltan en la oración" que la maestra te daba en la escuela. ¿Recuerdas los guiones para completar las letras? Esos guiones representaban una forma de "expresar" una determinada palabra en la cadena, sin definir su literal. Mira este ejemplo:

"el miedo a lo desconocido es tu ____ enemigo".

Con cuatro guiones bajos, tu maestra expresaba una palabra sin necesidad de escribirla de forma literal. Al momento de completar el ejercicio, tu cerebro intentaba obtener dicha palabra, analizando el "patrón" (espacio disponible para escribir la palabra) y el contexto (palabras que la rodeaban y sentido de

la frase). De esa forma, podías encontrar que la palabra expresada de forma no-literal, era *"peor"*.

Ese mismo procedimiento es el realizado cuando indicas a un sistema informático, encontrar una expresión regular en una determinada cadena.

Pero para los sistemas informáticos es mucho más simple que para nosotros (al menos, que para los ejercicios que nos daban nuestras maestras), ya que para definir una palabra sin expresarla literalmente, existe un "código".

Para realizar sustituciones iterativas no necesitas convertirte en experto de códigos para expresiones regulares. Simplemente, necesitas conocer el código que TU expresión regular va a requerir. No obstante, considero una buena idea, que si no conoces de expresiones regulares, le des una mirada al artículo sobre Expresiones Regulares en Wikipedia:

http://es.wikipedia.org/wiki/Expresión_regular

Volviendo a nuestro HTML y el ejemplo de la maestra, nosotros necesitamos definir una expresión regular para identificar la fracción de código sobre la cuál iterar. Siempre se debe pensar en lograr

código tan genérico como sea posible. Y si para cada fracción de código tuviésemos que definir una expresión regular diferente, no solo nunca hallaremos un código genérico sino que peor aún, sí deberíamos convertirnos en expertos de las expresiones regulares. Entonces, es mejor hallar una forma de solo necesitar una expresión regular genérica sin ensuciar el código HTML ni el de nuestros *scripts*. La forma que yo sugiero, es identificar en el HTML la fracción de código sobre la que se necesita iterar, con un comentario HTML bien sencillo. Por ejemplo:

```
<table>
  <!--ITERAR-->
  <tr>
    <td>NOMBRE</td>
    <td>APELLIDO</td>
  </tr>
  <!--FIN:ITERAR-->
</table>
```

Incluso, podríamos simplificarlo aún más, utilizando en ambas ocasiones el mismo comentario. Por ejemplo:

```
<table>
  <!--ITERAR-->
  <tr>
    <td>NOMBRE</td>
    <td>APELLIDO</td>
  </tr>
  <!--ITERAR-->
</table>
```

De esa forma, podríamos expresar el comentario HTML de manera literal y con una expresión regular, decirle a nuestro *script* que busque cualquier caracter o salto de línea repetido cualquier cantidad de veces y ya obtendríamos una expresión regular genérica para todos nuestros códigos:

```
<!--ITERAR-->(.|\n){1,}<!--ITERAR-->
```

En la expresión anterior, entre paréntesis estamos agrupando una condición:

```
.    representa cualquier carácter
\n   representa un salto de línea y
|    marca la condición diciendo “o” (or)
```

Luego, entre llaves, le indicamos la cantidad de veces que la condición debe repetirse.

```
1        indica el mínimo número de veces
,(nada)  indica que no existe un límite máximo
```

La expresión regular anterior, podría hacerse mucho más genérica aún, si la palabra “ITERAR” no la expresáramos de forma literal. De esta manera, podríamos utilizar diferentes palabras o frases en nuestros comentarios HTML:

```
<!--(.){1,}-->(.|\n){1,}<!--(.){1,}-->
```

Pero si nos excedemos en “programadores genéricos”, en vez de facilitar las cosas, las estamos limitando sin quererlo. Entonces ¿por qué no

simplificarlo utilizando una variable que nos permita en el código de nuestros *scripts* tener un mayor control sobre lo que estamos haciendo?

En Python:

```
"<!--%(k)s-->(.|\n){1,}<!--%(k)s-->" % dict(k=var)

var = 'ITERAR'
# producirá:
<!--ITERAR-->(.|\n){1,}<!--ITERAR-->

var = 'LISTA-DE-PERSONAS'
# producirá:
<!--LISTA-DE-PERSONAS-->(.|\n){1,}<!--LISTA-DE-PERSONAS-->
```

En PHP:

```
<!--$var-->(.|\n){1,}<!--$var-->

$var = 'ITERAR';
# producirá:
<!--ITERAR-->(.|\n){1,}<!--ITERAR-->

$var = 'LISTA-DE-PERSONAS';
# producirá:
<!--LISTA-DE-PERSONAS-->(.|\n){1,}<!--LISTA-DE-PERSONAS-->
```

Una vez definida la expresión regular, solo resta saber cómo ésta se implementa en cada lenguaje y cómo se le indica al *script* que la busque.

En Python:

```
import re
regex = re.compile(
  "<!--%(k)s-->(.|\n){1,}<!--%(k)s-->" % dict(k=var))
```

```
match = regex.search(plantilla).group(0)
```

En PHP:

```
$regex = "/<!--$var-->(.|\n){1,}<!--$var-->/";
preg_match($regex, $plantilla, $matches);
$match = $matches[0];
```

En ambos casos, la variable `match` será la que contenga el código sobre el cuál realizar la sustitución iterativa.

En PHP, es necesario hacer notar que por un lado, la expresión regular no solo se define entre comillas sino que además, toda la expresión debe estar envuelta entre dos barras diagonales //. Por otra parte, el *array* `$matches` se define "al vuelo" y es modificado por referencia, por lo cuál, no necesita ser declarado.

Ahora bien. Hemos obtenido nuestra fracción de código y así, solo restará iterar sobre la misma, acumulando en cada iteración, el resultado de la sustitución. Suponiendo que contamos con una propiedad colectora, nuestro código debería verse como el que sigue:

En Python:

```
<!-- el HTML para Python -->
<table>
```

```
  <!--PERSONAS-->
  <tr>
    <td>$nombre</td>
    <td>$apellido</td>
  </tr>
  <!--PERSONAS-->
</table>

# El Script:
import re
from string import Template

with open('plantilla.html', 'r') as archivo:
    plantilla = archivo.read()

var = 'PERSONAS'
# se pasaría x parámetro a una función

regex = re.compile(
  "<!--%(k)s-->(.|\n){1,}<!--%(k)s-->" % dict(k=var))
match = regex.search(plantilla).group(0)

render = ''

for objeto in coleccion:
    diccionario = dict(
      nombre=objeto.nombre,
      apellido=objeto.apellido)
    render += Template(match).safe_substitute(
      diccionario)
```

En PHP:

```
<!-- el HTML para PHP -->
<table>
  <!--PERSONAS-->
  <tr>
    <td>{NOMBRE}</td>
    <td>{APELLIDO}</td>
  </tr>
  <!--PERSONAS-->
</table>
```

```
# El Script:
plantilla = file_get_contents('plantilla.html');

$var = 'PERSONAS';
# se pasaría x parámetro a una función

$regex = "/<!--$var-->(.|\n){1,}<!--$var-->/";
preg_match($regex, $plantilla, $matches);
$match = $matches[0];

$render = '';

$sectores = array('{NOMBRE}', '{APELLIDO}');

foreach($coleccion as $obj) {
    $sustitutos = array($obj->nombre, $obj->apellido);
    $render += str_replace($sectores, $sustitutos,
      $match);
}
```

Con los *scripts* anteriores, lograríamos un HTML sustituido en la variable `render`, que se asemejaría al siguiente:

```
<!--PERSONAS-->
<tr>
  <td>Juan</td>
  <td>Pérez</td>
</tr>
<!--PERSONAS--><!--PERSONAS-->
<tr>
  <td>Pedro</td>
  <td>Gómez</td>
</tr>
<!--PERSONAS--><!--PERSONAS-->
<tr>
  <td>Ana</td>
  <td>Rodríguez</td>
</tr>
<!--PERSONAS-->
```

Podríamos incluso, eliminar los “feos muy feos”

comentarios HTML repetidos.

En Python:

```
render.replace('<!--%s-->' % var, '')
```

En PHP:

```
str_replace("<!--$var-->", '', $render);
```

Pero finalmente, nos restará que el código sustituido aparezca en toda la plantilla. Pues hemos logrado tenerlo por separado.

Nuevamente, igual que venimos haciendo desde el inicio, te pregunto:

¿Qué necesitas? ¿Qué tienes? ¿Cómo lo logras?

La respuesta está frente a tus ojos:

- ***Necesitas:*** que tu variable `render` no se muestre sola sino en el contexto de la plantilla.
- ***Tienes:*** el valor de sustitución (tu variable `render`) y el sector al cuál sustituir (en tu variable `match`) ¿te lo esperabas? Imagino que ¡sí!
- ***Lo logras:*** con una sustitución directa[8].

8 Notar que en el caso de Python, será mucho más simple utilizar

En Python:

```
plantilla.replace(match, render)
```

En PHP:

```
str_replace($match, $render, $plantilla);
```

No te creeré si me dices que no dijiste "¡Ah! ¡Claro!". Resulta obvio ¿cierto?.

Sustituciones combinadas

Las sustituciones combinadas son aquellos requerimientos gráficos sustitutivos, que demandan -en una misma GUI- tanto sustituciones directas como iterativas.

Aquí no hay mucho que explicar, sin embargo, siempre suele aparecer la duda sobre cómo lograrlo.

La forma más acertada es realizando las sustituciones de a una. En la primera sustitución, necesitarás leer la plantilla. En las siguientes sustituciones, necesitarás utilizar como base, el resultado de la sustitución anterior.

Siempre es más acertado comenzar por las sustituciones iterativas y finalizar con la sustitución directa. No obstante, a pesar de ser la forma más

---

`replace()` que `Template()`

acertada (correcta) no necesariamente es la más simple. Pero te sugiero “dar el gran salto” y hacerlo de la forma correcta.

Lo que siempre debes tener en cuenta es que el contenido del archivo HTML solo lo leerás una vez y sobre él, realizarás la primera de todas las sustituciones. Esa sustitución, arrojará como resultado, el archivo HTML sustituido. Sin embargo, a ese resultado, aún le quedarán sustituciones por delante. A partir de ese momento, siempre deberás utilizar como “cadena a sustituir”, el resultado de la última sustitución que hayas realizado:

```
contenido = archivo.html
realizo sustitución iterativa sobre contenido:
      La obtengo en 'render'.
Sustituyo match por render en plantilla:
      obtengo la variable 'sustitucion1'
* FIN sustitución iterativa 1 *

realizo 2da. sustitución iterativa sobre sustitucion1:
      La obtengo en 'render2'.
Sustituyo match2 por render2 en sustitucion1:
      obtengo la variable 'sustitucion2'
* FIN sustitución iterativa 2 *

realizo sustitución directa sobre 'sustitucion2':
      obtengo la variable 'sustitucion3'
      si no hay más sustituciones:
            imprimo sustitucion3
```

Funcionamiento interno de las vistas y su responsabilidad

Antes de seguir avanzado se hace inminente la necesidad de comprender con exactitud, cómo es el

funcionamiento interno de las vistas y cuáles son sus responsabilidades.

Introducción

Como bien se ha comentado, las vistas se encargan de mostrar al usuario, de forma humanamente legible, los datos solicitados por éste. Para ello, debe "ingeniárselas" para convertir los requerimientos gráficos visuales en "un sueño hecho realidad". Y más allá de la metáfora, esto, en MVC, es casi literal.

La lógica de las vistas, si bien se denomina "lógica", más que ésta es un verdadero arte. Un arte que se inicia con los diseñadores gráficos, quiénes deben capturar la idea abstracta de un cliente, en "imágenes y colores" creando así, verdaderas obras de arte.

Como programadores y Arquitectos, tenemos la obligación de conservar esas obras de arte intactas en su estado original y para ello, más allá de toda lógica, debemos intentar alcanzar un grado de creatividad suficiente para que el arte de nuestros diseñadores gráficos no deje de brillar y deslumbrar. Pues el peor error que podemos cometer, es realizar cambios visuales para que nuestra labor, se vea aliviada. Se hace necesario comprender -y por ello hago hincapié en esto-, que un cambio visual es apenas un eufemismo de la ruina de un arte.

Podemos tratar de convencernos de que un “cambio visual mínimo” no es más que un mero cambio. Sin embargo, debemos ser conscientes de que dicho cambio, jamás será sutil por mínimo que sea y por el contrario, será una “gran mancha” en la fisionomía de nuestra “Mona Lisa” informática. A caso ¿te animarías a hacerle una “sutil modificación” al David de Miguel Ángel?

Claramente, ninguna persona con humildad, se animaría a hacerlo. Y esa, debe ser nuestra postura frente al arte de nuestros diseñadores gráficos: respetar las GUI como si se tratara de una escultura de Miguel Ángel. Pues verdaderamente, lo son.

Pero lograr esto, jamás podría ser simple si se es ajeno al arte. Mucho menos, podremos lograrlo si continuamos empecinados en encerrarnos en la postura del “esto no se puede” o del “esto es imposible”. En la ingeniería de sistemas, no deben existir los imposibles, sino tan solo, la buena predisposición a encontrar la forma de lograrlo.

Y esa, es la responsabilidad de las vistas, para MVC: **lograr lo que se cree imposible**.

Recursos

Las vistas en MVC, básicamente cuentan con dos recursos fundamentales:

1. Las interfaces gráficas y
2. Los datos.

Sobre las interfaces gráficas, hemos hablado suficiente. Pero ¿y de los datos?

En aplicaciones basadas en MVC de la magnitud de las que planteamos en este libro, todo el estilo del sistema se basa en una organización orientada a objetos.

Dicho estilo arquitectónico, no debe confundirse con el paradigma de programación:

> *Puede existir una aplicación desarrollada con el paradigma de la programación orientada a objetos pero no puede existir un estilo arquitectónico organizado en objetos sin la implementación de su paradigma homónimo.*

El paradigma de la programación orientada a objetos, solo se trata de recursos de los cuáles dispone el programador: desde clases y objetos, hasta métodos y propiedades.

Sin embargo, puedo crear clases orientadas a datos; objetos orientados a objetos y objetos orientados a

datos. Lo anterior, es estancarse solo en la óptica del programador. Cuando esto mismo se mira como Arquitecto, nos encontramos con que la orientación a objetos puede ser más que un paradigma: puede ser la forma de organizar una aplicación. Es decir, el estilo arquitectónico que ésta implemente.

En dicho estilo, los datos en sí mismos no existen. Solo existen objetos que se relacionan entre sí. Y dichos objetos, son los que contendrán los datos (en realidad, "información" que las vistas deberán utilizar).

Y es responsabilidad de las vistas, intentar conservar hasta último momento, los objetos como tales, extraer la información que necesiten de dichos objetos y convertirla en datos a presentar en el contexto de una interfaz gráfica.

Es por ello que debe hacerse esfuerzo en entender que muchas veces, necesitaremos la información contenida en un objeto a un nivel de dependencia casi interminable.

Responsabilidades

Si analizamos lo anterior, podremos deducir que una de las responsabilidades de las vistas en MVC, será la exploración del (o los) objeto recibido desde el controlador, en busca de la información que le provea los datos necesarios. Por ejemplo:

Puede ser que necesite el “dato Z” que se extrae del objeto “Y” que pertenece al objeto “X” quien a la vez compone al objeto “W” que forma parte de la colección de objetos “W” que componen a la propiedad colectora del objeto “V” y así de forma regresiva, caminando hacia atrás como un cangrejo, hasta llegar al objeto “A”. Es entonces, que se hace necesario saber que la vista, recibirá por parte del controlador, al objeto “A” intrínseco y será ella, quien deberá explorar al objeto “A” hasta hallar el dato “Z”.

Para ello, la vista deberá conocer cuál es la relación de dependencia de los objetos y por consiguiente, los niveles de dependencia que estos manejan.

No puede pretenderse que la vista, reciba los datos “servidos en bandeja”: pues es responsabilidad de ella obtenerlos.

Y esto, es a lo que denomino *“El arte de las vistas”*. Y cada uno de los alumnos que he tenido (o los que actualmente tengo) en los cursos de MVC, pueden dar fe de ello, puesto que como he comentado en reiteradas ocasiones, siempre les digo que:

“El arte de las vistas consiste en convertir lo que se tiene en lo que se necesita”

Y aquella, es otra de las responsabilidades de las vistas.

Es decir, que las vistas cuentan con dos recursos y dos responsabilidades:

1. Hallar los datos necesarios para sustituir las GUI (explorando los objetos en busca de la información que se les provea);
2. Convertir aquello que tiene en aquello que necesita.

Problemática

Cuando nos enfrentamos a un estilo arquitectónico orientado a objetos, el problema de las vistas suele ser, paradójicamente, los propios objetos. Pues todo lo que hemos visto sobre requerimientos gráficos sustitutivos se basa en el trabajo previo realizado por las vistas. Es decir, que para llegar a ello, las vistas primero, deben haber transitado todo el camino de exploración del objeto y conversión de la información obtenida. Pues de lo contrario, los algoritmos de sustitución arrojarán errores o excepciones cuando el tipo de datos esperados no sea el apropiado.

Cuando se crea un diccionario, éste debe tener asociado a sus claves, valores de un único tipo de datos (tipo de dato simple y no colecciones). Los objetos, los arrays de PHP, las listas, las tuplas y los diccionarios de Python, son todas colecciones de datos, incluso aunque conceptualmente, se denominen “información” en el caso de los objetos.

Este tipo de problemas deben ser tenidos en cuenta por las vistas, ANTES de intentar cualquier sustitución.

Los *"coequiper"* de las vistas

Más adelante entraremos en el detalle de los controladores, pero sin embargo, se hace necesario mencionar ciertas generalidades de éstos, puesto que vistas y controladores, deberán trabajar como un verdadero equipo.

Sabemos que por cada recurso en el sistema, existirá un método en el controlador. El controlador es el encargado de solicitar a los modelos, la información requerida por las vistas y finalmente, entregarle a éstas dicha información.

Para que esto sea posible, debe existir una colaboración tácita mutua entre ambos (vista y controlador). Y por tácita me refiero a que a nivel código fuente, las vistas no conectarán al controlador, sino a que a nivel desarrollo del sistema, los recursos del controlador (métodos) deberán ser desarrollados a la par de los métodos de la vista. ¿Para qué? Para que antes de dar por finalizado un recurso en el controlador, observando

y analizando los requerimientos gráficos y lógicos de las vistas, podamos saber con precisión, que objetos debemos buscar en los modelos para ser entregados a las vistas. Pues los controladores no deben entregar "lo que se les antoje" sino aquello que las vistas realmente necesiten.

Sustituciones a nivel del *core*: la clase *Template*

Al comienzo del libro, cuando hablábamos de los archivos del núcleo de la aplicación, mencionábamos que nos enfocaríamos en *Template* cuando habláramos de las vistas. Pues ha llegado el momento.

Como habrás podido observar cuando hablábamos de los requerimientos gráficos sustitutivos, los algoritmos para realizar tanto sustituciones directas como iterativas, son bastante redundantes.

En el caso de las *sustituciones directas*, solo varía:

1. El archivo a sustituir;
2. El diccionario utilizado para la sustitución.

En las *sustituciones iterativas*, se suma, además de los dos anteriores, la expresión literal utilizada en los comentarios HTML que identifican la parte del código sobre la cual realizar la sustitución.

El *Template* a nivel del *core*, será quien defina las funciones necesarias para realizar de forma genérica dichas sustituciones para la lógica de las vistas, dejando así en éstas, la sola responsabilidad de explorar los objetos en busca de los datos que requiere y convertir dicha información, al formato necesario para que el *core* realice las sustituciones.

Dado que aquí el tratamiento difiere de forma considerable en lo que respecta a ambos lenguajes, trataré el *Template* en dos apartados diferentes: uno destinado a Python y otro a PHP.

El *Template* del core en Python

A diferencia de PHP, como veremos más adelante, en Python solo será necesaria la definición de un método para las sustituciones iterativas siendo también, muy recomendable crear una función para leer un HTML con tal solo una breve llamada sin pasar por la estructura `with`.

No obstante, Python ya dispone de una clase `Template()` como hemos visto anteriormente. Y lo que haremos, es aprovechar esto al máximo posible y heredar de ella (con un alias) reescribiendo el nombre para que de esta forma, si estamos acostumbrados a utilizar dicha clase, no notemos cambios al respecto.

Importaremos entonces la clase `Template()` bajo el alias `BaseTemplate` y escribiremos una nueva clase llamada `Template()` que herede de su original homónima, ahora, `TemplateBase()`.

Aprovecharemos este paso para ya importar también, el módulo `re`:

```
# Archivo: core/helpers/template.py
import re
from string import Template as BaseTemplate


class Template(BaseTemplate):

    def __init__(self, string):
        super(Template, self).__init__(string)
```

A continuación, crearemos el método para las sustituciones iterativas, al cual -ya que utiliza expresiones regulares- llamaremos `render_regex()`:

```
def render_regex(self, stack, key):
    dicc = dict(k=key)
    regex = re.compile(
      '<!--%(key)s-->(.|\n){1,}<!--%(k)s-->' % dicc)
    match = regex.search(self.template).group(0)
    strorig = self.template
    self.template = match
    render = ''
    for obj in stack:
        render += self.safe_substitute(vars(obj))
    render = render.replace('<!--%s-->' % key, '')
    return strorig.replace(match, render)
```

Por favor, notar que algo sumamente notorio y admirable de Python, es que no necesitamos "sanear" los diccionarios, puesto que al todo ser un objeto en Python, las problemáticas de las que hemos hablado en página 96, no lo afectan a nivel de objetos.

De esta manera, podrás seguir accediendo a los métodos de la original clase `Template()` en la misma forma que con anterioridad:

```
Template(string).safe_substitute(diccionario)
Template(string).substitute(diccionario)
```

Con la única salvedad de que ya no necesitarás importar la clase desde el módulo `string`, sino desde el *core* de tu propia aplicación:

```
from core.helpers.template import Template
```

Luego, cuando en las vistas requieras realizar sustituciones iterativas, lo harás como corresponde, invocando al método `render_regex()` de la clase `Template()`:

```
Template(string).render_regex(diccionario, sector)
```

Finalmente, pero **fuera de la clase `Template()`**, crearemos una función (un *Helper*) quien será "una especie de alias" para la estructura `with`.

Me tienta llamarla `file_get_contents()` para que resulte familiar. Pero sinceramente, en PHP crearía un alias de `file_get_contents()` llamado `get_file()` para sustituir al original. Pues `file_get_contents()` no es un nombre de función muy "feliz" puesto que parece haber sido definido por alguien que domina el inglés, menos que mi abuela. Si al menos le hubiesen puesto *"get file contents"*, sería algo redundante pero a mi criterio, menos bizarro... así que solo me limitaré a definir esta función, simplemente como `get_file()`:

```
def get_file(ruta_archivo):
    with open(ruta_archivo, 'r') as archivo:
        return archivo.read()
```

El *Template* del core en PHP

Antes de crear nuestra clase Template a nivel del *core* (`core/helpers/template.php`) es necesario considerar algunos aspectos a tener en cuenta para sumar valor a nuestros algoritmos y anticiparnos a que el error nos desespere. Y estos están dados por las problemáticas de las que hablamos en la página 96.

Es por ello entonces, que en los métodos de nuestra clase `Template()`, antes de decidir realizar una

sustitución, tendremos que revisar los diccionarios y eliminar todo dato representado por un objeto o cualquier otro tipo de colección, como es el caso de los *array*.

Como hemos visto anteriormente, en Python esta tarea es más que simple, ya que el lenguaje en sí mismo dispone de forma nativa de una clase que nos soluciona gran parte de nuestros requerimientos. En cambio, en PHP, se hace mucho más complejo. Pero que esto, no te haga mirar con mala cara a PHP. Después de todo, leemos el HTML con tan solo un `file_get_contents()` ¿o no? :)

En principio, dos métodos serán necesarios:

1. Un método para las sustituciones directas. Lo llamaremos `render()`;
2. Otro método para las sustituciones iterativas. Este último, deberá recurrir al anterior de forma cíclica. Ya que utilizará expresiones regulares, lo llamaremos `render_regex()`.

Finalmente, un tercer método es aconsejable: aquel que se encargue de configurar los diccionarios de forma adecuada. Cabe destacar además, que será una buena práctica "sanear" los diccionarios recibidos desde las vistas, para la cuál crearemos un método específico. Y dado que todos los métodos

anteriores, tarde o temprano requerirán de estos dos últimos, será por ellos por quienes comencemos, pero no sin antes definir la clase:

```
# Archivo: core/helpers/template.php
Class Template {

    function __construct() { }
}
```

Ahora sí, veremos como sanear los diccionarios. Para lograr esto, primero debemos asegurarnos convertir la colección recibida, en un verdadero diccionario. Luego, lo recorreremos para asegurarnos que todos los valores asociados a las claves, efectivamente no sean del tipo colección y en caso contrario, los eliminaremos. Haremos este método privado, ya que solo servirá a fines internos de la clase `Template()`:

```
private function sanear_diccionario(&$dict) {
    settype($dict, 'array'); $dict2 = $dict;
    foreach($dict2 as $key=>$value) {
      if(is_object($value) or is_array($value)) {
          unset($dict[$key]);
      }
    }
}
```

Notar que el método anterior modificará el diccionario por referencia.

Crearemos ahora el método encargado de encerrar entre llaves de apertura y cierre a las claves del diccionario. Esto permitirá que en sustituciones directas, por ejemplo, las vistas nos entreguen directamente un objeto.

Notar que al trabajar directamente con objetos, será indispensable que los identificadores de sector en los archivos HTML, se indiquen con el nombre de las propiedades del objeto. En caso de objetos diferentes a ser sustituidos en la misma vista cuyas propiedades presenten nombres idénticos, las vistas serán las encargadas de convertir esos diccionarios generando las claves faltantes y en los HTML, se deberán utilizar identificadores diferentes al nombre de la propiedad.

```
private function set_dict(&$dict) {
    $this->sanear_diccionario($dict); $dict2 = $dict;
    foreach($dict2 as $key=>$value) {
      $dict["{{$key}}"] = $value;
      unset($dict[$key]);
    }
}
```

Ahora, crearemos el método para efectuar las sustituciones directas. Para ello, haremos que primero el constructor de la clase reciba como parámetro, la cadena a sustituirse y la almacene en una propiedad clase:

```
# Modificación al método constructor
function __construct($str) {
    $this->template = $str;
}

function render($dict) {
    $this->set_dict($dict);
    return str_replace(array_keys($dict),
      array_values($dict), $this->template);
}
```

Finalmente, crearemos el método encargado de las sustituciones iterativas. Se debe tener en cuenta que este método recibirá, por lo general, propiedades colectoras (un *array* de objetos):

```
function render_regex($dict, $id) {
    $regex = "/<!--$id-->(.|\n){1,}<!--$id-->/";
    preg_match($regex, $this->template, $matches);
    $strorig = $this->template;
    $this->template = $matches[0];
    $render = '';
    foreach($dict as $possible_obj) {
      $render .= $this->render($possible_obj);
    }

    $render = str_replace("<!--$id-->", '', $strorig);
    return str_replace($this->template, $render, $strorig);
}
```

Capítulo VII: Controladores. Los «*coequipers*» de las vistas

En este capítulo abarcaremos los controladores más en detalle pero tal como decía mi papá, como *«en la vida las cosas no son compartimentos estancos aislados unos de los otros»*, lo haremos en paralelo a las vistas. Y si crees que sobre las vistas ya hemos dicho mucho, créeme que aún falta lo más interesante: la creación de los objetos `View`.

Características básicas de un Controlador y su anatomía

Cuando hablábamos de `FrontController`, mencionábamos que los Controladores debían estar preparados para recibir a través de sus métodos constructores, dos parámetros:

1. **`recurso:`** que representaría el nombre de su propio método al cuál debería realizar una llamada de retorno;
2. **`argumento`**, que opcionalmente algunos de los métodos como por ejemplo, editar, eliminar o ver (que son requeridos por

muchos objetos) podrían necesitar;

En el caso de Python, además, debería estar preparado para recibir un tercer parámetro:

3. **`environ`:** el diccionario entregado por WSGI al *application*. Este diccionario sería sumamente importante, a la hora de obtener los datos enviados por el usuario. Ya sea a través del método `GET` (que difícilmente lo vayas a utilizar en MVC) o como más comúnmente sucede, a través del método `POST`.

A la vez, cada método constructor, como comenté en el punto 1, debería hacer una llamada de retorno al recurso (método) correspondiente. Y en el caso particular de Python, almacenar el resultado de las vistas, en una propiedad de clase, a la cuál el `FrontController` pudiese acceder para que *application* la retornase a WSGI y éste, finalmente la imprimiera (hay que reconocer que en Python, MVC en este aspecto, es como un empleado del Estado: burocrático como pocos).

Se hace necesario aclarar, que todo controlador, antes de realizar la llamada de retorno a su propio método, deberá verificar que el recurso solicitado

por el usuario, realmente es un recurso existente.

Construyendo un controlador

Como se puede observar, en definitiva los métodos constructores de todos los controladores, serán idénticos. Entonces ¿por qué no crear una clase *Controller* a nivel del *core* y que todos los controladores hereden de ella? ¡Hagámoslo!

Bienvenida la clase *Controller* al *core*

Dado que nadie se opone a ello (o cuando se opongan, yo ya habré terminado de publicar este libro), vamos a crear el controlador a nivel del *core*. Y así, le damos la bienvenida a la clase `Controller` a nuestro núcleo:

En Python:

```
# Archivo: core/controller.py
class Controller(object):

    def __init__(self, recurso, arg, env):
      self.output = ''  # para almacenar las vistas
      if hasattr(self, recurso):
         self.output = getattr(self, recurso)(arg, env)
      else:
         self.output = 'Recurso inexistente'
```

En PHP:

```
# Archivo: core/controller.php
class Controller {

    function __construct($recurso, $arg) {
      if(method_exists($this, $recurso)) {
         call_user_func(array($this, $recurso), $arg);
      } else {
         print 'Recurso inexistente';
      }
    }

}
```

Preparación de recursos: los métodos del controlador

En el controlador de prueba que creamos al comienzo, ya habíamos hablado de que por cada modelo debía existir un controlador. Así que siempre, el primer e ineludible paso, es al menos, crear el archivo con su clase correspondiente que ahora, heredará de la clase `Controller()`.

Recordar que en los controladores, siempre se debe importar al modelo y a la vista correspondiente.

Dejaremos preparados los archivos de los controladores para nuestros modelos `MateriaPrima` y `Producto`:

En Python:

```
# Archivo:
# modules/produccion/controllers/materia_prima.py
from core.controller import Controller
from modules.produccion.models.materia_prima import MateriaPrima
from modules.produccion.views.materia_prima import MateriaPrimaView

class MateriaPrimaController(Controller):

    def __init__(self):
        super(Controller).


# Archivo:
# modules/produccion/controllers/producto.py
from core.controller import Controller
from modules.produccion.models.producto import Producto
from modules.produccion.views.producto import ProductoView

class ProductoController(Controller):
    pass
```

En PHP:

```
# Archivo:
# modules/produccion/controllers/materia_prima.php
require_once 'core/controller.php';
require_once 'modules/produccion/models/materia_prima.php';
require_once 'modules/produccion/views/materia_prima.php';


class MateriaPrimaController extends Controller {
}


# Archivo:
# modules/produccion/controllers/producto.php
require_once 'core/controller.php';
require_once 'modules/produccion/models/producto.php';
require_once 'modules/produccion/views/producto.php';


class ProductoController extends Controller {
}
```

Una vez armadas las clases, lo que se debe tener en cuenta antes de desarrollar los métodos, es lo siguiente:

Habrá un método por recurso:

Esto significa que un controlador, como mínimo, tendrá un método por cada recurso relativo al modelo. Los recursos, en principio, serán diseñados partiendo de los requerimientos visuales de la aplicación. Por ejemplo, si se desea mostrar al usuario un formulario para "agregar nueva materia prima al sistema" (traducido a "lenguaje programador": crear un nuevo objeto `MateriaPrima`), existirán en realidad, un mínimo de dos recursos para dicho requerimiento:

1) un recurso destinado a mostrar el formulario para ingreso de datos;
2) otro recurso para crear un objeto `MateriaPrima` persistente.

Dependiendo del requerimiento visual de la aplicación para estos dos recursos, el controlador, podrá necesitar otro u otros métodos relacionados. Por ejemplo ¿qué se requiere tras guardar el objeto? Se puede requerir que un mensaje de confirmación sea mostrado el usuario. Entonces, aquí tendremos un nuevo recurso.

Cada método llevará el nombre del recurso que será utilizado en la URI:

Siguiendo el ejemplo anterior, tendríamos tres recursos:

- *Mostrar el formulario para ingreso de datos*
  - URI: `/producción/materia-prima/agregar`
  - Método: `agregar()`
- *Guardar el objeto*
  - URI: `/producción/materia-prima/guardar`
  - Método: `guardar()`
- *Mostrar mensaje de confirmación*
  - URI: `/producción/materia-prima/confirmar`
  - Método: `confirmar()`

Se hace necesario mencionar que el nombre de un recurso es de elección totalmente libre. Solo habrá que tener la precaución de adoptar medidas de conversión, en caso que los nombres de los recursos sean palabras compuestas.

Por ejemplo:

Si para el tercer recurso del ejemplo quisiéramos ser más descriptivos y optáramos por una frase como ***"confirmar ingreso"***, a nivel **URI** deberíamos transportarlo como **`confirmar-ingreso`** y en cuanto al **método**, como **`confirmar_ingreso`**. En estos casos, el constructor de la clase `Controller`, debería estar preparado para realizar la conversión pertinente.

En Python:

```
def __init__(self, recurso, arg, env):
    self.output = ''
    metodo = recurso.replace('-', '_')
    if hasattr(self, metodo):
      self.output = getattr(self, metodo)(arg, env)
    else:
      self.output = 'Recurso inexistente'
```

En PHP:

```
function __construct(recurso, arg) {
    $metodo = str_replace('-', '_', $recurso);
    if(method_exists($this, $metodo)) {
      call_user_func(array($this, $metodo), $arg);
    } else {
      print 'Recurso inexistente';
    }
}
```

Ten en cuenta que los nombres de los recursos y por consiguiente, los nombres de los métodos del controlador, deben ser descriptivos para el usuario y no necesariamente "técnicos".

El rol de los controladores frente a diversos requerimientos gráficos y la construcción de objetos *View*

Vamos ahora a ir completando nuestros controladores para los modelos `MateriaPrima` y `Producto` al tiempo que nos servirá para:

1) comprobar la forma en la cuál los controladores trabajan a la par de las vistas, como un verdadero equipo;
2) ver cómo los controladores responden a los diferentes requerimientos gráficos.

Preparación de los objetos *View*

Antes de continuar, dejaremos las clases para los objetos *View* preparadas así luego, solo nos enfocamos en sus métodos.

En Python:

```
# Archivo: produccion/views/materia_prima.py
from settings import STATIC_DIR
from core.helpers.template import Template, get_file


class MateriaPrimaView:
```

```
    def __init__(self):
        pass


# Archivo: produccion/views/producto.py
from settings import STATIC_DIR
from core.helpers.template import Template


class ProductoView:

    def __init__(self):
        pass
```

En PHP:

```
# Archivo: produccion/views/materia_prima.php
require_once 'settings.php';
require_once 'core/helpers/template.php';


class MateriaPrimaView {

    function __construct() {
    }

}


# Archivo: produccion/views/producto.php
require_once 'settings.php';
require_once 'core/helpers/template.php';


class ProductoView {

    function __construct() {
    }

}
```

Caso práctico 1: requerimiento gráfico directo

Es el caso de la mayoría de los recursos de tipo *agregar*, donde el objeto a crear se trata de un objeto simple, no dependiente de ningún otro, para el cual solo es necesario mostrar un formulario para ingreso de datos. En nuestro ejemplo, estaríamos hablando del recursos *agregar* del objeto `MateriaPrima`.

En estos casos, la vista no tiene más que "comentarle" al controlador, que ella se encargará de todo. Siendo así, el controlador, no tiene más que avisar a la vista de que está siendo solicitada.

Recordemos que siempre debemos tener las GUI antes de desarrollar las vistas. El controlador puede ir preparándose mientras la lógica de la vista aguarda a las GUI. Sin embargo, no podrá estar completo hasta tanto las vistas reciban las GUI. Por este motivo, a lo largo del libro -y a partir de este momento- siempre comenzaremos desarrollando una GUI sencilla a modo de ejemplo.

URI:

```
http://myapp.net/produccion/materia-prima/agregar
```

La GUI:

```
<!-- Archivo: /static/html/mp/agregar.html -->
<h1>Agregar nueva materia prima</h1>

<form
  method='POST'
  id='mp_agregar'
  enctype='text/plain'
  action='/produccion/materia-prima/guardar'>

    <label for='mp'>Nombre de la materia
      prima:</label><br/>
    <input type='text' name='mp' id='mp'/><br/>
    <input type='submit' value='Agregar'/>

</form>
```

Código Python:

```
# La vista
def agregar(self):
    return get_file('%shtml/mp/agregar.html' % STATIC_DIR)


# El controlador
def agregar(self, *args):
    view = MateriaPrimaView()
    return view.agregar()
```

Código PHP:

```
# La vista
function agregar() {
    print file_get_contents(
      STATIC_DIR . 'html/mp/agregar.html');
}


# El controlador
function agregar() {
    $view = new MateriaPrimaView();
    $view->agregar();
}
```

Los recursos de este tipo, generalmente producen recursos de acción asociados. En este caso, el recurso guardar. Los recursos de acción asociados, suelen estar a cargo exclusivamente del controlador. Siendo así, dicho recurso no tendrá una presentación gráfica directa y por lo tanto, deberá redirigir al usuario, a un recurso diferente, pero no necesariamente de forma literal, sino que tan solo, podría encargarse de hacer una llamada a otro método.

Tendremos entonces, el caso del recurso `guardar` quien mostrará al usuario la confirmación mediante una llamada al recurso `confirmar-ingreso` que veremos en el caso práctico 2.

URI:

```
http://myapp.net/produccion/materia-prima/guardar
```

Código Python:

```
# El controlador
def guardar(self, *args):
    env = args[1]
    data = env['wsgi.input'].read().split('\r\n')
    model = MateriaPrima()
    model.denominacion = data[0].split('=')[1]
    model.save()
    self.confirmar_ingreso(model.materiaprima_id)
```

Código PHP:

```
# El controlador
function guardar() {
    $model = new MateriaPrima();
    $model->denominacion = $_POST['mp'];
    $model->save();
    $this->confirmar_ingreso($model->materiaprima_id);
}
```

Si se quisiera redirigir al usuario, literalmente hablando, solo bastaría con reemplazar la última línea del método, por la siguiente instrucción:

```
# En Python:
oid = model.materiaprima_id;
self.output = ('Refresh',
 '0; /produccion/materia-prima/confirmar-ingreso/%i' % oid)


# En PHP:
$id = $model->materiaprima_id;
header(
  "Location: /produccion/materia-prima/confirmar-ingreso/$id");
```

Por favor, notar que los datos recibidos desde el formulario, se están asignando de forma directa al objeto, sin pasar previamente por acciones de aseguración y saneamiento. Estos datos, tanto en el caso de Python como en el de PHP, deben ser saneados y asegurados desde el controlador, de forma directa o a través de un Helper, antes de ser asignados al objeto, a fin de garantizar la seguridad de la aplicación.

Caso práctico 2: sustitución directa

La sustitución directa se suele dar en escasos recursos. El más frecuente, es el caso de mensajes de confirmación en respuesta a una determinada acción. El segundo caso, pero menos frecuente, es cuando se debe mostrar cualquier objeto que no posea dependencias. Este tipo de situación, suele darse en el recurso *editar* de este tipo de objetos. Luego, el siguiente uso más frecuente, es independiente a los recursos: es el caso de plantillas HTML para unificación del diseño gráfico.

Tomaremos aquí, el caso del recurso `confirmar-ingreso` de `MateriaPrima`.

URI:

```
http://myapp.net/produccion/materia-prima/confirmar-ingreso
```

GUI para Python:

```
<!-- Archivo: /static/html/mp/confirmar.html -->
<h1>Materia prima guardada con éxito</h1>
<p>Se ha guardado la materia prima $denominacion.</p>
<p>Código de identificación: $materiaprima_id</p>
```

Código Python:

```
# La vista: necesito los datos del objeto
def confirmar(self, obj):
    string = get_file('%shtml/mp/confirmar.html' % STATIC_DIR)
    return Template(string).safe_substitute(vars(obj))
```

```
# En la zona de imports del controlador
from core.helpers.patterns import make


# El controlador: necesito traer el objeto
def confirmar_ingreso(self, oid=0, *args):
    model = make(MateriaPrima, int(oid))
    view = MateriaPrimaView()
    return view.confirmar(model)
```

GUI para PHP:

```
<!-- Archivo: /static/html/mp/confirmar.html -->
<h1>Materia prima guardada con éxito</h1>
<p>Se ha guardado la materia prima {denominacion}.</p>
<p>Código de identificación: {materiaprima_id}</p>
```

Código PHP:

```
# La vista: necesito los datos del objeto
function confirmar($obj) {
    $str = file_get_contents(
      STATIC_DIR . 'html/mp/confirmar.html');
    print new Template($str)->render($obj); # PHP 5.4
    /* En versiones anteriores a PHP 5.4 utilizar:
      $tmpl = new Template($str);
      print $tmpl->render($obj); */
}


# En la zona de importaciones del controlador
require_once 'core/helpers/patterns.php';


# El controlador: necesito traer el objeto
function confirmar_ingreso($id=0) {
    $model = make('MateriaPrima', $id);
    $view = new MateriaPrimaView();
    $view->confirmar($model);
}
```

Caso práctico 3: sustitución iterativa

Este caso mayormente se da cuando se necesita mostrar una colección de objetos del mismo tipo. Por lo general, sucede en los recursos *agregar* de un objeto compuesto al que se le debe seleccionar su compositor, aunque igual de frecuente, es el recurso *listar*, donde se debe mostrar la colección completa de objetos de un mismo tipo.

Frente a estos requerimientos, siempre será necesario efectuar una llamada al método `get()` del objeto colector. Por lo tanto, antes de continuar, se recomienda crear el objeto **`MateriaPrimaCollection`** en el modelo **`materia_prima.py|php`** como se muestra a continuación.

En Python:

```
from core.helpers.patterns import compose, make


class MateriaPrimaCollection(object):

    __materiaprimacollection = None

    def __new__(cls):
        if cls.__materiaprimacollection is None:
            cls.__materiaprimacollection = super(
              MateriaPrima, cls).__new__(cls)
            return cls.__materiaprimacollection

    def __set(self):
        self.__materiaprimacollection.objetos = []
```

```
    def __add_objeto(self, objeto):
        self.__materiaprimacollection.objetos.append(
            compose(objeto, MateriaPrima))

    def get(self):
        self.__objectcollection.__set()

        sql = """SELECT materiaprima_id
               FROM materiaprima"""
        fields = run_query(sql)

        for field in fields:
            self.__objectcollection.__add_objeto(
              make(field[0], Objeto))

        return self.__objectcollection
```

En PHP:

```
class MateriaPrimaCollection {

    private static $materiaprimacollection;

    private function __construct() {
        $this->materiaprima_collection = array();
    }

    private function add_object(MateriaPrima $obj) {
        $this->materiaprima_collection[] = $obj;
    }

    public static function get() {
        if(empty(self::$materiaprimacollection)) {
            self::$materiaprimacollection =
                      new MateriaPrimaCollection();
        }

        $sql = "SELECT materiaprima_id
                FROM materiaprima
                WHERE materiaprima_id > ?";
        $data = array("i", "0");
        $fields = array("materiaprima_id" => "");
        DBObject::ejecutar($sql, $data, $fields);
```

```
        foreach(DBObject::$results as $array) {
            self::$materiaprimacollection->add_object(
              make('MateriaPrima',
                    $array['materiaprima_id']));
        }

        return self::$materiaprimacollection;
    }

}
```

Una vez creado el colector, tomaremos como ejemplo de sustitución iterativa al recurso *listar* del objeto `MateriaPrima`.

URI:

```
http://myapp.net/produccion/materia-prima/listar
```

GUI para Python:

```
<!-- Archivo: /static/html/mp/listar.html -->
<h1>Listado de materias primas</h1>
<p>La siguiente tabla, muestra el catálogo completo de
materias primas agregadas en el sistema. Puede <a
href='/produccion/materia-prima/agregar'><b>agregar una
nueva</b></a> si lo desea.</p>

<table>
  <thead>
    <tr>
      <th>Código</th>
      <th>Denominación</th>
    </tr>
    <!--LISTADO-->
    <tr>
      <td>$materiaprima_id</td>
      <td>$denominacion</td>
    </tr>
```

```
    <!--LISTADO-->
  </thead>
</table>
```

Código Python:

```
# La vista: necesito una colección de objetos
def listar(self, collection):
    string = get_file('%shtml/mp/listar.html' % STATIC_DIR)
    return Template(string).render_regex('LISTADO',
      collection)


# El controlador
# en el sector de imports agregar:
from modules.produccion.models.materia_prima import \
    MateriaPrimaCollection


# Necesito traer la colección
def listar(self, *args):
    collection = MateriaPrimaCollection().get()
    view = MateriaPrimaView()
    return view.listar(
             collection.materiaprima_collection)
```

GUI para PHP:

```
<!-- Archivo: /static/html/mp/listar.html -->
<h1>Listado de materias primas</h1>
<p>La siguiente tabla, muestra el catálogo completo de
materias primas agregadas en el sistema. Puede <a
href='/produccion/materia-prima/agregar'><b>agregar una
nueva</b></a> si lo desea.</p>

<table>
  <thead>
    <tr>
      <th>Código</th>
      <th>Denominación</th>
    </tr>
    <!--LISTADO-->
```

```
    <tr>
      <td>{materiaprima_id}</td>
      <td>{denominacion}</td>
    </tr>
    <!--LISTADO-->
  </thead>
</table>
```

Código PHP:

```
# La vista: necesito una colección de objetos
function listar($collection) {
    $str = file_get_contents(
      STATIC_DIR . 'html/mp/listar.html');
    print new Template($str)->render_regex($collection,
       'LISTADO'); # PHP 5.4
}


# El controlador: necesito traer la colección
function listar() {
    $collection = MateriaPrimaCollection::get();
    $view = new MateriaPrimaView();
    $view->listar(
      $collection->materiaprima_collection);
}
```

Caso práctico 4: sustitución combinada

Es frecuente encontrar este tipo de sustituciones, en objetos persistentes que poseen una o más propiedades colectoras. Podría ser el caso de nuestro objeto `Producto` quien cuenta con la propiedad colectora `materiaprima_collection`.

En estos casos, la sustitución combinada estará dada por:

1. Sustitución iterativa: en lo que respecta a la

propiedad colectora;

2. Sustitución directa: en lo que respecta a las propiedades simples del objeto.

Estos requerimientos son tradicionalmente observados en recursos como *editar* y *ver* (o *mostrar*). Nos enfocaremos aquí en el recurso *ver* del objeto `Producto`. Así que esta vez, el **controlador** y la **vista** que modificaremos, será **`producto.py|php`** y NO, `materia_prima`.

URI:

```
http://myapp.net/produccion/producto/ver/PRODUCTO_ID
```

Por ejemplo, para ver el producto con ID 15, la URI sería: `http://myapp.net/produccion/producto/ver/15`

GUI para Python:

```
<!-- Archivo: /static/html/producto/ver.html -->
<h1>$denominacion</h1>
<p>Código de producto: $producto_id</p>
<h2>Composición</h2>
<p>El producto <b>$denominacion</b> se encuentra
compuesto por las siguientes materias primas:</p>

<ul>
  <!--MP-->
    <li>$denominacion ($materiaprima_id)</li>
  <!--MP-->
</ul>
```

Código Python:

```
# La vista: necesito el objeto Producto
def ver(self, obj):
    string = get_file('%shtml/producto/ver.html')
    mp = Template(string).render_regex('MP',
      obj.materiaprima_collection)
    obj.materiaprima_collection = None
    return Template(mp).safe_substitute(obj)
# El controlador: necesito traer el objeto
def ver(self, id=0, *args):
    self.model.producto_id = id
    self.model.get()
    self.view.ver(self.model)
```

GUI para PHP:

```
<!-- Archivo: /static/html/producto/ver.html -->
<h1>{denominacion}</h1>
<p>Código de producto: {producto_id}</p>
<h2>Composición</h2>
<p>El producto <b>{denominacion}</b> se encuentra
compuesto por las siguientes materias primas:</p>

<ul>
  <!--MP-->
    <li>{denominacion} ({materiaprima_id})</li>
  <!--MP-->
</ul>
```

Código PHP:

```
# La vista: necesito el objeto Producto
function ver($obj) {
    $str = file_get_contents(
      STATIC_DIR . 'html/producto/ver.html');
    $mp = new Template($str)->render_regex('MP',
       $obj->materiaprima_collection); # PHP 5.4
    unset($obj->materiaprima_collection);
    print new Template($str)->render($obj);
}
```

```
# El controlador: necesito traer el objeto
function ver($id=0) {
    $this->model->producto_id = $id;
    $this->model->get();
    $this->view->ver($this->model);
}
```

Requerimientos gráficos especiales

En toda aplicación existen sin dudas, decenas y tal vez cientos de requerimientos gráficos especiales, que la mayoría de las veces, la bibliografía no contempla. Sin embargo, no existe requerimiento tan especial que no pueda ser resuelto con algunas de las técnicas aquí explicadas o la combinación de dos o más de ellas.

Si se tuviesen que abarcar cada uno de los posibles requerimientos "especiales" de una aplicación, sucederían dos cosas:

1) Ni yo podría comprar la versión impresa de mi libro, porque la cantidad de papel elevaría el costo por las nubes;
2) Me presentaría frente a Benedicto XVI y le diría: "Señor, disculpe, pero usted está equivocado. Dios soy yo. Ergo, Dios no es católico".

Puedo tener un humor aún más patético, pero sería demasiado. Lo cierto es, que se hace imposible predecir todos los requerimientos gráficos que pueden darse en una aplicación. Pues es sabido que *"la creatividad no tiene límites"*. Y por consiguiente, tampoco lo tienen las ideas.

En este apartado, describiré como "requerimientos gráficos especiales" a aquellos por los cuáles, más me han consultado mis alumnos y programadores en general. Sin embargo, cuando leas y practiques estos códigos y hagas una retrospectiva de todo lo visto hasta ahora, lograrás darte cuenta que todos -absolutamente todos- los requerimientos gráficos de una aplicación, con más o menos líneas de código, se resuelven con las mismas técnicas.

Me concentraré en los tres casos por los cuáles, mayor cantidad de consultas he recibido. Estos son:

- Sustituciones para más de un objeto en la misma GUI (sobretodo, cuando los objetos comparten el mismo nombre de propiedades): Caso práctico 5.

- MVC y AJAX. El que más escozor me genera. ¿Por qué? Porque AJAX no es más que una técnica mediante la cuál, con el uso de

JavaScript, se realiza una petición en segundo plano, sin necesidad de refrescarle la vista actual al usuario. Ergo, MVC no sufre cambios. Utilizas AJAX de la misma forma que utilizas el navegador. El problema real, es que la mayoría de programadores que se han iniciado en el mundo del desarrollo de Software en los último 5 o 6 años, en realidad, no saben JavaScript ni AJAX. Solo conocen JQuery que no es más que una librería desarrollada en JavaScript. La razón por la cual muchos programadores encuentran a AJAX como un requerimiento gráfico especial en MVC, es simplemente porque no saben JavaScript. Por ello, en el caso práctico 6, me limitaré a mostrar un ejemplo utilizando JavaScript en crudo, sin librerías (y con tan solo un par de líneas de código).

- Incorporación de sustituciones previas en una plantilla HTML general. Este caso es el más habitual y a la vez, el más sencillo de todos. Sin embargo, es uno de los que más dudas genera: Caso práctico 7.

Caso práctico 5: sustituciones de objetos y polimorfismo

Sucede mayormente en los casos donde más de una sustitución directa es requerida y los objetos a ser sustituidos poseen mismo nombre de propiedades. Para evitar la colisión de éstas durante la sustitución, se deben emplear medidas extraordinarias.

En nuestro ejemplo, podríamos tener el caso de un objeto `MateriaPrima` y un objeto `Producto` que deban ser visualizados en la misma GUI.

Se supone habrá dos sustituciones directas. Si los HTML conservaran como identificadores de sector, los verdaderos nombres de las propiedades, al momento de realizar la sustitución de la propiedad `denominacion` entraríamos en conflicto, puesto que ambos objetos poseen una propiedad con el mismo nombre. En estos casos, la técnica más simple consiste en:

1. Utilizar identificadores de sector alternativos;
2. Regenerar los diccionarios en las vista.

GUI para Python:

```
<!-- Archivo: /static/html/dashboard.html -->
<h1>Último producto agregado</h1>
```

```
<p><b>$producto.denominacion</b></p>
<p>Código de producto: $producto_id</p>

<h1>Última Materia prima agregada</h1>
<p><b>$mp.denominacion</b></p>
<p>Código interno: $materiaprima_id</p>
```

GUI para PHP:

```
<!-- Archivo: /static/html/dashboard.html →
<h1>Último producto agregado</h1>
<p><b>{producto.denominacion}</b></p>
<p>Código de producto: {producto_id}</p>

<h1>Última Materia prima agregada</h1>
<p><b>{mp.denominacion}</b></p>
<p>Código interno: {materiaprima_id}</p>
```

De esta forma, la vista solo debería regenerar el diccionario.

Código Python:

```
# La vista: necesitaré dos objetos (producto y mp)
def dashboard(self, prod, mp):
    string = get_file('%shtml/dashboard.html')
    dicc = {}
    dicc['producto.denominacion'] = prod.denominacion
    dicc['producto_id'] = prod.producto_id
    dicc['mp.denominacion'] = mp.denominacion
    dicc['materiaprima_id'] = mp.materiaprima_id
    return Template(string).safe_substitute(dicc)
```

Código PHP:

```
# La vista: necesitaré dos objetos (producto y mp)
```

```
function dashboard($prod, $mp) {
    $str = file_get_contents(
      STATIC_DIR . 'html/dashboard.html');
    $dict = array();
    $dict['producto.denominacion'] = $prod->denominacion;
    $dict['producto_id'] = $prod->producto_id;
    $dict['mp.denominacion'] = $mp->denominacion;
    $dict['materiaprima_id'] = $mp->materiaprima_id;
    print new Template($str)->render($dict);
}
```

El controlador, solo deberá encargarse de obtener ambos objetos. Si será el controlador de `Producto` o de `MateriaPrima`, dependerá de cuál sea el modelo y/o el recurso sobre el cual se esté trabajando.

Podría tratarse, por ejemplo, tanto de una vista principal para el módulo como de un estadístico visual a mostrar en alguno de los recursos de un determinado modelo.

Por ejemplo, si esto si quisiera mostrar en el recurso *agregar* de `Producto`, el controlador de `Producto` debería encargarse de obtener ambos objetos. Pero más adelante, cuando veamos como reutilizar los recursos MVC para crear una API *RESTFul*, veremos que simple resultará a los controladores de un modelo, solicitar datos de un modelo diferente a otro controlador. De esta forma, se mantiene la independencia de ambos.

Caso práctico 6: manipulación gráfica con JavaScript mediante AJAX

Hemos llegado al tan esperado -por muchos de ustedes, ¡lo apuesto!- caso práctico Nº6 y el tan preocupante AJAX.

Primero que nada, aclarar ¿qué es AJAX? AJAX NO es JQuery. AJAX es una técnica que consiste en enviar una solicitud HTTP en segundo plano (a "escondidas" del usuario), capturar la respuesta HTTP de dicha solicitud y mostrarla al usuario sin "moverlo" de la URI en la que se encontraba al momento de realizar dicha solicitud.

No soy experta en JavaScript ni nunca lo fui. Sí, he programado en JavaScript, muchos años antes de la aparición de JQuery. Por lo tanto, el ejemplo que colocaré aquí, seguramente podría ser mejorado con mejores técnicas. No obstante, es la base de implementación de la técnica AJAX.

Las solicitudes HTTP en segundo plano, se efectúan a través del objeto **`XMLHttpRequest`** de JavaScript. A decir verdad, es una API (interfaz) disponible en los navegadores con soporte de JavaScript.

Como todo objeto, tiene sus propiedades, métodos y eventos. Uno de ellos, es el evento **`onreadystatechange`** que, como su nombre lo indica, es el que se produce al cambiar el estado de

la solicitud. Dicho evento, es mediante el cual, se activará el "cambio" en la vista del usuario. Es decir, durante este evento será invocada la función (o argumentos) que se encarguen de capturar la respuesta HTTP recibida, procesarla y mostrarla al usuario.

Entre las propiedades de este objeto, encontramos tres que son fundamentales a la hora de capturar la respuesta HTTP:

1. **`readyState`**, nos permite saber si la respuesta se encuentra lista (completada, estado 4); en curso (estado 3); al menos ya se han recibido las cabeceras HTTP (estado 2); abierta (estado 1) o inicializada (estado 0).
2. **`status`**, es quien nos dará el código de respuesta HTTP, que podría ser, entre otros, 200 (encontrado), 404 (no encontrado), 500 (error interno), etc.
3. **`responseText`**, que será quien almacene la respuesta HTTP como una cadena de texto (esta será, la que finalmente se mostrará al usuario).

Luego, entre los métodos de este objeto, nos encontramos dos, que sí o sí requeriremos para iniciar la solicitud ("abrir" la URI en segundo plano)

y enviar la solicitud correspondiente, a dicha URI. Estos métodos son:

1. **`open()`**, para abrir la URI en segundo plano.
2. **`send()`**, para enviar la solicitud a dicha URI.

Todo esto, significa que:

- En el recurso actual, a nivel gráfico, debemos tener un "lugar" reservado para escribir la respuesta. Puede ser un *layer* (capa) u otro elemento HTML.
- La solicitud tendrá que hacerse a otro recurso y lo mismo que ese recurso mostraría en el navegador si entráramos de forma directa, lo estaríamos capturando e imprimiendo en el lugar reservado para imprimir la respuesta.

O sea, es lo mismo que hacerlo sin AJAX. Y para demostrarlo, haremos lo siguiente:

¿Recuerdas el recurso *agregar* del objeto `MateriaPrima`? ¿Recuerdas que luego de guardarlo, el recurso *guardar* mostraba al usuario al recurso *confirmar-ingreso*? Pues bien. Haremos que cuando el usuario pulse el botón *"Agregar"* del formulario HTML, en vez de refrescarle la página, se muestre la respuesta en un *layer* del archivo HTML `mp/agregar.html`.

Antes de continuar, me gustaría mencionar que **los únicos cambios a realizar, son a nivel diseño gráfico** (GUI). **A nivel programación, no hay nada que modificar**.

Lo primero que haremos, entonces, será crear una función JavaScript genérica, para realizar solicitudes HTTP en segundo plano:

```
// archivo: static/js/ajax.js
function enviar(recurso, layer_id) {
    var xmlhttp;
    xmlhttp = new XMLHttpRequest();
    xmlhttp.onreadystatechange = function() {
        respuesta =  xmlhttp.readyState;
        estado =  xmlhttp.status;
        if (respuesta == 4 && estado == 200) {
            layer = document.getElementById(layer_id);
            layer.innerHTML = xmlhttp.responseText;
        } else {
            layer.innerHTML = 'Cargando...';
        }
    }
    xmlhttp.open("GET", recurso, true);
    xmlhttp.send();
}
```

Es cierto que JQuery te salva la vida en muchos aspectos. Pero ¿no son tan solo 16 líneas de código? Que, vale aclarar, podrían haber sido solo 12 si lo hubiese hecho en menos pasos.

Ahora nos toca agregar el *layer* correspondiente, en el HTML que contiene el formulario, agregar el enlace al archivo JavaScript y desde el botón de envío, hacer la llamada a la función `agregar()` de nuestro JavaScript.

```
<!-- Archivo: /static/html/mp/agregar.html -->
<h1>Agregar nueva materia prima</h1>

<script language='javascript' type='text/javascript'
  src='/static/js/ajax.js'></script>

<div id='respuesta'></div>

<form
  method='POST'
  id='mp_agregar'
  action=''>

    <label for='mp'>Nombre de la materia
      prima:</label><br/>
    <input type='text' name='mp' id='mp'/><br/>
    <input type='button' value='Agregar'
      onclick="enviar('/produccion/materia-prima/guardar',
       'respuesta');"/>

</form>
```

Bueno, sé que el título del libro es *Aplicaciones Web Modulares con MVC en Python y PHP,* pero no tengo la culpa de que AJAX, sea solo HTML y JavaScript :D

En caso que desees utilizar efectos del tipo fade-in/fade-out, imágenes de "carga en progreso" y demás cuestiones gráficas, utilizar JQuery puede ser una gran alternativa si no sabes como

implementarlo utilizando DOM y JavaScript. No obstante, no dejes de tener siempre bien presente, que todo lo que hagas, será a nivel gráfico. Las vistas, modelos, controladores así como cualquier archivo del core, jamás deben verse afectados por ello.

Por cierto: eso es todo lo necesario cuando se requiere AJAX (solo lo aclaro, por si aún queda alguna duda, la respuesta es "NO. No hace falta que busquen librerías o descarguen/enlacen JQuery". Pueden probarlo así como está. Claro que, también pueden darle unos colores bonitos con CSS).

Vale aclarar además, que en mucha bibliografía y documentación al respecto, suelen emplearse capítulos enteros a este tema. Pero como tengo la "maldita" costumbre de ver el vaso "medio lleno" y para colmo, al vaso siempre le busco la "simplicidad", lo he explicado con solo unos párrafos. ¿Para qué hacerlo ver complejo? Parafraseando a Nietzsche ¿Qué necesidad hay de esforzarse por ser oscuro? :)

Caso práctico 7: pseudo sustituciones directas

Las pseudo sustituciones directas, son claramente, sustituciones directas que a simple vista no parecen serlo. Es el caso en el cual, por ejemplo, cualquiera de los recursos anteriores que hemos visto, necesitan

ser mostrados dentro de una plantilla HTML general. Es decir, aquella que contenga el diseño gráfica base de la aplicación, incluyendo la estructura HTML completa (`head` y `body`. Este último, con un sector que identifique dónde debe ser sustituido el contenido que previamente, estábamos imprimiendo en pantalla).

En estos casos es recomendable:

- Crear una plantilla HTML general, con un sector de identificación para cada sección que vaya a contener información variable;
- Crear un método a nivel del *core* (en el *Template*), para que los métodos de todas las vistas recurran a él sin necesidad de andar redundando código.

Vamos a suponer el caso de querer incluir todas nuestras vistas anteriores en una misma plantilla HTML (excepto la del recurso confirmar-ingreso la cual no tendría sentido), en la cuál, la etiqueta `<title>` y el contenido interno del `<body>` deban ser sustituidos. Para ello, debemos crear primero el HTML.

GUI para Python:

```
<!-- Archivo: static/html/template.html -->
<!doctype html>
<html lang='es'>
  <head>
    <charset='utf-8'>
    <title>Python MVC App - $title</title>
  </head>
  <body>
    <header>
      <h1>Python MVC App</h1>
    </header>
    <section>
      $body
    </section>
  </body>
</html>
```

GUI para PHP:

```
<!-- Archivo: static/html/template.html -->
<!doctype html>
<html lang='es'>
  <head>
    <charset='utf-8'>
    <title>PHP MVC App - {title}</title>
  </head>
  <body>
    <header>
      <h1>PHP MVC App</h1>
    </header>
    <section>
      {body}
    </section>
  </body>
</html>
```

Ahora, crearemos un método de sustitución directa a nivel del *core*, al cual solo sea necesario pasarle el título y contenido para que realice la sustitución correspondiente.

En Python:

```
def show(self, titulo, contenido):
    self.template = get_file(
      '%shtml/template.html' % STATIC_DIR)
    dicc = dict(title=titulo, body=contenido)
    return self.safe_substitute(dicc)
```

En PHP:

```
function show($titulo, $contenido) {
    $this->template = file_get_contents(
      STATIC_DIR . 'html/template.html);
    $dict = array('title'=>$title, 'body'=>$contenido);
    return $this->render($dict);
}
```

De esta forma, cada uno de los métodos de las vistas, llamarán al método `show()` del objeto `Template()`, pasándole como parámetro, el contenido que actualmente imprimen en pantalla anteponiendo como argumento, la cadena de texto que sustituirá al título y lo que imprimirán finalmente, será el resultado de esa llamada.

Presentaremos el ejemplo del método `listar()` del objeto `MateriaPrimaView()`.

Código Python:

```
def listar(self, collection):
    string = get_file('%shtml/mp/listar.html')
    render1 = Template(string).render_regex('LISTADO',
```

```
      collection)
    titulo = 'Listado de materias primas'
    return Template('').show(titulo, render1)
```

Código PHP:

```
# La vista: necesito una colección de objetos
function listar($collection) {
    $str = file_get_contents(
      STATIC_DIR . 'html/mp/listar.html');
    $render1 = new Template($str)->render_regex(
      'LISTADO', $collection);
    $titulo = 'Listado de materias primas';
    print new Template('')->show($titulo, $render1);
}
```

El proceso anterior, será realizado por cada uno de los métodos de la vista, que así lo requiera.

Capítulo VIII: restricción de acceso a recursos

Otra de las consultas más frecuentes que me han hecho mis alumnos de las clases de MVC, es justamente, qué hacer *"con los login"*.

Realmente es algo muy sencillo, si se sabe con exactitud, de qué se trata esto del "manejo de

sesiones".

Desmitificando el concepto de "Sesiones"

El concepto de "sesiones", básicamente es una entelequia. No existe un concepto de "sesión" como tal y que sea independiente al concepto de *cookies*.

Las sesiones, son en realidad, un modo de persistencia de datos mediante la interacción de archivos en el propio servidor y en el ordenador del usuario (o sea, *cookies*).

El concepto de sesión, se refiere a la mecánica utilizada para generar la persistencia temporal de ciertos datos del usuario.

Dicha mecánica, consiste básicamente en:

Identificación del usuario: se identifica a cualquier usuario que ingresa a un sitio Web, mediante un identificador único (ID, que por cierto, nada tiene que ver con las bases de datos). Por ejemplo: olvidándonos de Python y PHP por un instante, imaginemos que estamos desarrollando un nuevo lenguaje de programación. Entonces, cada vez que una persona ingresa en el sitio Web, de forma automática, nuestro *script* le "inventa" un "código o cadena" de identificación, único para ese usuario.

Persistencia del identificador: el proceso de persistencia del identificador, es un proceso interno que debe realizarse en el servidor. Consiste en implementar una mecánica que permita almacenar de forma temporal, el identificador asignado al usuario, hasta que éste, abandone el sitio. La mayoría de los manejadores de sesiones o de los lenguajes como PHP, que implementan dicho concepto de forma nativa en el propio lenguaje, dividen esta mecánica en dos acciones:

1) Almacenamiento del identificador en el servidor, el cual puede imaginarse como un archivo temporal;

2) Almacenamiento del identificador en el ordenador del usuario, el cual no es más que una *cookie*.

El uso de *cookies* en el ordenador del usuario, podría evitarse si por ejemplo, en el archivo del usuario en el servidor, se guardara información diversa sobre el usuario: desde su IP hasta la versión del sistema operativo. Sin embargo, esto podría traer aparejadas consecuencias indeseables: desde una sobrecarga del servidor, hasta la colisión de usuarios que mediante un *proxy*, pudieran utilizar la misma IP. Es por este motivo, que la mejor solución encontrada hasta ahora, ha sido el uso múltiple de archivos: en el servidor y en el cliente, a modo de *Cookies*.

Esta, es la base del manejo de sesiones y nada tiene que ver con un concepto tangible ni mucho menos, con los procesos de registro de usuarios e identificación mediante el ingreso de claves privadas.

Por este motivo, hay que aprender a ***diferenciar entre el manejo de sesiones y la gestión de usuarios***.

Gestión de Usuarios vs. Manejo de Sesiones

La gestión de usuarios, debe verse como la gestión de cualquier otro objeto del sistema. Es así que si se requiere gestionar usuarios se deberá contar con un módulo de gestión de usuarios.

El usuario en sí mismo, no es más que un objeto del sistema. Es otro componente como tantos. Y como tal, tendrá su clase con métodos y propiedades como cualquier otra, su vista y controlador correspondiente.

Cuando se analiza el total de la aplicación, se puede observar que un modelo de usuarios, no puede relacionarse de forma directa, con ningún otro módulo del sistema. De allí, la necesidad de diseñar un módulo independiente para los usuarios. Incluso, aunque éste, tenga solo un modelo.

Si se describen las cualidades del objeto usuario, respetando de forma estricta la organización orientada a objetos de una arquitectura e incluso, su paradigma homónimo, el usuario no tendrá más que dos propiedades:

- El usuario es de nombre Juan
- El usuario es de apellido Pérez

El usuario como objeto, NO "es de" contraseña ni "tiene una" contraseña. El usuario, solo debe ingresar un dato privado para acceder al sistema. Dicho dato, es conocido como "contraseña" y no es más que un requerimiento gráfico a los ojos de una orientación a objetos estricta.

Por este motivo, la contraseña (la cual es un dato privado que solo el propietario debe conocer como si se tratase del PIN de su tarjeta de crédito), es el único requerimiento gráfico que requerirá un tratamiento especial en el modelo. Y éste, NO consistirá en agregar una nueva propiedad de clase, sino tan solo, en preparar al método `save()` para recibir como parámetro, dicho dato. Pues la forma más viable de hacer persistir dicho dato, es manipulando datos más allá del objeto. Y a fin de reutilizar recursos, no debe crearse una nueva tabla en la base de datos, sino, aprovechar la destinada a hacer persistir los objetos Usuario.

Por otra parte, debe hacerse notar que el objeto Usuario como tal, tendrá los mismos métodos que cualquier otro: `save()`, `get()` y `destroy()`.

Cuando un acceso restringido sea requerido, será el manejador de sesiones quien entre en juego, pero no ya manipulando objetos sino datos. Pues la restricción de acceso y el ingreso de contraseñas, no son más que requerimientos gráficos que recurren a "datos puros" para su tratamiento.

Es decir que:

Un módulo de usuarios es totalmente independiente de un manejador de sesiones.
Solo se aprovecha la base de datos utilizada para hacer persistir los objetos, a fin de evitar la sobre-saturación de recursos.

Incluso, un módulo de usuarios podría no ser necesario, si el sistema no requiere su gestión y por el contrario, solo cuenta con el requerimiento gráfico de ingresar datos determinados que coincidan con los esperados. Es el caso de restricción de acceso por un único nombre de usuario y contraseña. Para el sistema y a nivel arquitectónico, éstos, son solo datos que el "usuario" (no el programador) conoce (o ubica) como nombre de usuario y contraseña. A nivel algorítmico, son tan solo, dos cadenas de texto que deben coincidir con

las esperadas.

El modelo usuario

Si se desea entonces, contar con un módulo de gestión de usuarios, no habrá más que crear dicho módulo. Las sesiones, son un caso aparte e independiente, ya que ni siquiera interactúan con el o los objetos de este módulo. Repito: simplemente reutilizan la misma base de datos.

A tal fin, colocaré aquí un típico modelo Usuario, solo a modo de ejemplo por si la gestión de los mismos es requerida. Las vistas y controladores, serán como las de cualquier otro modelo.

En Python:

```
from core.dblayer import run_query


class Usuario(object):

    def __init__(self):
        self.usuario_id = 0
        self.nombre = ''

    def save(self, pwd):
        if self.usuario_id == 0:
            sql = """INSERT INTO usuario (nombre, pwd)
                   VALUES ('%(u)s', '%(p)s')""" % dict(
                       u=self.nombre, p=pwd)
```

```
            self.usuario_id = run_query(sql)
        else:
            sql = """UPDATE usuario
                   SET nombre = '%(u)s',
                       pwd = '%(p)s'
                   WHERE usuario_id = %(uid)i""" % dict(
                       u=self.nombre, p=pwd,
                       uid=self.usuario_id)
            run_query(sql)

    def get(self):
        sql = """SELECT usuario_id, nombre FROM usuario
                 WHERE usuario_id = %i""" % self.usuario_id
        fields = run_query(sql)[0]
        self.nombre = fields[1]

    def destroy(self):
        sql = """DELETE FROM usuario
                WHERE usuario_id = %i""" % self.usuario_id
        run_query(sql)
```

En PHP:

```
require_once 'core/dblayer.php';


class Usuario {

    function __construct() {
        $this->usuario_id = 0;
        $this->nombre = '';
    }

    function save($pwd) {
        if($this->usuario_id == 0) {
            $sql = "INSERT INTO usuario (nombre, pwd)
                    VALUES (?, ?)";
            $data = array('ss',
               "{$this->nombre}", "$pwd");
            $this->usuario_id = DBObject::ejecutar(
                                          $sql, $data);
        } else {
```

```
            $sql = "UPDATE usuario
                    SET nombre = ?, pwd = ?
                    WHERE usuario_id = ?";
            $data = array("ssi", "{$this->nombre}",
               "$pwd", "{$this->usuario_id}");
            DBObject::ejecutar($sql, $data);
        }
    }

    function get() {
        $sql = "SELECT usuario_id, nombre
                FROM usuario
                WHERE usuario_id = ?";
        $data = array("i", "{$this->usuario_id}");
        $fields = array(
             'usuario_id'=>'', 'nombre'=>'');
        DBObject::ejecutar($sql, $data, $fields);
        $this->nombre = $fields['nombre'];
    }

    function destroy() {
        $sql = "DELETE FROM usuario
                WHERE usuario_id = ?";
        $data = array("i", "{$this->usuario_id}");
        DBObject::ejecutar($sql, $data);
    }

}
```

Es necesario aclarar que cuando el controlador llame al método `save()` deberá hacerlo pasándole como parámetro, el dato correspondiente a la contraseña ya cifrado.

En Python:

```
from hashlib import md5
...
...
self.model.save(md5(password).hexdigest())
...
```

En PHP:

```
...
...
$this->model->save(md5($password));
...
```

Clasificación de las restricciones

Las restricciones de acceso que pueden existir en una aplicación, se clasifican según su ámbito de restricción. Éste puede ser:

- ***Restricción a nivel de la aplicación:*** ningún usuario accede a ninguna parte de la aplicación, sin haber ingresado correctamente su clave. En estos casos, la llamada al verificador de permiso del manejador de sesiones, se realiza directamente desde el *Front Controller*;
- ***Restricción a nivel del módulo:*** ningún usuario accede a ningún modelo del módulo, sin haber ingresado correctamente su clave. Aquí, la llamada al verificador, será también realizada desde el *Front Controller*, pero a través de un archivo de configuración especial, a nivel del módulo;
- ***Restricción a nivel del modelo:*** ningún

usuario accede a ningún recurso del modelo, sin haber ingresado correctamente su clave. En este caso, la llamada al verificador, es realizada directamente desde el controlador del modelo correspondiente;

- ***Restricción a nivel del recurso:*** ningún usuario accede a un recurso determinado, sin haber ingresado correctamente su clave. Aquí, la llamada al verificador, se realiza mediante cada uno de los recursos, cuyo acceso requiere de permisos especiales.

Restricción de acceso en los diversos lenguajes

El manejo de sesiones en Python y PHP, si bien guarda la misma lógica, es de una diferencia radical: mientras que PHP tiene un soporte nativo para el manejo de sesiones, en Python se hace necesario recurrir a alguna de las siguientes alternativas:

a) Crear la lógica del manejo de sesiones;

b) Recurrir a una herramienta de terceros.

En ambos lenguajes, será necesario construir un manejador de sesiones propio de la aplicación, sin embargo, no debe confundirse al “manejador” de sesiones con el “manejo” de sesiones. Es en este último, donde se ve reflejada la diferencia entre

ambos lenguajes.

Manejo de Sesiones en Python

Como comentaba anteriormente, para el manejo de sesiones en Python, tenemos dos alternativas:

1) Crear un *middleware* (software conector o intermediario);
2) Reutilizar alguno de los *middleware* existentes;

Crear un *middleware* propio, es una tarea realmente sencilla. Si te fijas al comienzo de este capítulo y te centras en cómo es que las sesiones son manejadas, te darás cuenta de que solo necesita crear una *cookie* con id de sesión en el cliente y grabar en el servidor los datos relacionados. Solo será necesario leer la *cookie* y acudir al archivo guardado en el servidor, para así recuperar las sesiones creadas.

Este libro no pretende ser una guía para Python. Así que dado que en PHP dicho *"middleware"* no es necesario crearlo (puesto que es nativo) y que ya existen diversos *middleware* para Python, solo me limitaré a explicar brevemente la implementación de uno de uno de ellos para así aprovechar el tiempo en

centrarnos en el manejador de sesiones -que sí tiene que ver con MVC- en vez de hacerlo en el manejo de las mismas. Para esto, citaré un ejemplo de implementación utilizando **Beaker**[9] especialmente diseñado para trabajar con WSGI.

Lo primero que debemos hacer es instalar dicho paquete. En distribuciones basadas en Debian, el paquete se encuentra disponible en los repositorios oficiales:

```
# apt-get install python-beaker
```

Pero también puede instalarse en cualquier distribución (esté o no basada en Debian) desde PyPI mediante `pip`:

```
$ pip install Beaker
```

Si lo instalamos mediante `pip`, nos aseguramos contar con la última versión disponible.

Una vez instalado, solo necesitaremos configurarlo y activarlo en nuestro `application.py`

```
from sys import path
# Importamos el middleware
from beaker.middleware import SessionMiddleware
```

9 https://beaker.readthedocs.org

```
# La función application cambia su nombre
def wsgi_app(environ, start_response):
    ruta = environ['SCRIPT_FILENAME'].replace(
      'application.py', '')
    path.append(ruta)
    from core.front_controller import FrontController

    headers = []
    salida = FrontController().start(environ)
    if isinstance(salida, tuple):
        headers.append(salida)
        salida = ''
    else:
        headers.append(('Content-Type',
             'text/html; charset=utf-8'))
    start_response('200 OK', headers)

    return salida


# Configuramos el Middleware
middleware_config = {
    'session.type': 'file',
    'session.cookie_expires': True,
    'session.auto': True,
    'session.data_dir': '/ruta/a/data/sessions'
}

# Conectamos WSGI con application mediante Beaker
application = SessionMiddleware(
      wsgi_app, middleware_config)
```

La ruta indicada en `session.data_dir` será el directorio en cuál Beaker guardará los datos de sesión del usuario mediante `pickle`. Dicho directorio deberá tener permisos de escritura para el usuario `www-data` (o usuario de Apache correspondiente). Para ello, se puede optar por crear un directorio cuyo propietario sea directamente el usuario `www-data` o sino, asignarle permisos de escritura para cualquier usuario.

```
/ruta/a/data$ mkdir sessions

# para modificar el propietario
/ruta/a/data$ chown www-data:www-data sessions

# o de lo contrario, asignar permisos de escritura
/ruta/a/data$ chmod -R 777 sessions
```

Una vez concluidos los pasos anteriores, ya estamos en condiciones, de concentrarnos en el diseño de nuestro *Session Handler*.

Diseño del *Session Handler*

Un SessionHandler, no suele tener "demasiadas vueltas". Básicamente, cuatro procesos son necesarios:

1. Verificación de los datos de acceso ingresados por el usuario
2. Creación de una sesión;
3. Verificación de la existencia de una sesión (ya iniciada);
4. Destrucción de una sesión.

Los dos primeros procesos, se llevan a cabo una única vez al momento que el usuario "se loguea". Cuando datos de acceso son requeridos, se presenta al usuario un formulario para que los ingrese. Dichos datos, se envían al proceso 1. Éste, verifica

que sean correctos. De ser así, el proceso 1 llama al proceso 2 y éste, inicia una nueva sesión.

El proceso 3, es llamado toda vez que se necesite constatar si el usuario ya ha iniciado una sesión.

El proceso 4, será llamado por alguno de los siguientes casos:

- Por el proceso 1, durante un intento de inicio de sesión fallida (por ejemplo: el usuario ingresa datos incorrectos);
- Por el usuario, cuando decida desconectarse del sistema (cerrar su sesión).

A la vez, un requerimiento visual será necesario: el que muestre el formulario para ingreso de datos. Dicha vista, será manejada por el módulo de usuarios y a fin de canalizar los recursos manteniendo coherencia lógica con la estructura anterior, será el controlador del modelo de usuarios, quien realiza la llamada a los procesos 1 y 4 del *Session Handler*.

Por ahora, nos limitaremos a crear el *Session Handler* y luego, veremos como la vista y los recursos son controlados desde el módulo de

usuarios.

Lo primero que debemos hacer ahora, es crear el un `SessionHandler` a nivel del *core*, con cuatro métodos que se encargarán de cada uno de los procesos que mencionábamos párrafos arriba:

1. `check_user_data()`

   Validación de los datos de accesos indicados por el usuario;

2. `start_session()`

   Inicialización de la sesión;

3. `check_session()`

   Verificación de sesión iniciada;

4. `destroy_session()`

   Destrucción de una sesión.

Antes de pasar al código, vamos a definir en el `settings`, una constante llamada `LOGIN_FORM`. Le asignaremos como valor la ruta al módulo de usuarios: `/usuarios/usuario/login`

Más adelante veremos como adaptar el controlador del objeto `Usuario`.

En Python:

```
# Archivo core/session_handler.py
from hashlib import md5
from settings import LOGIN_FORM


class SessionHandler(object):

    def __init__(self, environ):
        self.session = environ['beaker.session']
        self.data = environ['wsgi.input'] \
             if 'wsgi.input' in environ else ''

    def check_user_data(self):
        data = self.data.split('&')
        user = data[0].replace('user=', '')
        pwd = md5(data[1].replace('pwd=', '')).hexdigest()
        sql = """SELECT usuario_id
                 FROM usuario
                 WHERE nombre = '%(user)s'
                 AND pwd = '%(pwd)s'""" % dict(
                    user=user, pwd=pwd)
        result = run_query(sql)
        if len(result) == 1:
            userid = result[0][0]
            return self.start_session(userid)
        else:
            return self.destroy_session()

    def start_session(self, userid):
        self.session['userid'] = userid
        self.session['logueado'] = 1
        return ('Refresh', '0; /')

    def session_destroy(self):
        self.session.delete()
        return ('Refresh', '0; %s' % LOGIN_FORM)

    def check_session(self):
```

```
        if 'logueado' in self.session:
            return True
        else:
            self.session_destroy()
```

El método `check_session()` será invocado por un decorador, dentro del mismo archivo pero independiente de la clase:

```
def check_session(funcion):
    def wrapper(obj, arg, env):
        if not SessionHandler(env).check_session():
            return ('Refresh', '0; %s' % LOGIN_FORM)
        else:
            return funcion(obj, arg, env)
    return wrapper
```

Algunas notas:

¿Para qué se retorna una tupla en algunos casos? Para que application pueda efectuar una redirección en caso de necesitarlo. Recordemos que application, verifica si el dato retornado es una tupla y en ese caso, lo agrega a la lista headers que envía a start_response.

¿Para qué se necesita un decorador? Para que desde los recursos restringidos, se pueda efectuar una llamada directa y sea modificado en caso de necesitar una autenticación previa. Recordemos que los decoradores, reciben como parámetro a la función “a decorar”. Si no lo hiciéramos de esta forma, nos veríamos obligados a validar la sesión en cada uno de los recursos con acceso restringido.

En PHP:

```
# Archivo core/session_handler.php
session_start();


class SessionHandler {

    function check_user_data() {
      $user = $_POST['user'];
      $pwd = md5($_POST['pwd']);
      $sql = "SELECT usuario_id
              FROM usuario
              WHERE nombre = ? AND pwd = ?";
      $data = array('ss', $user, $pwd);
      $fields = array('userid'=>'');
      if(isset($fields['userid'])) {
         $userid = $fields['userid'];
         $this->start_session($userid);
      } else {
         $this->destroy_session();
      }
    }

    function start_session($userid) {
      $_SESSION['userid'] = $userid;
      $_SESSION['logueado'] = 1;
      header('Location: /');
    }

    function destroy_session() {
      session_destroy();
      header('Location: ' . LOGIN_FORM);
    }

    function check_session() {
      if(!isset($_SESSION['logueado'])) {
         $this->destroy_session();
      }
    }

}
```

Si estamos trabajando con versiones anteriores a PHP 5.4, podemos crear además, un alias para check_session() -fuera de la clase- a fin de poder invocar dicho método durante la instancia:

```
function SessionHandler() {
    return new SessionHandler();
}
```

De esa forma, podremos emular una llamada durante una pseudo instancia, al igual que en PHP 5.4:

```
SessionHandler()->metodo();
```

Vistas y Controladores para el *Login* y el *Logout*

A decir verdad, nuestro *logout* no va a requerir ninguna vista, aunque sí un controlador. Quien va a necesitar además del controlador, una vista, será nuestro *login*. Como de costumbre, el primer paso será crear el formulario para ingreso de datos.

```
# Archivo: static/html/login.html
<form method='POST'
action='/usuarios/usuario/ingresar'>
  <h1>Ingreso al sistema</h1>
  <label for='user'>Nombre de usuario:</label><br/>
  <input type='text' name='user' id='user'/><br/><br/>
  <label for='pwd'>Contraseña:</label><br/>
  <input type='password' name='pwd'
     id='pwd'/><br/><br/>
```

```
  <input type='submit' value='Ingresar'/>
</form>
```

En el controlador de Usuario, necesitaremos 3 recursos:

1. `login()`

   muestra el formulario para ingreso de datos;

2. `ingresar()`

   Llamará a `check_user_data()` del `SessionHandler`;

3. `logout()`

   Llamará a `destroy_session()` del `SessionHandler`.

En Python:

```
from core.session_handler import SessionHandler
...
...

def login(self, *args):
    return self.view.login()  # debemos definirlo

def ingresar(self, arg, env):
    return SessionHandler(env).check_user_data()

def logout(self, arg, env):
```

```
    return SessionHandler(env).destroy_session()
```

En PHP:

```
require_once 'core/session_handler.php';
...
...

function login() {
    $this->view->login();  # debemos definirlo
}

function ingresar() {
    SessionHandler()->check_user_data();
}
```

```
function logout() {
    SessionHandler()->destroy_session();
}
```

En la lógica de la vista, solo necesitaremos hacer un *render* del formulario de *login*, en un método llamado `login()`.

Restringiendo el Acceso

Ahora, para restringir el acceso, solo será efectuar una llamada a `check_session()` en cada recurso donde se necesitar estar *"logueado"* para ingresar.

En Python:

```
# Ejemplo de Controlador con recursos restringidos
from core.session_handler import check_session


class AlgunControlador(Controller):

    def recurso_sin_restriccion(self, *args):
        pass

    @check_session
    def recurso_restringido(self, arg, env):
        pass
```

En PHP:

```
# Ejemplo de Controlador con recursos restringidos
require_once 'core/session_handler.php';

class AlgunControlador extends Controller {

    function recurso_sin_restriccion() { }

    function recurso_restringid() {
      SessionHandler()->check_session();
    }

}
```

Capítulo IX: Aprovechamiento de MVC en las arquitecturas cliente-servidor con REST

¿Qué puede tener en común una arquitectura MVC con una cliente-servidor? Imagino que con un poco de esfuerzo, se podrán encontrar algunas (o muchas) cosas en común entre ambas. Pero como no tengo ganas de realizar dicho esfuerzo, me quedaré con "cara de poker" sin responder a mi propia pregunta :)

Pero, mi cara se transforma al momento de pensar *"¿cómo puedo aprovechar una arquitectura MVC para crear Web Services públicos, de forma automática, sin necesidad de realizar grandes cambios?"* ya que lograrlo, es sumamente simple. Solo necesitamos:

1. Una forma de reconocer que la llamada a un recurso se está realizando desde una API;

2. Servir los datos en JSON en vez de hacerlo en formato gráfico mediante una interfaz.

Lo primero, se resuelve de manera muy simple:

agregar a la URI del recurso, un dato más. Por ejemplo, podría ser "api":

```
/api/modulo/modelo/recurso/arg
```

De esa forma, solo necesitaremos preparar a nuestro `ApplicationHandler` para verificar si se trata o no de una llamada desde la API.

Para el segundo caso, solo tendremos que "avisarle" a nuestros controladores, si están siendo llamados -o no- por una API. Entonces, los recursos públicos de nuestros controladores, se encargarán de retornar "datos puros" sin interfaz gráfica. ¿Simple, no?

Cambios sobre *Application Handler*, *Front Controller* y *Application*

Un breve cambio será necesario en nuestro *Application Handler* (se resaltan en negritas).

En Python:

```
# Modificación en archivo: core/apphandler.py
...

    def analizer(cls, environ):
        uri = environ['REQUEST_URI']

        api = False
        if uri.startswith('/api'):
```

```
            api = True
            uri = uri.replace('/api', '')

        datos = uri.split('/')
        datos.pop(0)
        ...
        ...
        ...
        return (modulo, modelo, recurso, arg, api)
```

FrontController, debe ahora, estar preparado para recibir el nuevo dato de retorno:

```
# Modificación al archivo: core/front_controller.py
...
from core.api import APIRESTFul  # a crear luego

...
def start(cls, environ):
    modulo, modelo, recurso, arg, environ['api'] = \
       ApplicationHandler( ).analizer(environ)
    ...
    ...
    if environ['api']:
        return APIRESTFul(controller.output).json_data
        # Crearemos la clase APIRESTFul en un sig. paso
    else:
        return controller.output
```

Y finalmente, nuestro **Application**, se encargará de modificar el `Content-Type` de las cabeceras HTTP:

```
# Modificación al archivo: application.py
...

def wsgi_app(environ, start_response):
    ...
   ...
    headers = []
    salida = FrontController().start(environ)
```

```
    api = environ['REQUEST_URI'].startswith('/api')
    content = 'json' if api else 'html'

    if isinstance(salida, tuple):
        headers.append(salida)
        salida = ''
    else:
        headers.append(('Content-Type',
           'text/%s; charset=utf-8' % content))
    ...
    ...
```

En PHP:

```
# Modificación en archivo: core/apphandler.php
...

public static function analizer() {
    $uri = $_SERVER['REQUEST_URI'];

    $api = False;
    if(strpos('/api', $uri) === 0) {
        $api = True;
        $uri = str_replace('/api', '', $uri);
    }

    $datos = explode('/', $uri);
    ...
    ...

    return array($modulo, $modelo, $recurso,
      $arg, $api);
}
```

Ahora, preparamos al **`FrontController`** para recibir el nuevo dato:

```
 # Modificación al archivo: core/front_controller.php
...
require_once 'core/api.php';  # a crear luego

...
public static function start() {
    list($modulo, $modelo, $recurso,
    $arg, $api) = ApplicationHandler::analizer();
    ...
    ...

    $controller = new $cname($recurso, $arg, $api);
    if($api) {
      return new APIRestFul($controller->apidata);
      # Crearemos la clase APIRESTFul en un sig. paso
    }
}
```

En el caso de PHP, nuestro *Application* no sufrirá cambios de ningún tipo.

Crear la API a nivel del *core*

No necesitaremos mucho para crear la API. Lo único que se requerirá, será una clase que convierta el objeto retornado por el controlador, en el formato JSON apropiado.

Mientras que en PHP, se dispone de la función `json_enconde()` en Python, contamos con el método `dumps()` del módulo `json`.

Vamos a ver un ejemplo muy simple. Demás está

decir que esta API se podría mejorar muchísimo. Sin embargo, **aquí no entraremos en detalles sobre cómo desarrollar una API, sino que solo tomaremos lo básico para concentrarnos en cómo reutilizar una arquitectura MVC para generar Web Services automáticos en REST y JSON.**

En Python:

```
# Archivo: core/api.py

import json


class APIRESTFul(object):

    def __init__(self, obj):
        self.json_data = '{}'
        if not isinstance(obj, str):
            self.json_data = self.dict2json(vars(obj))

    def obj2dict(self, obj):
        types = (str, int, float, bool,
             tuple, list, dict)
        for propiedad, valor in obj.iteritems():
            notinstance = not isinstance(valor, types)
            if notinstance and not valor is None:
                obj[propiedad] = vars(valor)
                sub = self.obj2dict(obj[propiedad])
                if not obj[propiedad] == sub:
                    obj[propiedad] = sub
        return obj

    def dict2json(self, obj):
        obj = json.dumps(self.obj2dict(obj),
             indent=4, sort_keys=True)
        return '\n'.join(
          [line.rstrip() for line in obj.splitlines()])
```

En PHP:

```
# Archivo: core/api.php

class APIRESTFul {

    function __construct($apidata) {
        header(
          "Content-Type: text/json; charset=utf-8");
        # Para PHP 5.4 (o sup.) usar:
        print json_encode($apidata, JSON_PRETTY_PRINT);
        # Para PHP 5.3, usar:
        print json_encode($apidata);
    }

}
```

Como se puede notar, en Python es necesario convertir los objetos a diccionarios de forma recursiva, antes de presentarlo como JSON.

Modificando los controladores

De entrada, podemos modificar nuestro controlador "madre" para que en él, se disponga de una propiedad que ayude a los recursos a saber si trata o no, de la llamada desde un *Web Service*.

En Python:

```
class Controller(object):

    def __init__(self, recurso, arg, env):
        self.output = '' # para almacenar las vistas
        self.api = env['api']
```

```
        model = self.__class__.__name__.replace(
             'Controller', '')
        view = '%sView' % model
        self.model = model()
        self.view = view()

        metodo = recurso.replace('-', '_')

        if hasattr(self, metodo):
            self.output = getattr(self, metodo)(arg,
               env)
        else:
            self.output = 'Recurso inexistente'
```

En PHP:

```
class Controller {

    function __construct(recurso, arg, $api) {
        if(method_exists($this, $recurso)) {
            $this->api = $api;
            $this->apidata = '';
            call_user_func(array($this, $recurso), $arg);
        } else {
            print 'Recurso inexistente';
        }
    }
}
```

Ahora, solo restará constatar el valor de la propiedad `api` en los recursos públicos donde se quiera habilitar un *Web Service*. Si el valor de la propiedad `api` es `True`, entonces, almacenaremos el objeto en `apidata` (para PHP) y el variable de salida para Python. Caso contrario, invocaremos a la vista. Veamos un ejemplo abstracto.

En Python:

```
class ObjetoController(Controller):

    def ver(self, id=0, env):
        self.model.objeto_id = id
        self.model.get()
        if self.api:
            return self.model
        else:
            return self.view.mostrar(self.model)
```

En PHP:

```
class ObjetoController extends Controller {

    function ver($id=0) {
        $this->model->objeto_id = $id;
        $this->model->get();
        if($this->api) {
            $this->apidata = $this->model;
        } else {
            $this->view->mostrar($this->model);
        }
    }

}
```

Cuando un recurso no quiera habilitar un servicio Web, simplemente, seguirá funcionando como hasta antes de crear la API, sin requerir de ningún cambio.